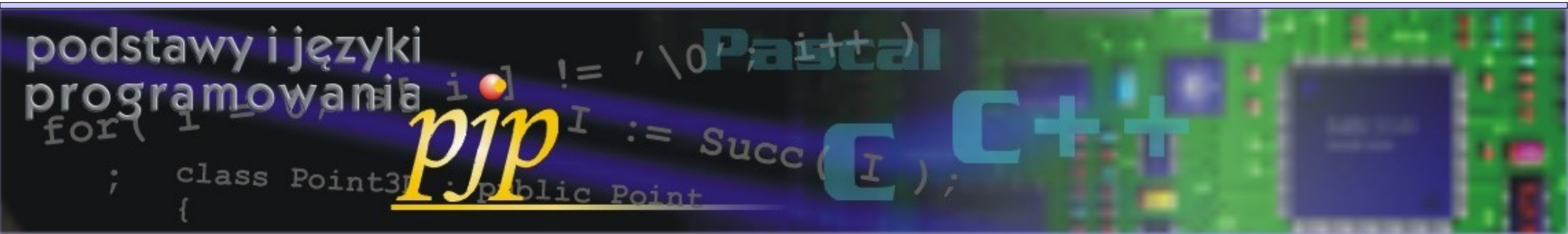
Programowanie obiektowe w języku C++

Roman Simiński
roman.siminski@us.edu.pl
www.siminskionline.pl

Dziedziczenie — wprowadzenie



Ponowne wykorzystanie elementów oprogramowania

- W trakcie projektowania i realizacji systemów informatycznych wielokrotnie powtarzają się *jednakowe* lub *podobne* elementy oprogramowania.
- Powtarzalność taką można obserwować zarówno etapie analizy, projektowania jak i programowania.
- Na etapie analizy są to zwykle *obiekty dziedziny problemu*, na etapie projektowania i programowania zarówno obiekty *dziedziny problemu* jak i *obiekty implementacyjne*.

Ponowne wykorzystanie (ang. *reuse*) pozwala:

- na zwiększenie wydajności procesu programowania,
- poprawia jakość i niezawodność kodu,
- ułatwia rozwój i pielęgnację oprogramowania.

Dziedziczenie
Koncepcja

Ponowne wykorzystanie kodu w programowaniu obiektowym

- Podejście obiektowe eksponuje rolę obiektu – jest on abstrakcją pewnego konkretnego bytu ze świata rzeczywistego, reprezentuje *rzecz* (obiekt fizyczny) lub *pojęcie* (obiekt konceptualny).

- Obiekt może być związany z *dziedziną problemu* (*osoba*, *auto*, *student*, *faktura*), ze *środowiskiem programu* (*dysk*, *plik*, *folder*, *serwer*, *pakiet*) lub mieć charakter *implementacyjny* (*lista*, *wątek*, *okno*, *przycisk*).

Klasa: Auto

Cechy

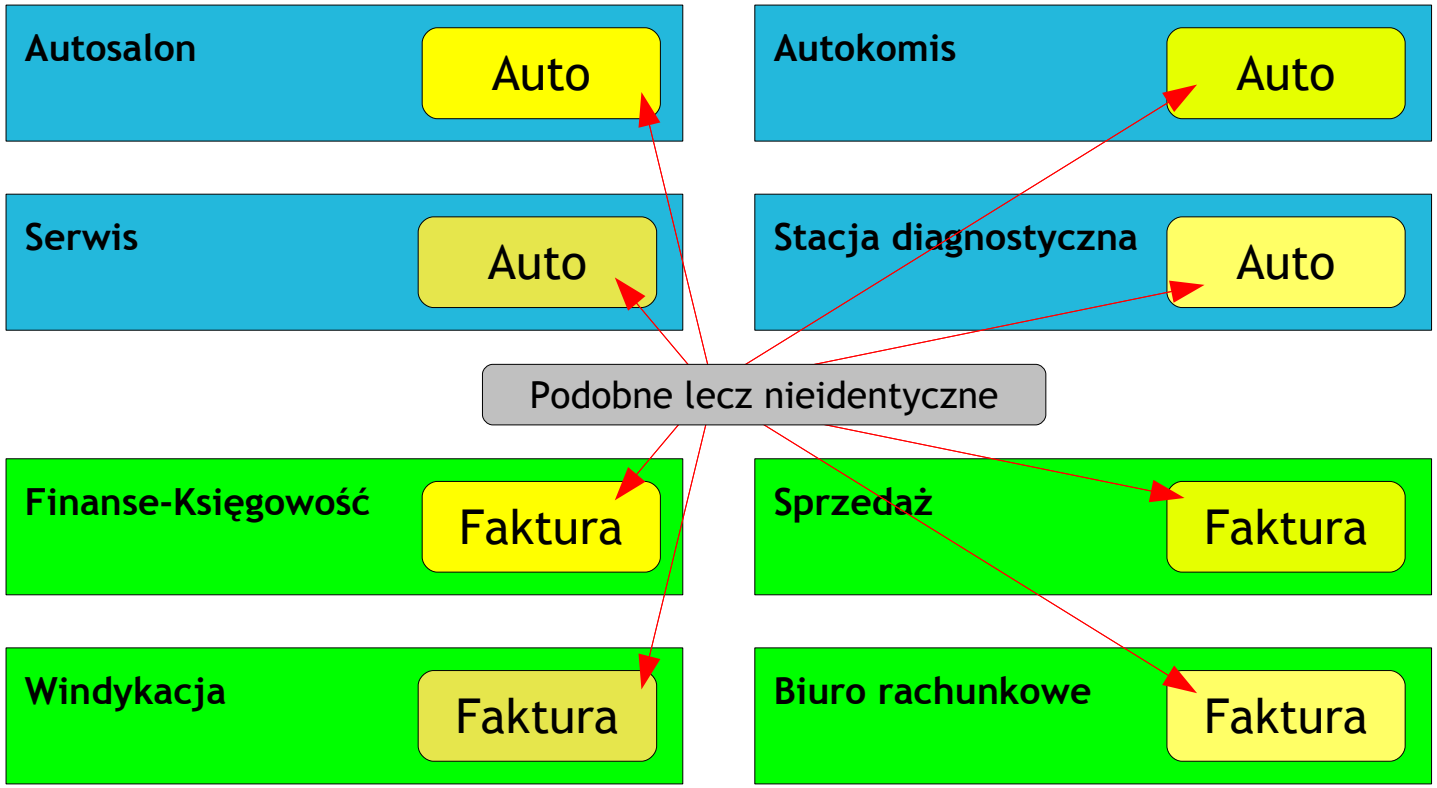
Marka
Model
Rok produkcji
Nr rejestracyjny
Kolor nadwozia
Nr silnika
. . .

Akcje

Podaj średnie spalanie
Podaj przebieg
. . .

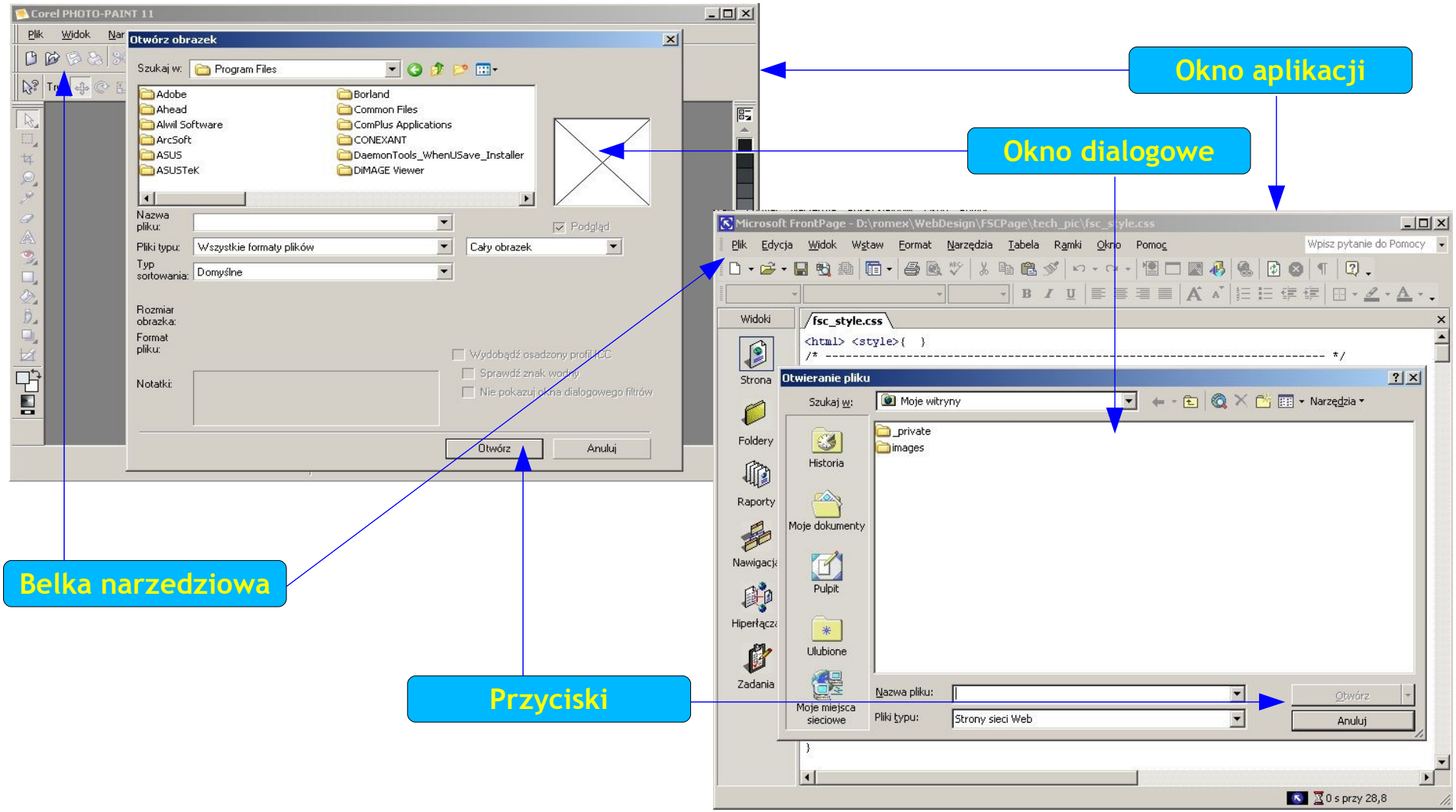
Podobne ale nie identyczne

- W programowaniu obiektowym *ponowne wykorzystanie kodu* polega na *wielokrotnym wykorzystaniu obiektów* pewnej, raz zdefiniowanej klasy, w różnych projektach.
- Niestety..., obrębie obiektów dziedziny problemu *powtarzają się obiekty podobne* lecz *bardzo rzadko identyczne*.



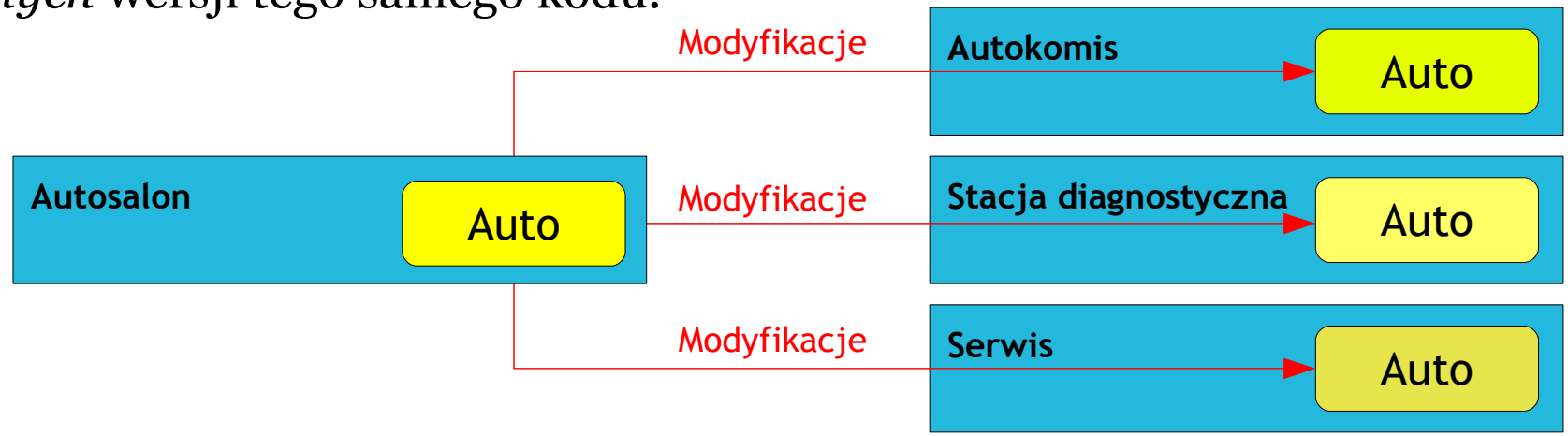
Podobne ale nie identyczne

- W obrębie obiektów *implementacyjnych* powtarzalność obiektów identycznych jest znacznie częstsza, nie jest jednak to regułą.



Ponowne wykorzystanie kodu — problemy

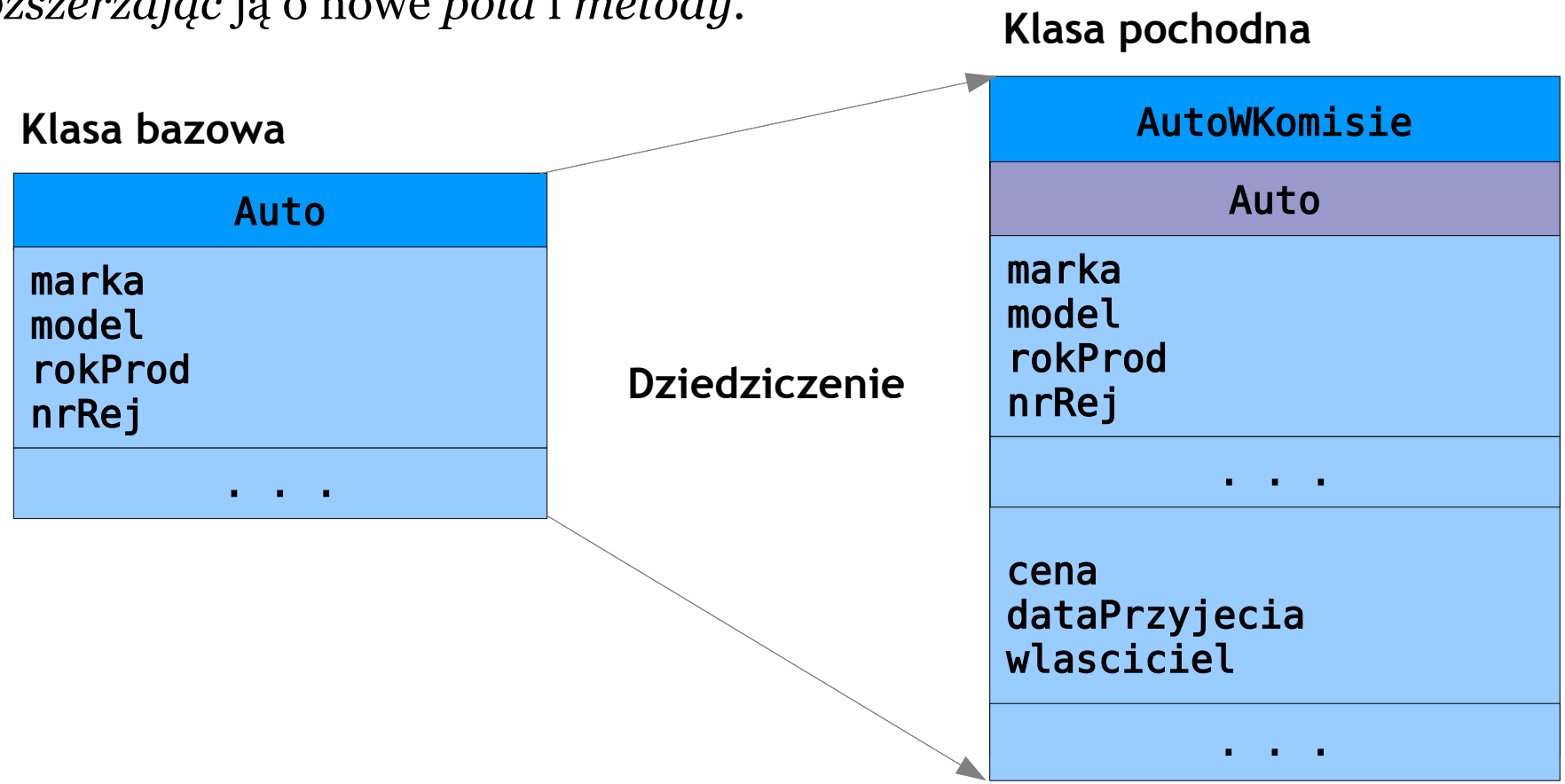
- Powtarzalność nieidentycznych obiektów powoduje to, że musimy *modyfikować istniejący kod*, dostosowując go do *specyfiki projektu*.
- Modyfikacje potrzebne z punktu widzenia jednego systemu mogą być *nieużyteczne* lub *nieakceptowalne* z punktu widzenia innego systemu.
- Modyfikacje dostosowujące kod do specyfiki systemu zmuszają do *utrzymywania różnych* wersji tego samego kodu.



Konieczność wprowadzenia w kodzie zmian o *charakterze podstawowym*, może spowodować konieczność zmodyfikowania *wielu różnych jego wersji*. Jest to uciążliwe, nieproduktywne i może być przyczyną błędów.

Ponowne wykorzystanie kodu z wykorzystaniem dziedziczenia

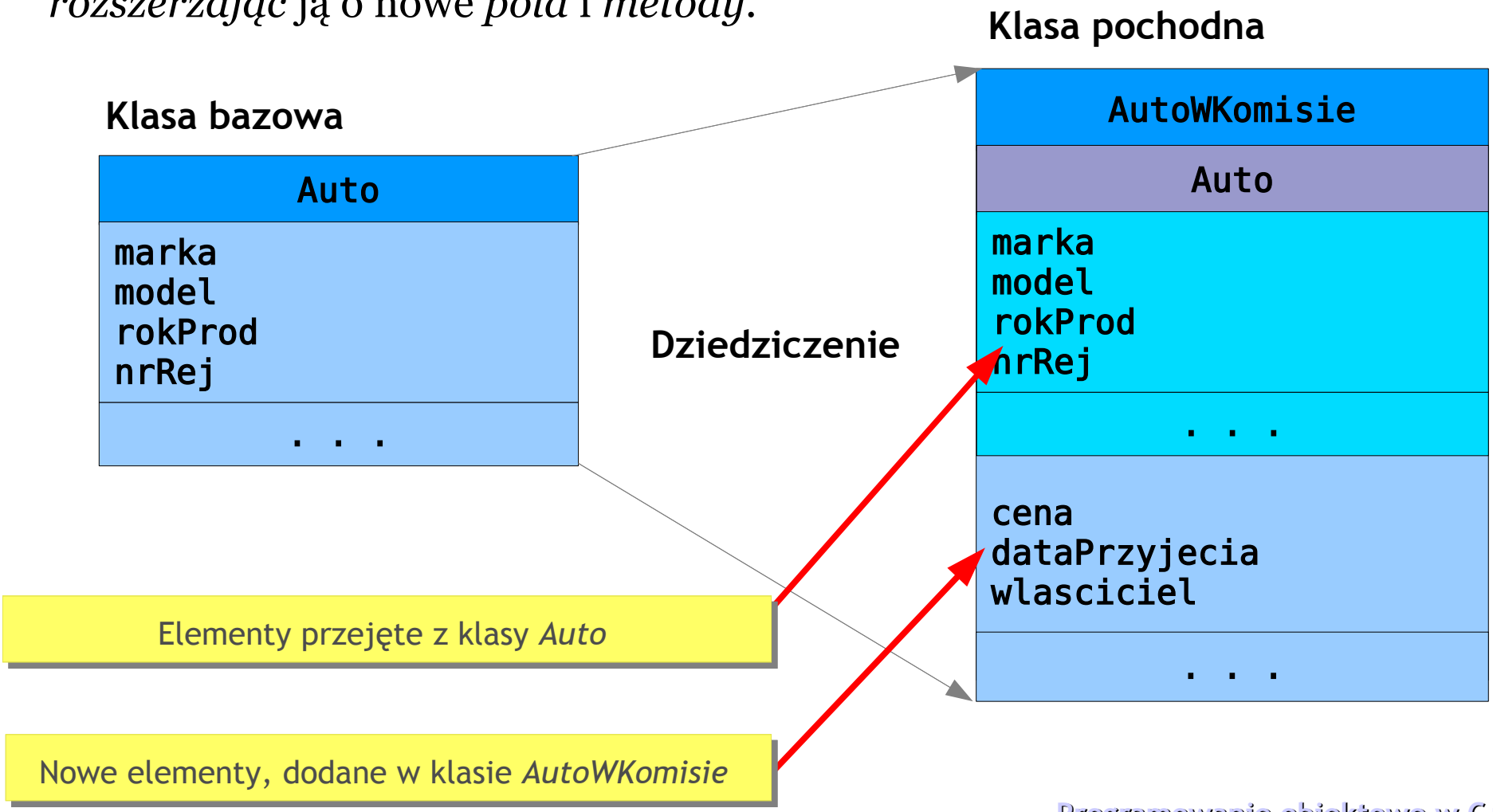
- ***Dziedziczenie*** polega na budowaniu *nowych klas*, zwanych klasami *potomnymi*, na podstawie klas już istniejących, zwanych klasami *bazowymi*.
- Każda klasa *pochodna* ***dziedziczy*** wszystkie właściwości klasy bazowej, być może *rozszerzając* ją o nowe *pola* i *metody*.



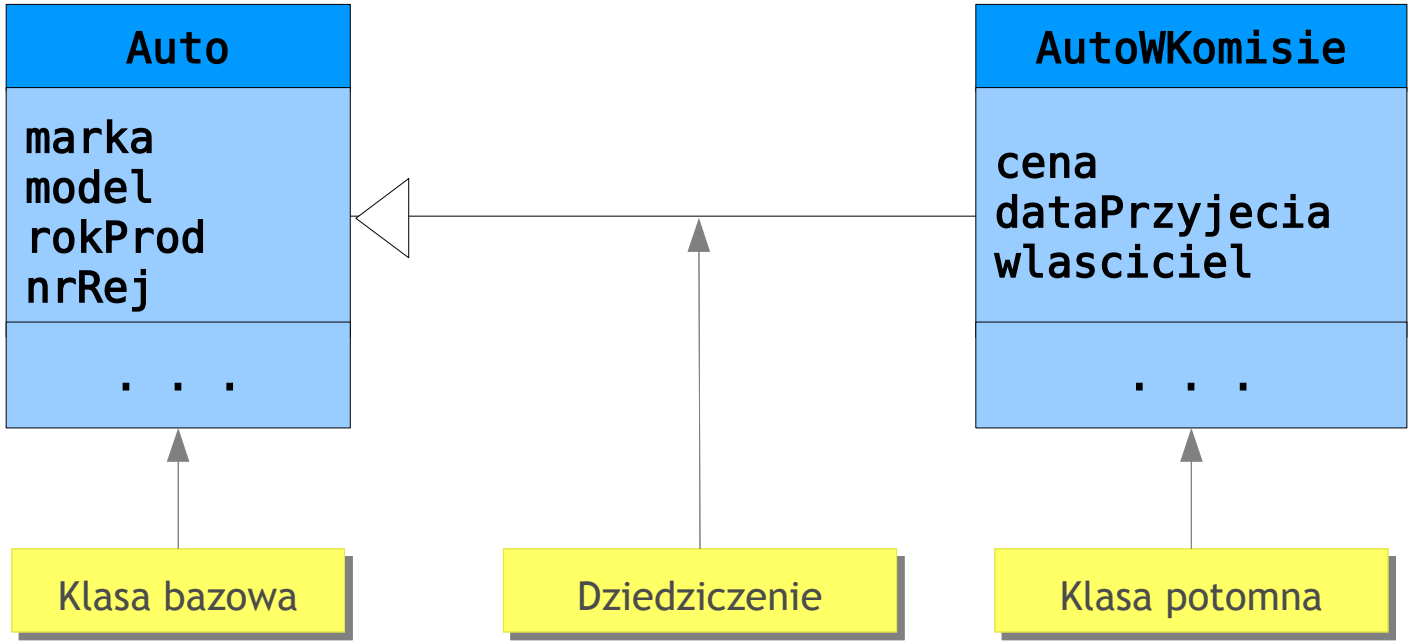
Uwaga — chwilowo na przedstawianych diagramach klas nie będą stosowane oznaczenia widoczności pól: (+) dla pól publicznych i (-) dla pól prywatnych..

Ponowne wykorzystanie kodu z wykorzystaniem dziedziczenia

- ***Dziedziczenie*** polega na budowaniu *nowych klas*, zwanych klasami *potomnymi*, na podstawie klas już istniejących, zwanych klasami *bazowymi*.
- Każda klasa *pochodna* ***dziedziczy*** wszystkie właściwości klasy bazowej, być może *rozszerzając* ją o nowe *pola* i *metody*.

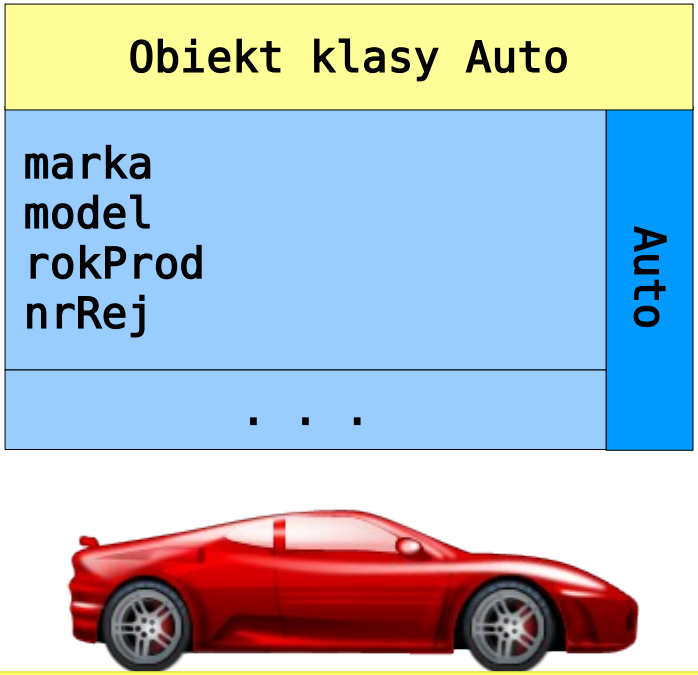


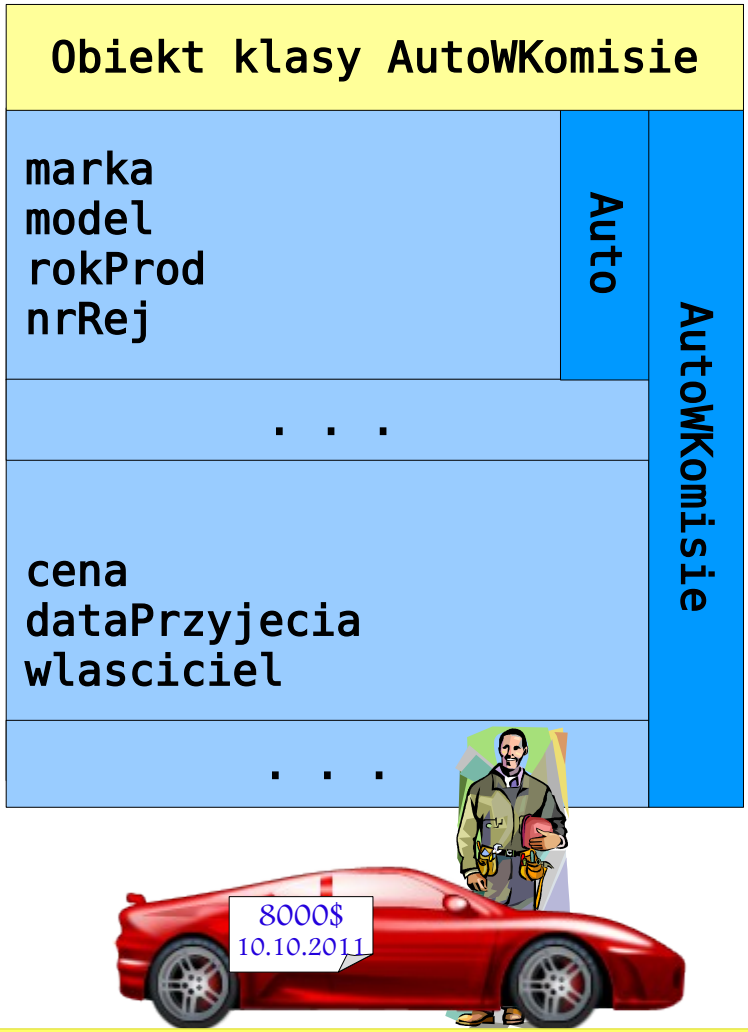
Klasa Auto jako klasa bazowa klasy AutoWKomisie – notacja UML



Obiekt klasy pochodnej otrzymuje wszystkie pola i metody klasy bazowej.
100% elementów klasy bazowej występuje w klasie pochodnej.

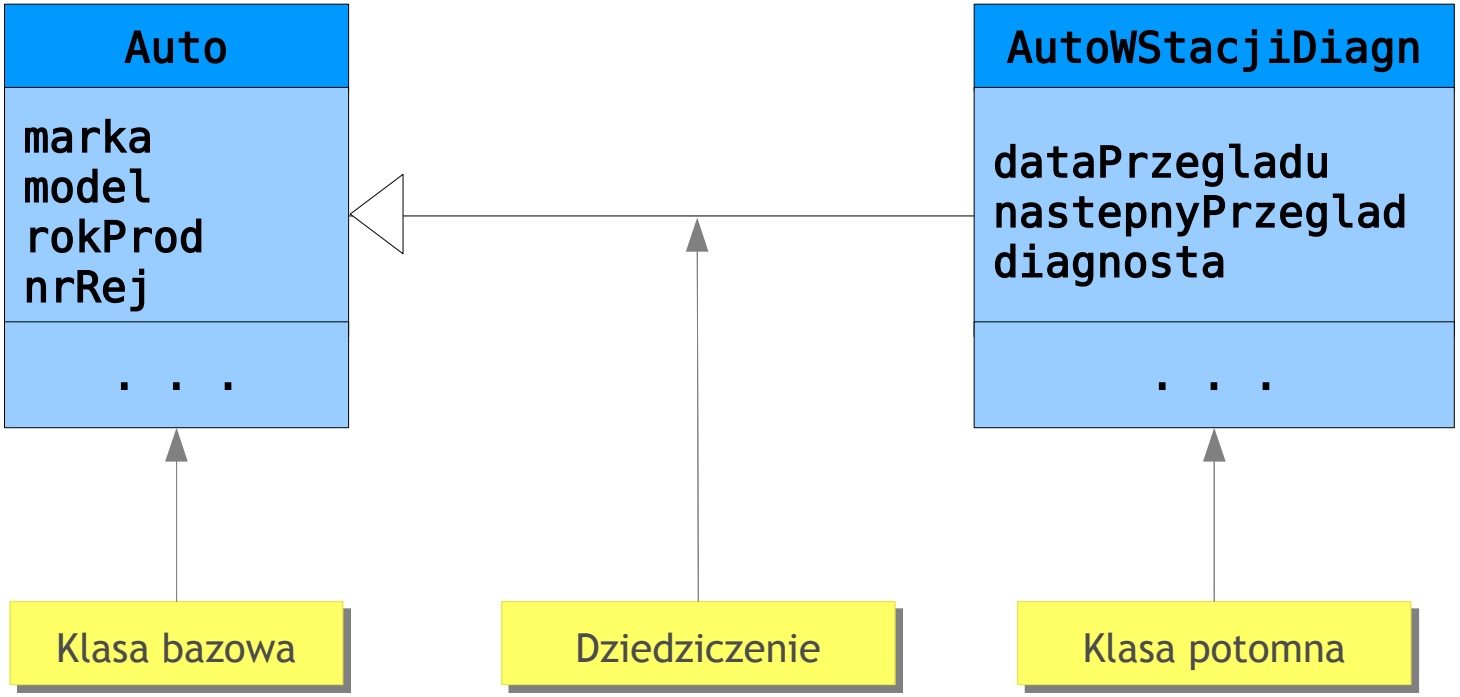
Obiekt klasy pochodnej a obiekt klasy bazowej



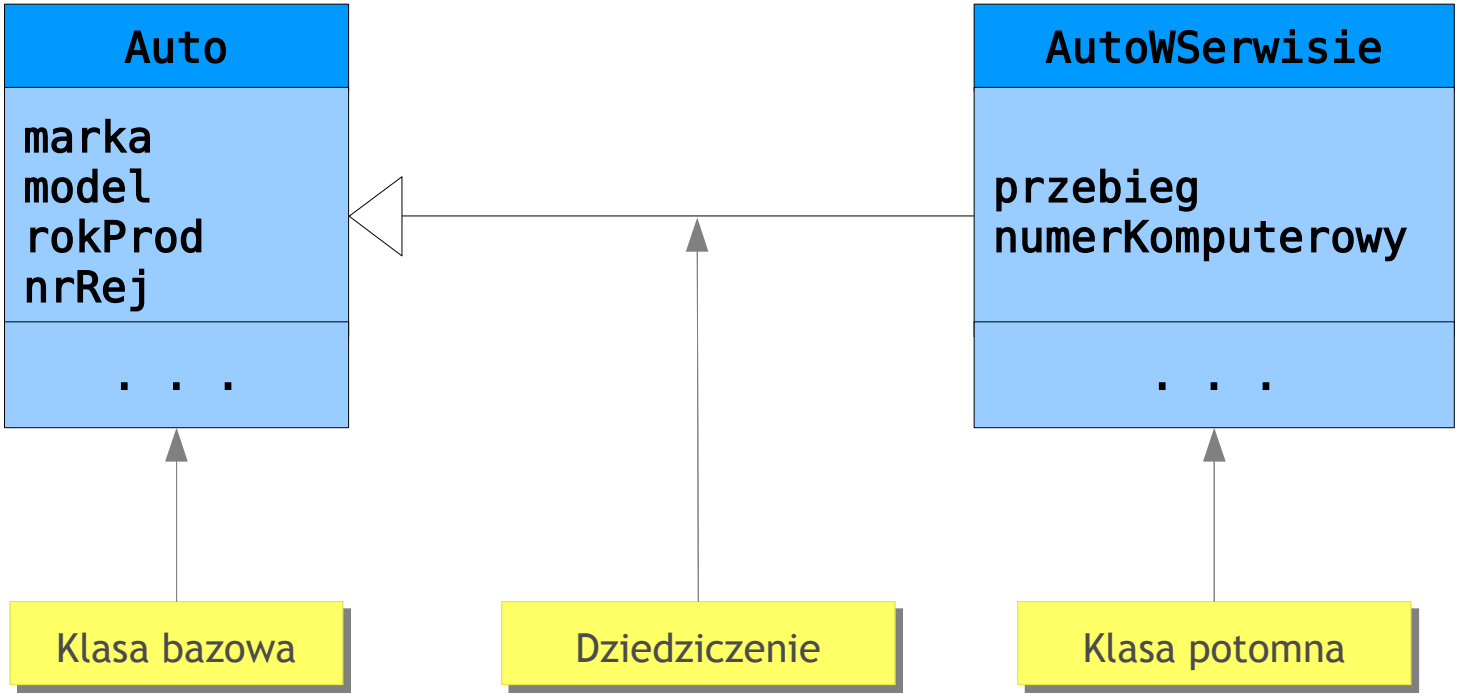


Dziedziczenie pozwala na rozszerzanie klasy o *nowe informacje* i/lub *nowe zachowanie*. Można również *zmieniać* dotychczasowe *zachowanie*.

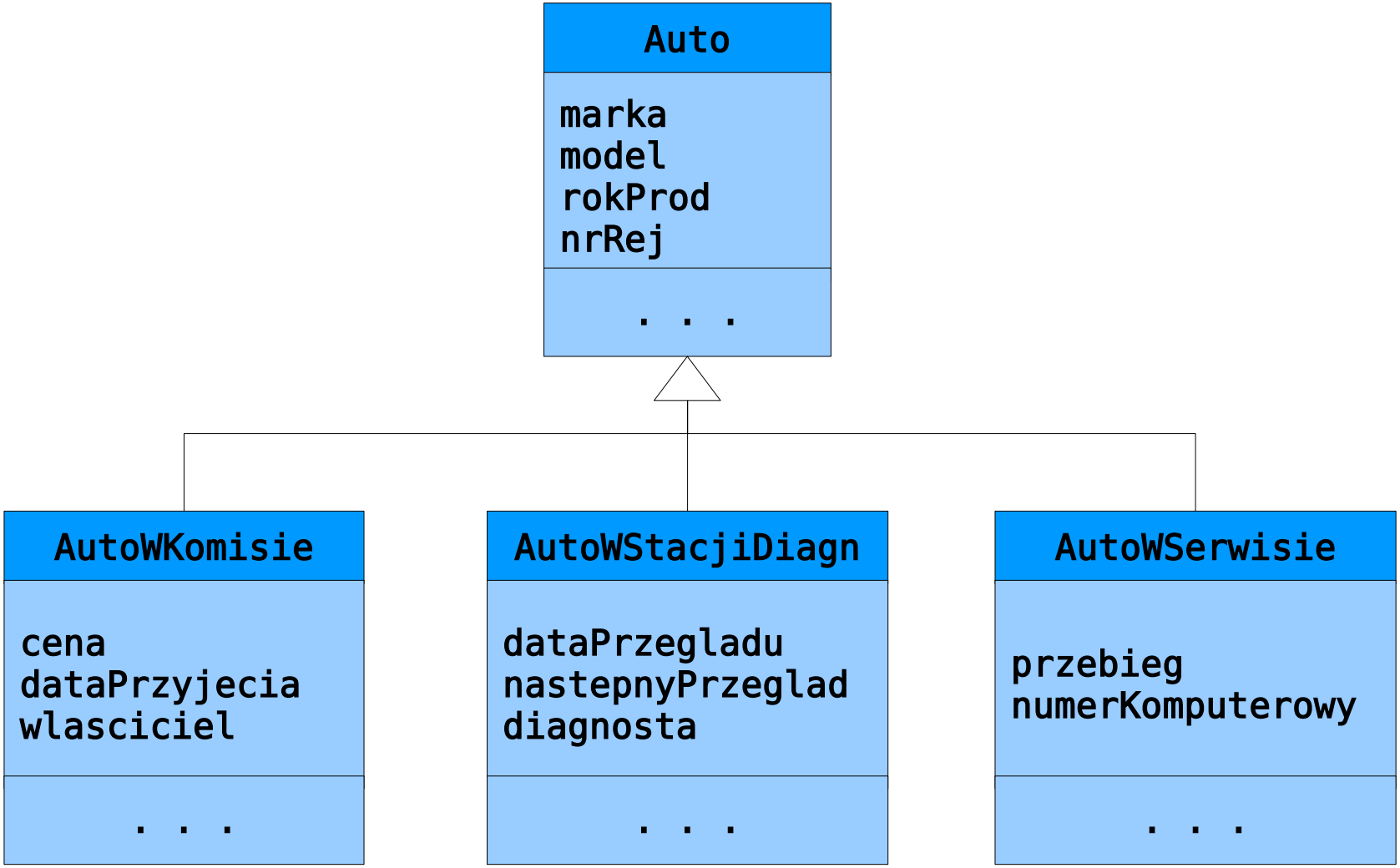
Klasa Auto jako klasa bazowa klasy AutoWStacjiDiagn



Klasa Auto jako klasa bazowa klasy AutoWSerwisie

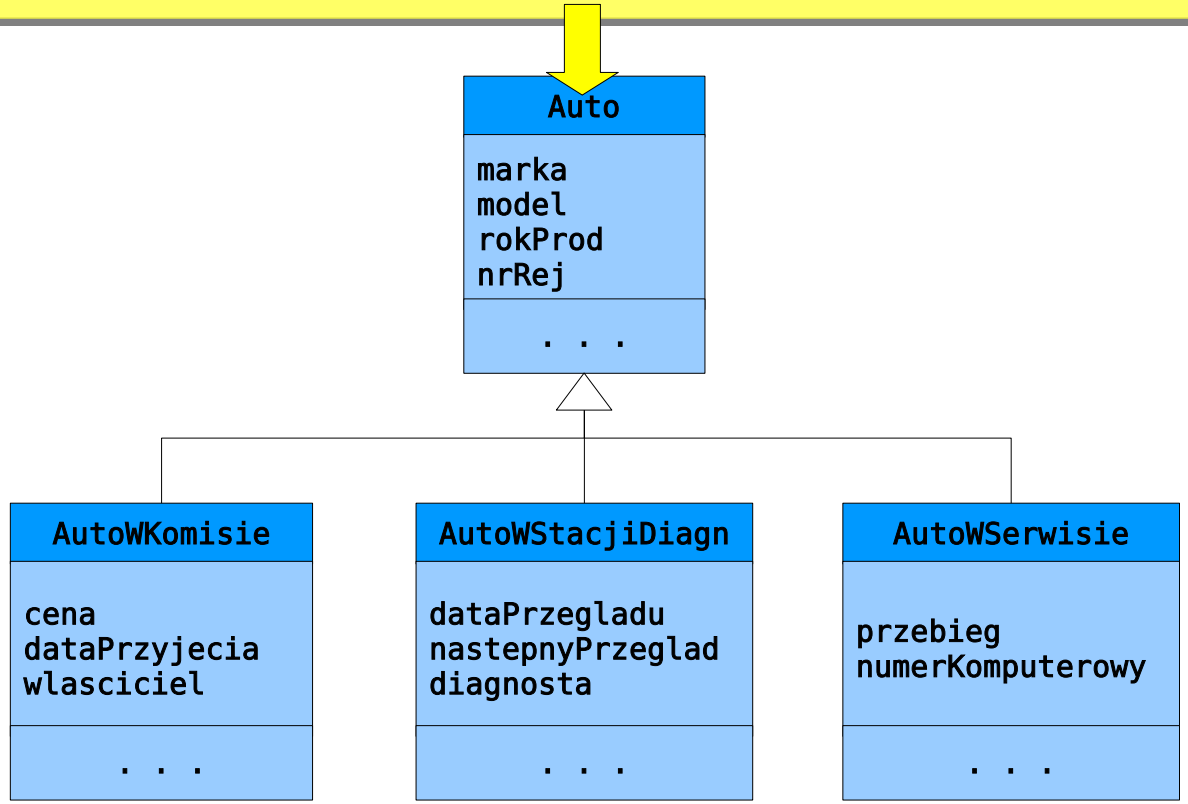


Klasa bazowa i klasy potomne tworzą hierarchię klas

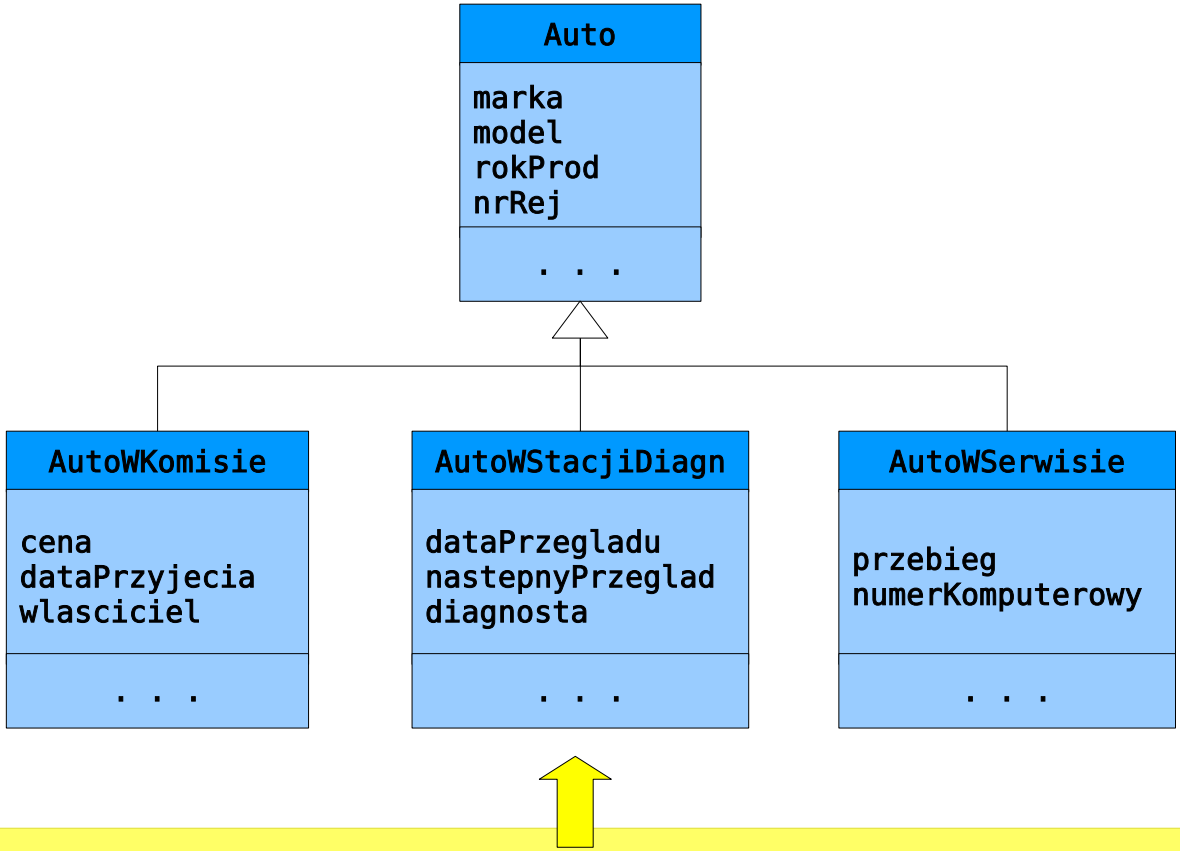


Klasa bazowa jako klasa ogólna

Klasa *bazowa* (*nadklasa*) reprezentuje pojęcie bardziej *ogólne*, znajdujące się na *wyższym poziomie abstrakcji*.



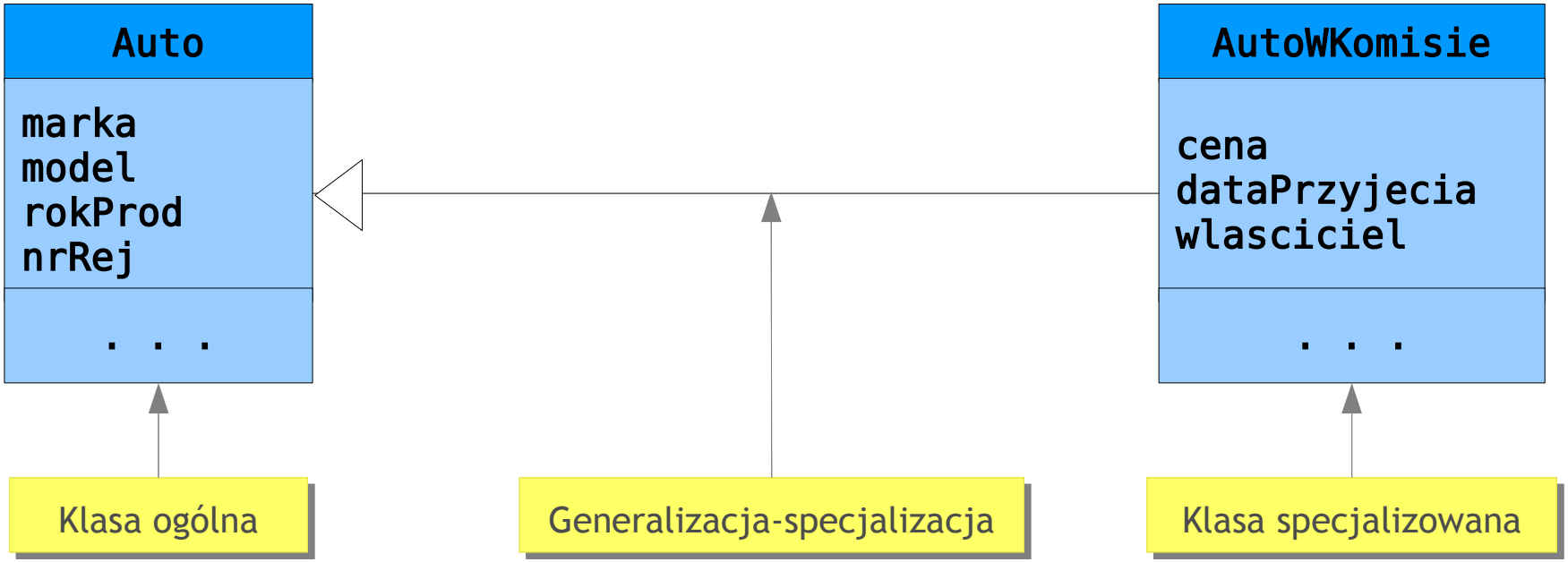
Klasa bazowa jako klasa ogólna



Klasy *pochodne* (*podklasay*) reprezentuje pojęcie bardziej *szczegółowe*, *specjalizowane*, znajdujące się na *niższym poziomie abstrakcji*.

Dziedziczenie jako środek realizacji modeli generalizacja-specjalizacja

Pojęcie specjalizacja-generalizacja umożliwia organizowanie klas w struktury hierarchiczne, w których *nadklasa* reprezentuje pojęcia *bardziej ogólne*, a *podklasa* pojęcia *specjalizowane*.



Większa Ogólność i wyższy poziom abstrakcji

Większa szczegółowość i niższy poziom abstrakcji

Jak zweryfikować poprawność wykorzystania dziedziczenia?

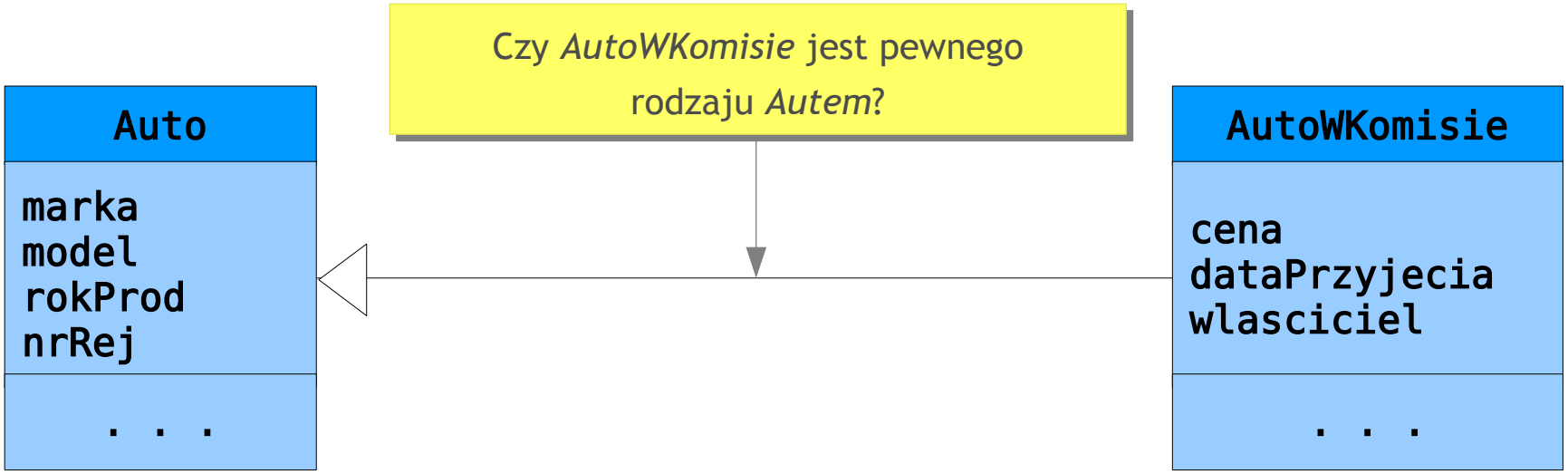
Reguła ***is-a*** (ang. *is a kind of*) pozwala na sprawdzenie czy zachodzi związek dziedziczenia (specjalizacji–generalizacji) pomiędzy klasami.

Realizowane jest to poprzez sprawdzenie poprawności zdania:

- Czy *klasa pochodna* jest pewnego rodzaju (*is-a*) *klasą bazową*?

precyzyjniej

- Czy *obiekt klasy pochodnej* jest pewnego rodzaju (*is-a*) *obiektem klasy bazowej*?



Przykład weryfikacji poprawności dziedziczenia

Zadajemy dwa pytania:

- czy obiekt klasy *A* jest obiektem klasy *B*? (*A is-a B*?)
- czy obiekt klasy *B* jest obiektem klasy *A*? (*B is-a A*?)

Możliwe odpowiedzi:

- zawsze,
- nigdy
- czasami.

Interpretacja:

- *dwie odpowiedzi nigdy*: brak związku specjalizacja-generalizacja,
- *dwie odpowiedzi zawsze*: obiekty *A* i *B* są ***synonimiczne*** (np. *dwie różne nazwy* dla tej *samej klasy*)
- jeżeli: *A is-a B* = *zawsze* oraz *B is-a A* = *czasami* wtedy ***A jest specjalizacją B*** (*A is-a B*).

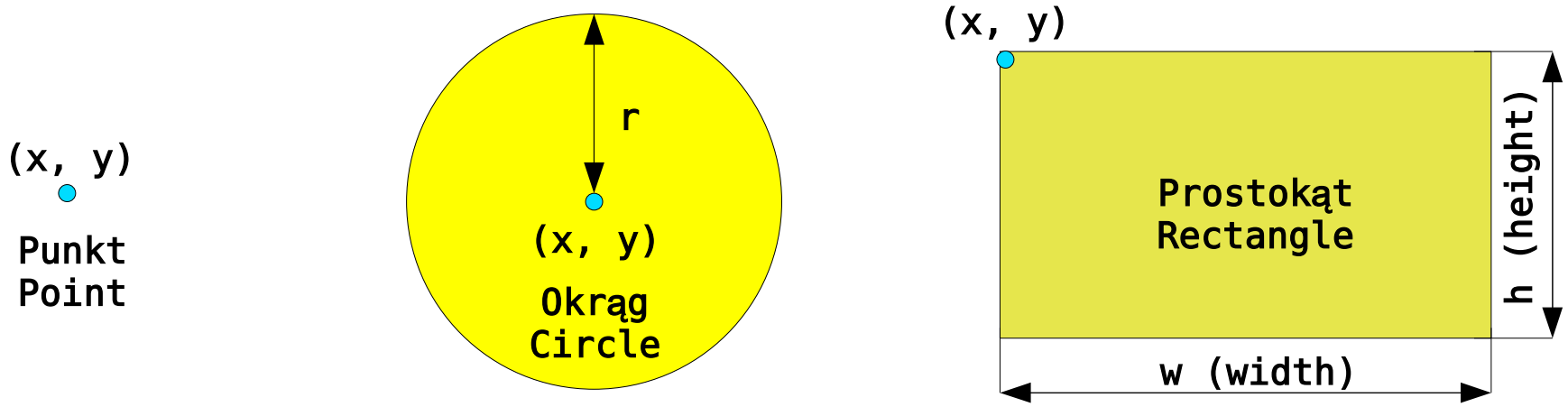
Przykład weryfikacji poprawności dziedziczenia

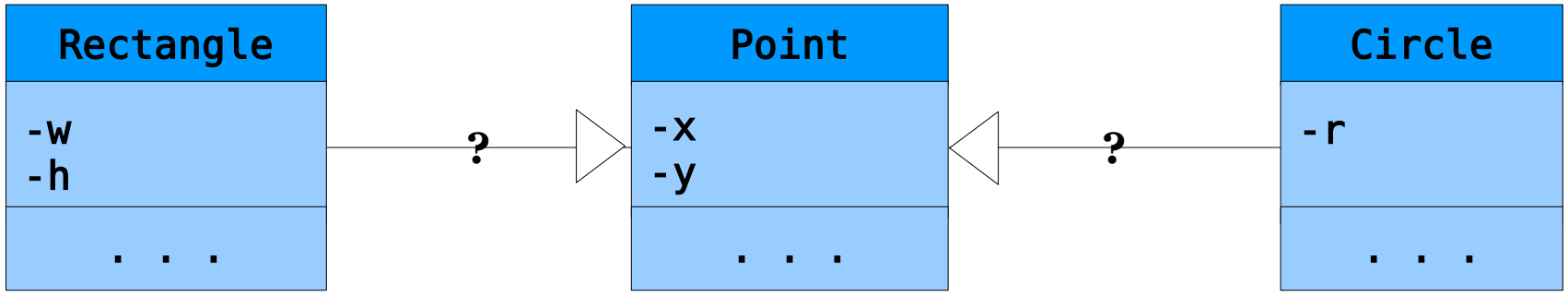
Czy klasa **`AutoWKomisie`** jest klasą pochodną klasy **`Auto`**:

- czy obiekt klasy **`AutoWKomisie`** jest obiektem klasy **`Auto`**? ← **Zawsze**
- czy obiekt klasy **`Auto`** jest obiektem klasy **`AutoWKomisie`**? ← **Czasami**

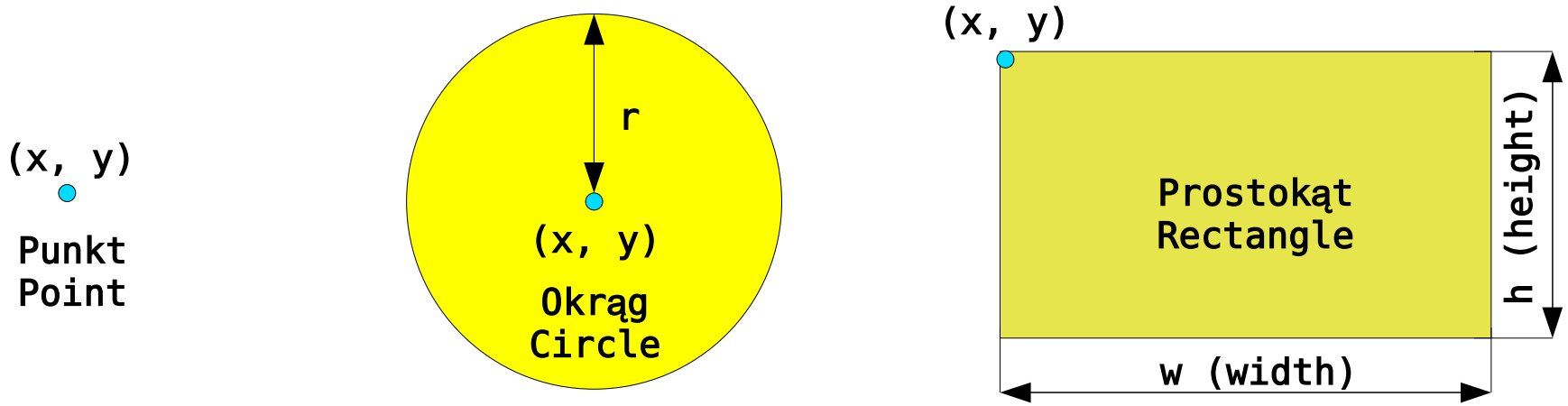
A jak ten test przejdą standardowe przykłady dziedziczenia z popularnych książek o programowaniu obiektowym?

Dziedziczenie prosto z niejednej książki





Dziedziczenie prosto z niejednej książki



Czy klasa **`Circle`** jest klasą pochodną klasy **`Point`**:

- czy obiekt klasy **`Circle`** jest pewnego rodzaju obiektem klasy **`Point`**? ← **?**
- czy obiekt klasy **`Point`** jest pewnego rodzaju obiektem klasy **`Circle`**? ← **?**

Czy klasa **`Rectangle`** jest klasą pochodną klasy **`Point`**:

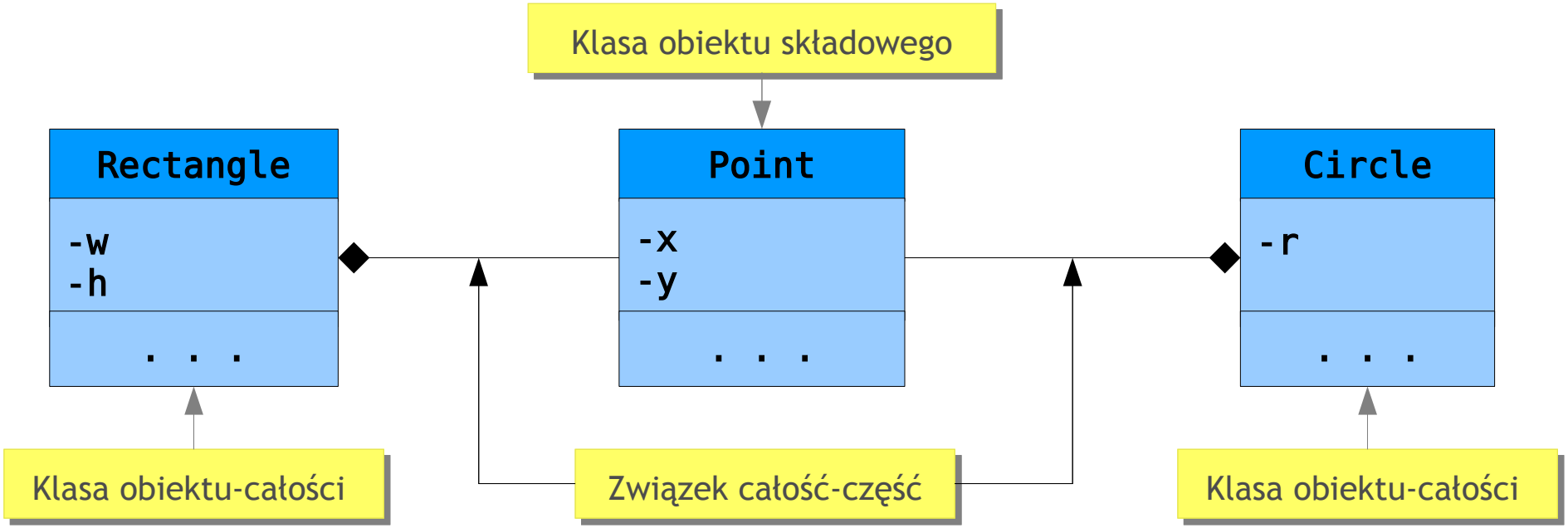
- czy obiekt klasy **`Rectangle`** jest pewnego rodzaju obiektem klasy **`Point`**? ← **?**
- czy obiekt klasy **`Point`** jest pewnego rodzaju obiektem klasy **`Rectangle`**? ← **?**

Koncepcja dziedziczenia a związki całość-część

Często obiekty łączy relacja ***is-part*** czyli *jest częścią* (ang. *is a part of*). Relacja ta określana jest pojęciem *całość-część*.

Przy związkach całość-część zadajemy pytania:

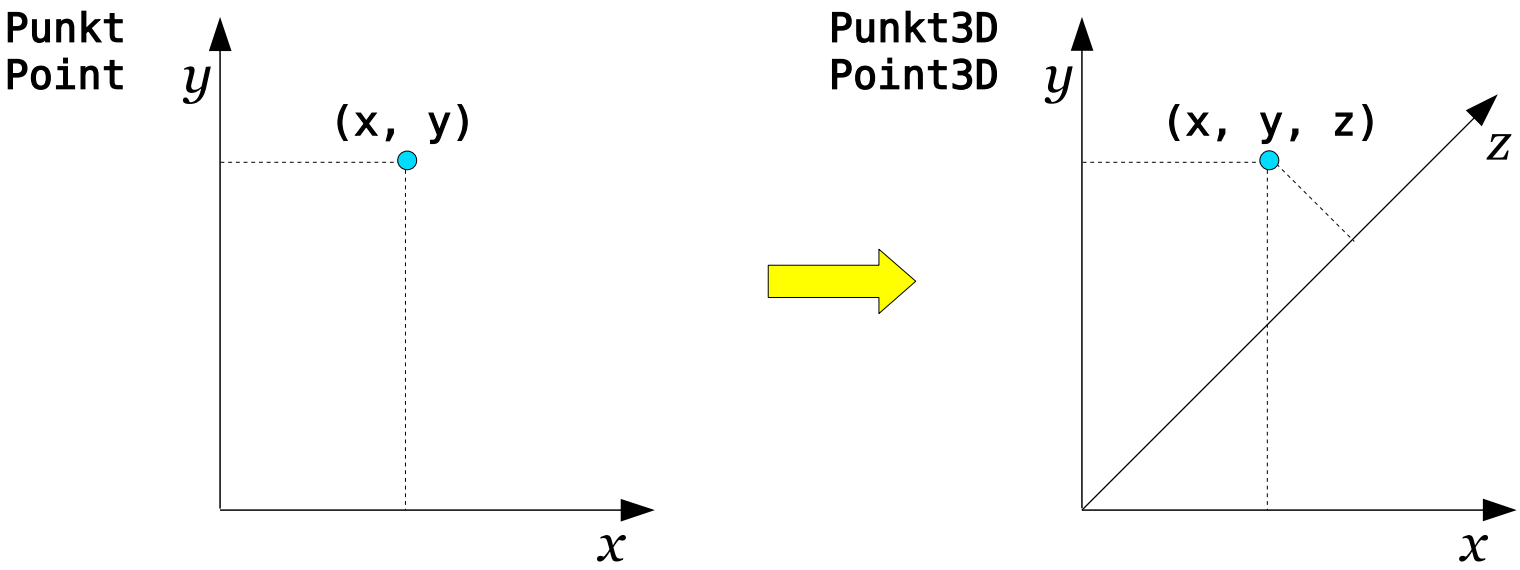
- czy obiekt klasy **A** jest częścią obiektu klasy **B**?
- czy obiekt klasy **B** zawiera w sobie obiekt klasy **A**?



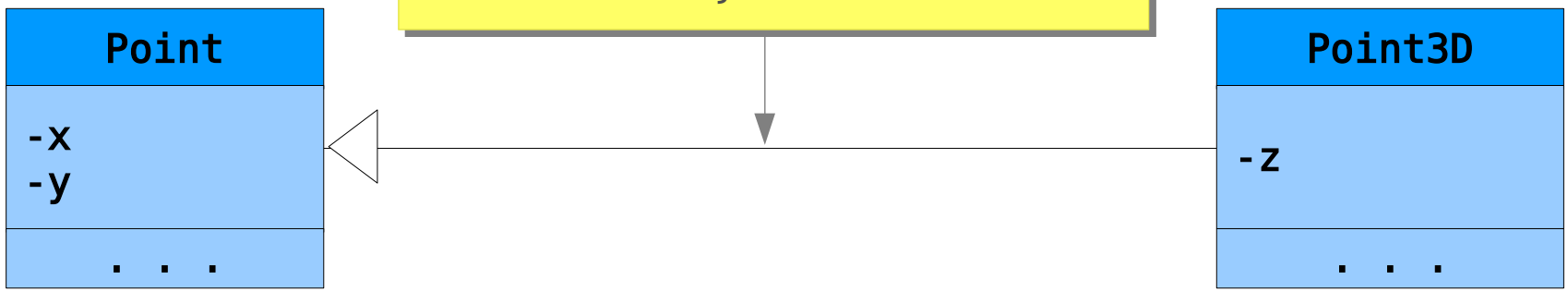
Związki całość-część są bardzo istotne i do nich jeszcze wrócimy

Anatomia dziedziczenia w języku C++

Dziedziczenie w C++ — przykład



Czy *Punkt3D* jest pewnego rodzaju *Punktem*?



Przykład wykorzystania klasy Point i Point3D:

```
#include <iostream>
using namespace std;

// Definicje klas Point i Point3D

int main()
{
  Point    p1( 10, 12 );
  Point3D  p2( 2, 5, 4 );

  cout << endl << "Punkt 2D ma wspolrzedne" << endl;
  cout << "x = " << p1.getX() << endl;
  cout << "y = " << p1.getY() << endl;

  cout << endl << "Punkt 3D ma wspolrzedne" << endl;
  cout << "x = " << p2.getX() << endl;
  cout << "y = " << p2.getY() << endl;
  cout << "z = " << p2.getZ() << endl;

  return EXIT_SUCCESS;
}
```

Obiekt p2 posiada odziedziczone pola *x* i *y* oraz funkcje składowe klasy *Point*.

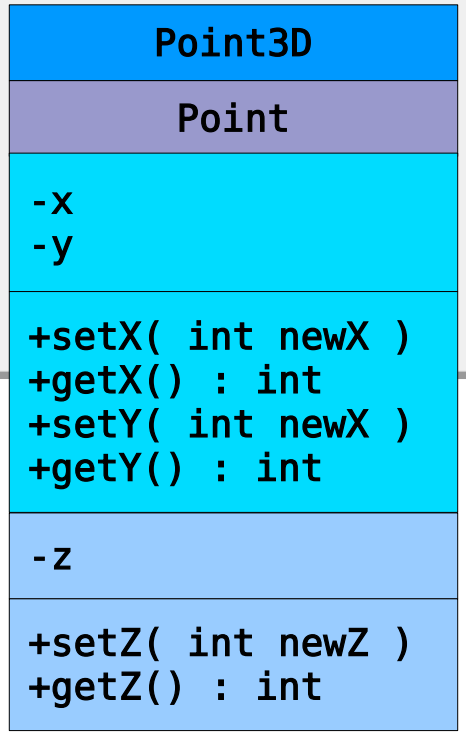
Przykład klasy bazowej Point

```
class Point
{
  public:

    // Konstruktor – ogólny i domyślny
    Point( int startX = 0, int startY = 0 )
    : x( startX ), y( startY )
    {
    }

    // Modyfikator i akcesor dla pola x
    void setX( int newX ) { x = newX; }
    int  getX() const { return x; }

    // Modyfikator i akcesor dla pola y
    void setY( int newY ) { y = newY; }
    int  getY() const { return y; }

  private:
    int  x, y;
};
```

Dziedziczenia – od klasy Point do Point3D

```
class Point3d: public Point
{
  public:

    // Konstruktor – ogólny i domyślny
    Point3D( int startX = 0, int startY = 0, int startZ = 0 )
    : Point( startX, startY ), z( startZ )
    {
    }

    // Modyfikator i akcesor dla pola z
    void setZ( int newZ ) { x = newZ; }
    int  getZ() const { return z; }

  private:
    int  z;
};
```

Dziedziczenie – klasa *Point3D* powstaje na bazie klasy *Point*, dziedzicząc wszystkie jej elementy.



Występuje tutaj dziedziczenie w trybie *publicznym* – zachowuje ono *dotychczasową widoczność elementów określoną w klasie bazowej*. Co było prywatne, prywatnym pozostaje, co było publiczne, jest publiczne dalej.

Anatomia klasy pochodnej Point3D

```
class Point3d: public Point
{
  public:

    // Konstruktor – ogólny i domyślny
    Point3D( int startX = 0, int startY = 0, int startZ = 0 )
    : Point( startX, startY ), z( startZ )
    {
    }

    // Modyfikator i akcesor dla pola z
    void setZ( int newZ ) { x = newZ; }
    int  getZ() const { return z; }

  private:
    int  z;
};
```

Własny konstruktor klasy Point3D.

Nowe funkcje składowe klasy Point3D

Nowe pole klasy Point3D

Lista inicjalizacyjna raz jeszcze...

```
class Point3d: public Point
{
  public:

    // Konstruktor – ogólny i domyślny
    Point3D( int startX = 0, int startY = 0, int startZ = 0 )
    : Point( startX, startY ), z( startZ )
    {
    }

    // Modyfikator i akcesor dla pola z
    void setZ( int newZ ) { x = newZ; }
    int  getZ() const { return z; }

  private:
    int  z;
};
```

```
Point( startX, startY ) , z( startZ )
```

Konstruktor(y) klasy(klas) bazowych

Inicjatory pól obiektu rozważanej klasy

Konstruktor klasy Point a konstruktor klasy pochodnej Point3D

```
class Point3d: public Point
{
  public:

    // Konstruktor – ogólny i domyślny
    Point3D( int startX = 0, int startY = 0, int startZ = 0 )
    : Point( startX, startY ),
    {
    }

    // Modyfikator i akcesor d
    void setZ( int newZ ) { x
    int  getZ() const { returr

  private:
    int  z;
};
```

```
class Point
{
  public:

    // Konstruktor – ogólny i domyślny
    Point( int startX = 0, int startY = 0 )
    : x( startX ), y( startY )
    {
    }

    . . .
};
```

Konstruktor klasy pochodnej *aktywuje* konstruktor klasy bazowej *umieszczony* na *liście inicjalizacyjnej*, odbywa się to przed wywołaniem ciała konstruktora klasy pochodnej.

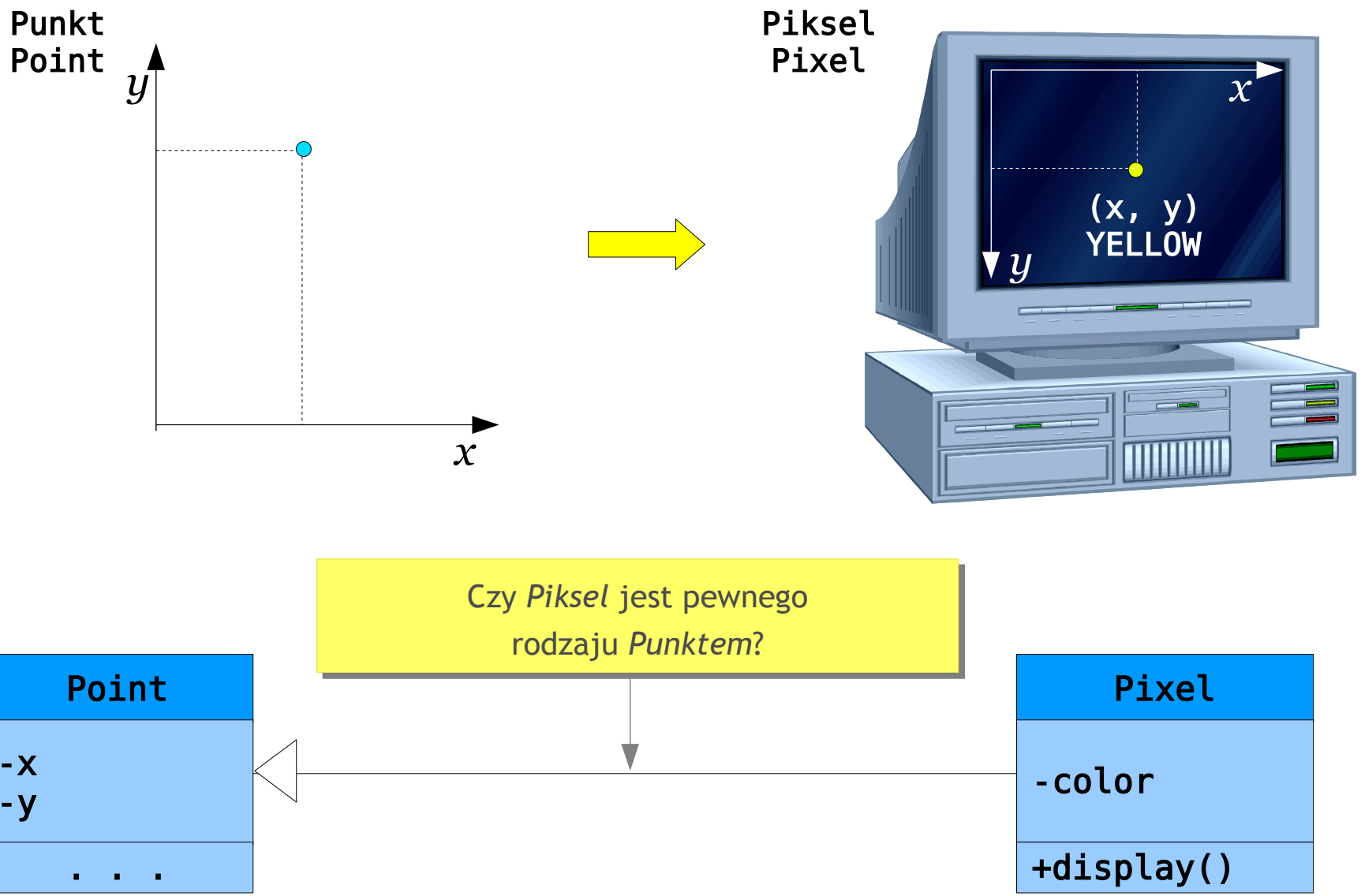
Konstruktor klasy `Point` a konstruktor klasy pochodnej `Point3D`

```
class Point3d: public Point
{
  public:

    // Konstruktor – ogólny i domyślny
    Point3D( int startX = 0, int startY = 0, int startZ = 0 )
    : Point( startX, startY ), z( startZ )
    {
    }

    // Modyfikator i akcesor dla pola z
    void setZ( int newZ ) { x = newZ; }
    int  getZ() const { return z; }

  private:
    int  z;
};
```

Używamy konstruktorów klasy bazowej do zainicjowania pól odziedziczonych, *nowe* pola inicjuje *własny* konstruktor.

W klasie pochodnej dążymy do maksymalnego wykorzystania elementów klasy bazowej w zakresie „obsługi” elementów po tej klasie odziedziczonych.

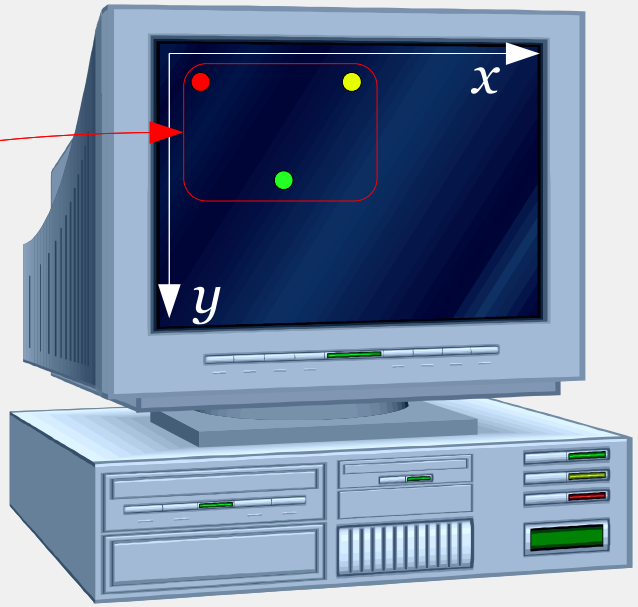
Inny przykład dziedziczenia — od klasy `Point` do klasy `Pixel`



Uwaga — z punktu widzenia praktyki programowania klasa *Pixel* jest dyskusyjna z powodu potencjalnie nikłej użyteczności praktycznej. Proszę ją potraktować jako przykład wykładowy.

Przykład wykorzystania klasy Pixel

```
#include <iostream>
using namespace std;

// Definicje klas Point i Pixel

int main()
{
  Pixel p1( 10,  10,  RED );
  Pixel p2( 100, 10,  YELLOW );
  Pixel p3( 55,  100, GREEN );

  p1.display();
  p2.display();
  p3.display();

  return EXIT_SUCCESS;
}
```



Przykład klasy pochodnej Pixel

```
class Pixel : public Point
{
  public :
    // Konstruktor
    Pixel( int startX = 0, int startY = 0, int startColor = 0 )
    : Point( startX, startY ), color( startColor )
    {
    }

    // Modyfikator i akcesor dla pola color
    void setColor( int newColor ) { color = newColor; }
    int  getColor() const { return color; }

    // Realizator
    void display() const
    {
      // Funkcja wyświetl. piksela z odpowiedniej bibl.
      // graficznej, np. BGI:
      putpixel( getX(), getY(), color );
    }

  private:
    int color;
};
```

Dziedziczenie w trybie publicznym

Własny konstruktor

Nowe funkcje składowe i nowe pole

Przykład klasy pochodnej Pixel

```
class Pixel : public Point
{
  public :
    // Konstruktor
    Pixel( int startX = 0, int startY = 0, int startColor = 0 )
    : Point( startX, startY ), color( startColor )
    {
    }

    // Modyfikator i akcesor dla pola color
    void setColor( int newColor ) { color = newColor; }
    int  getColor() const { return color; }

    // Realizator
    void display() const
    {
      // Funkcja wyświetl. piksela z odpowiedniej bibl.
      // graficznej, np. BGI, WinBGI:
      putpixel( getX(), getY(), color );
    }

  private:
    int color;
};
```

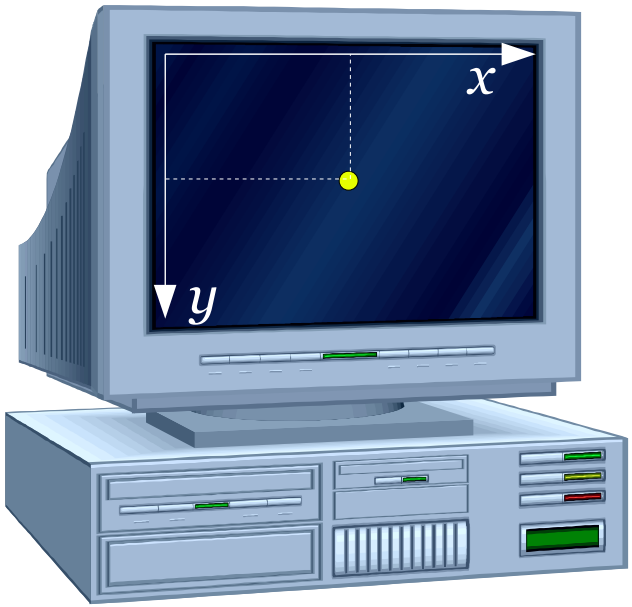
Aktywowanie konstruktora klasy bazowej.

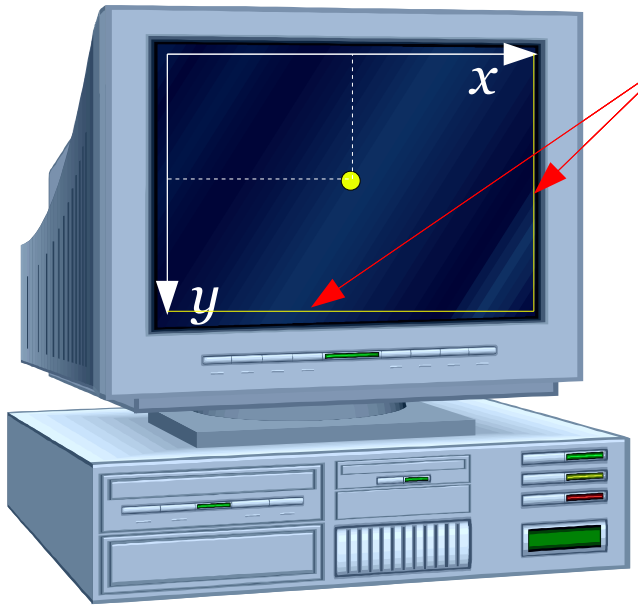
Pola *x* i *y* klasy *Point* są prywatne — również dla klasy pochodnej!, wymagany dostęp poprzez akcesory.

Klasa pochodna może zostać klasą bazową : `Point` → `Pixel` → `ScrPixel`

Piksel
Pixel

PikselEkranowy
ScrPixel





Ograniczenia

minX, *minY*, *maxX*, *maxY*.

Czy *PikselEkranowy* jest pewnego rodzaju *Pikselem*?

Pixel
-color
. . .

ScreenPixel
+minX, +minY +maxX, +maxY
. . .

Przykład klasy pochodnej ScreenPixel — dwie nowości

```
class ScreenPixel : public Pixel
{
  public :
    ScreenPixel( int startX = 0, int startY = 0, int startColor = 0 )
    : Pixel( startX, startY, startColor )
    {
    }

    void setX( int newX )
    {
      Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 );
    }

    void setY( int newY )
    {
      Point::setY( ( newY >= minY && newY <= maxY ) ? newY : 0 );
    }

    static int minX, minY, maxX, maxY;
};

int ScreenPixel::minX = 0;
int ScreenPixel::minY = 0;
int ScreenPixel::maxX = 799;
int ScreenPixel::maxY = 599;
```

Redefinicja funkcji odziedziczonych po klasie bazowej

2

1

Statyczne pola klasy — *static*

Nowość nr 1 — pola statyczne

Pola statyczne dla danej klasy występują *tylko raz*, są *współdzielone przez wszystkie obiekty* tej klasy. Istnieją nawet wtedy, gdy nie został utworzony *żaden* obiekt klasy.

Pola statyczne muszą być zdefiniowane poza klasą, nadaje się im wtedy wartość początkową.

```
class ScreenPixel : public Pixel
{
  public :
    . . .

    static int minX, minY, maxX, maxY;
};

int ScreenPixel::minX = 0;
int ScreenPixel::minY = 0;
int ScreenPixel::maxX = 799;
int ScreenPixel::maxY = 599;
```

Deklaracja pól statycznych

Definicja i inicjalizacja pól statycznych

Pola statyczne jako współdzielona pamięć obiektów

- Pola statyczne stanowią *wspólną, współdzieloną pamięć* wszystkich obiektów danej klasy.
- Wykorzystywane są do zapamiętywania informacji, które są wspólne dla wszystkich obiektów i *nie muszą być pamiętane osobno* w każdym z obiektów.

```
int main()
{
  ScreenPixel p1( 1, 1, 0 ), p2( 2, 2, 0 ), p3, p4;

  cout << endl << p1.minX ;
  cout << endl << p2.minY ;
  cout << endl << p3.maxX ;
  cout << endl << p4.maxY ;
  . . .
}
```

p2
x : 2
y : 2
color : 0

p4
x : 0
y : 0
color : 0

ScreenPixel
minX : 0
minY : 0
maxX : 799
maxY : 599

p3
x : 0
y : 0
color : 0

p1
x : 1
y : 1
color : 0

Statyczne pola publiczne — dostępne nawet gdy nie istnieje obiekt

- Pola statyczne są związane z klasą i występują nawet wtedy, gdy nie zdefiniowano żadnego obiektu klasy.
- Do pól statycznych klasy można zatem odwoływać się bez obiektu:

```
cout << endl << "Min. x = " << ScreenPixel::minX;
cout << endl << "Min. y = " << ScreenPixel::minY;
cout << endl << "Max. x = " << ScreenPixel::maxX;
cout << endl << "Max. y = " << ScreenPixel::maxY;
```

Zaraza, zaraz..., stosowanie pól publicznych jest niezgodne z zasadą *hermetyzacji* — a to jedna z podstawowych zasad programowania obiektowego...

- Pola statyczne mogą być prywatne, należy jednak wtedy zdefiniować odpowiednie funkcje dostępowe.
- Ponieważ pola statyczne są specyficzne, wymagają tez specyficznych funkcji dostępowych — takie funkcje powinny być *również statyczne*.

Pola statyczne w wersji *private*

```
class ScreenPixel : public Pixel
{
  public :
    . . .

  private :

    static int minX, minY, maxX, maxY;
};

int ScreenPixel::minX = 0;
int ScreenPixel::minY = 0;
int ScreenPixel::maxX = 799;
int ScreenPixel::maxY = 599;
```

Deklaracja statycznych pół prywatnych

Definicja i inicjalizacja pól statycznych, obowiązkowa również dla *pól prywatnych*

- Dostęp do statycznych pól prywatnych realizuje się zwyczajowo poprzez *statyczne funkcje składowe*, które są zwyczajowo publiczne
- Takie funkcje *nie mogą* odwoływać się do *niestatycznych* elementów klasy (dotyczy to niestatycznych *pól* i niestatycznych *funkcji składowych*).

Pola i funkcje statyczne

```
class ScreenPixel : public Pixel
{
  public :
    . . .
    static int getMinX()
    {
        return minX;
    }

  private :

    static int minX, minY, maxX, maxY;
};
ScreenPoint p;
. . .
cout << endl << "Min. x = " << p.getMinX();
cout << endl << "Min. x = " << ScreenPixel::getMinX();
```

Statyczna funkcja składowa — dostęp jedynie do *pól i innych funkcji statycznych*.

Nie może odwoływać się do ***niestatycznych*** elementów klasy, ***nie może*** odwoływać się do wskaźnika ***this***.

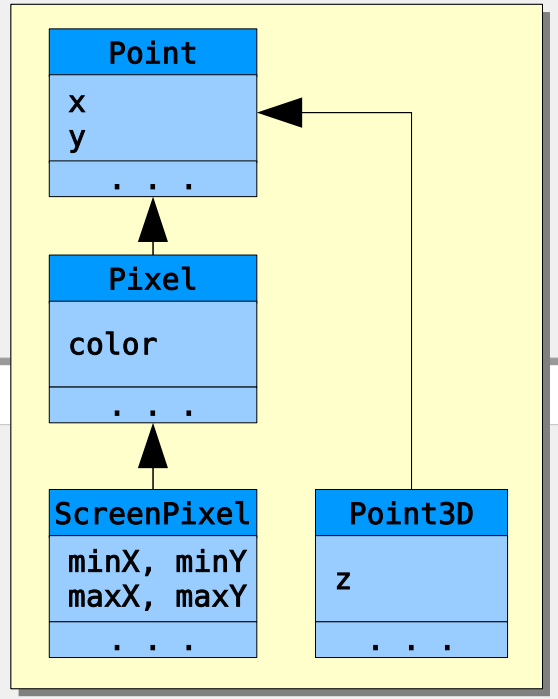
Funkcja statyczna ***nie może*** być ***virtual***, ***const***, ***volatile***.

Statyczne funkcje składowe można wywoływać używając dowolnego obiektu danej lub *bez niego*.

Uwaga — każda niestatyczna funkcja składowa *musi* być wywołana z *użyciem obiektu*, wskaźnik do tego obiektu — nazwany *this* — dostępny jest w każdej niestatycznej funkcji składowej. Ten wskaźnik *nie jest dostępny* w funkcjach statycznych.

Nowość nr 2 — redefinicja metody w klasie pochodnej

```
class Point
{
  public:
    . . .
    void setX( int newX ) { x = newX; }
    . . .
};
```




```
class ScreenPixel : public Pixel
{
  public :
    . . .
    void setX( int newX )
    {
      Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 );
    }
    . . .
};
```

- W klasie pochodnej można *zmienić* sposób realizacji metody odziedziczonej poprzez jej *redefinicję*, czyli jej ponowne *zdefiniowanie*.

Redefinicja — ponowne zdefiniowanie takiej samej funkcji

```
class Point
{
  public:
    . . .
    void setX( int newX ) { x = newX; }
    . . .
};
```

Takie same funkcje — Te same nagłówki

```
class ScreenPixel : public Pixel
{
  public :
    . . .
    void setX( int newX )
    {
      Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 );
    }
    . . .
};
```

- Redefinicja funkcji składowej nie powinna zmieniać jej nagłówka — *rezultat* oraz *liczba* i *typy parametrów* nie powinny się zmienić. Nie jest to zabronione, lecz takie zmiany zmieniają interfejs klasy, a zwykle nie o to nam chodzi.

Redefinicja — taka sama funkcja, inna realizacja

```
class Point
{
  public:
    . . .
    void setX( int newX ) { x = newX; }
    . . .
};
```

Nowa funkcja działa inaczej

```
class ScreenPixel : public Pixel
{
  public :
    . . .
    void setX( int newX )
    {
      Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 );
    }
    . . .
};
```

- Redefinicja funkcji składowej polega na „przykryciu” jej działania nową wersją. Dzięki temu obiekt klasy pochodnej *inaczej reaguje* na ten sam *komunikat*.
- Zmienia się *zachowanie* obiektów.

Obiekt klasy ScreenPixel wywołuje własną wersję funkcji setX

```
class Point
{
  public:
    . . .
    void setX( int newX ) { x = newX; }
};
```

Funkcja *setX* klas *Point* i *Pixel*

```
class ScreenPixel : public Pixel
{
  public :
    . . .
    void setX( int newX )
    {
      Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 );
    }
};
```

```
Point       pt;
Pixel       px;
ScreenPixel sp;

pt.setX( -10 );
px.setX( -10 );
sp.setX( -10 );
```

Funkcja *setX* klasy *ScreenPixel*

Realizacja funkcji setX klasy ScreenPixel

```
class Point
{
  public:
    . . .
    void setX( int newX ) { x = newX; }
};
```

```
class ScreenPixel : public Pixel
{
  public :
    . . .
    void setX( int newX )
    {
      Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 ) ;
    }
};
```

Wywołanie funkcji *setX* klasy bazowej *Point*.

Użycie prefiksu *Point*:: jest konieczne!

Rezultatem tego wyrażenia jest wartość parametru *newX* gdy jest on prawidłowy, w przeciwnym przypadku rezultatem jest wartość 0.

- Do z klasy bazowej można dalej odwoływać się stosując prefiks w postaci **`nazwa_klasy_bazowej::`**.

Różne warianty realizacji funkcji setX klasy ScreenPixel

```
void ScreenPixel::setX( int newX )
{
  Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 );
}
```

Zerowanie x gdy wartość newX jest nieprawidłowa

```
void ScreenPixel::setX( int newX )
{
  Point::setX( ( newX >= minX && newX <= maxX ) ? newX : getX() );
}
```

Gdy wartość newX jest nieprawidłowa, x bez zmian

```
void ScreenPixel::setX( int newX )
{
  if( newX >= minX && newX <= maxX )
    Point::setX( newX );
  else
    Point::setX( 0 );
}
```

Zerowanie x gdy wartość newX jest nieprawidłowa

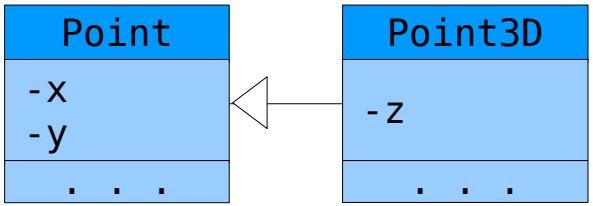
```
void ScreenPixel::setX( int newX )
{
  if( newX >= minX && newX <= maxX )
    Point::setX( newX );
}
```

Gdy wartość newX jest nieprawidłowa, x bez zmian

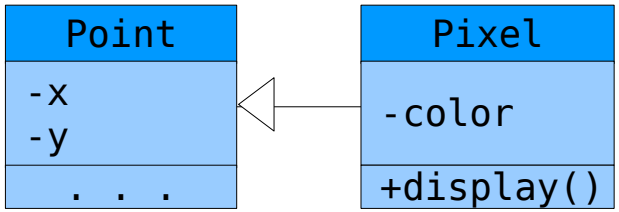
Dziedziczenie
Krótkie podsumowanie

Co daje dziedziczenie?

- Dziedziczenie pozwala na (zwykle) szybkie i łatwe zbudowanie *nowej klasy pochodnej* na podstawie istniejącej już *klasy bazowej*.
- Klasa pochodna zazwyczaj posiada *więcej informacji niż klasa bazowa* — informacje te są przechowywane w *nowych polach klasy pochodnej*.



- Klasa pochodna zazwyczaj posiada *potrafi więcej niż klasa bazowa* — umiejętności te są wynikiem zdefiniowania *nowych funkcji składowych klasy pochodnej*.



- Najczęściej klasa pochodna *wie więcej* (*nowe pola*) i *potrafi więcej* (*nowe funkcje*).

Konstruktory a dziedziczenie

- Każda klasa powinna być wyposażona w konstruktor(y). Zazwyczaj jest to konstruktor *domyślny, ogólny* i *kopiujący* (gdy trzeba).
- Konstruowanie obiektu *klasy pochodnej* wymaga *zainicjowania pól odziedziczonych* oraz *nowych pól* tej klasy. Zwykle do zainicjowania pól odziedziczonych używa się *odpowiedniego konstruktora klasy bazowej*.
- *Aktywowanie* konstruktora *klasy bazowej* polega na umieszczeniu jego nazwy (z parametrami gdy to konieczne) na liście inicjalizacyjnej *klasy pochodnej*.
- Konstruktor *klasy bazowej* uruchomi się zatem ***przed*** ciałem konstruktora *klasy pochodnej*.
- Jeżeli nasza klasa bazowa jest pochodną jakiejś klasy *X*, to zwykle konstruktor uaktywni najpierw konstruktor tejże klasy *X*.
- W efekcie *najpierw* uaktywni się konstruktor tej klasy bazowej, która nie ma już żadnego przodka, a *potem* dopiero aktywowane zostaną konstruktory klas pochodnych.

Kolejność aktywowania konstruktorów

Ten konstruktor wykona się jako pierwszy.
Po jego wykonaniu dokończone zostaną konstruktory klas pochodnych.

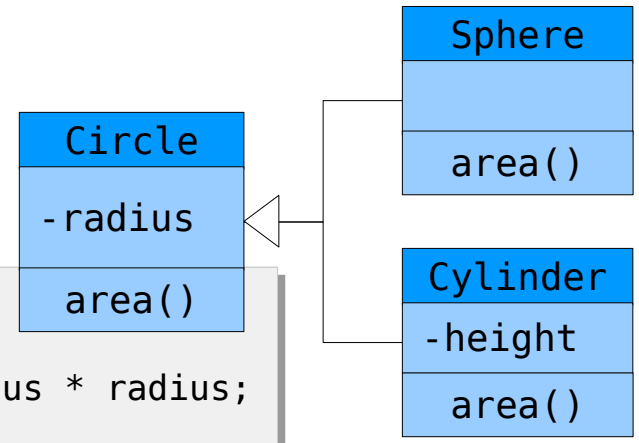
```
class Point
{
  public:

    Point( int startX, int startY )
    : x( startX ), y( startY )
    {
    }
    . . .
};
```

```
class Pixel : public Point
{
  public :

    Pixel( int startX, int startY, int startColor )
    : Point( startX, startY ), color( startColor )
    {
    }
    . . .
};
```

```
class ScreenPixel : public Pixel
{
  public :

    ScreenPixel( int startX, int startX, int startColor )
    : Pixel( startX, startY, startColor )
    {
    }
    . . .
};
. . .
ScreenPixel p( 10, 10, YELLOW );
```

Redefiniowanie funkcji składowych

- W klasie pochodnej można *zmienić* sposób realizacji metody odziedziczonej poprzez jej *redefinicję*, czyli jej ponowne *zdefiniowanie*.
- Redefinicja funkcji składowej polega na „przykryciu” jej działania nową wersją. Dzięki temu obiekt klasy pochodnej *inaczej reaguje* na ten sam *komunikat*.
- Redefinicja funkcji składowej w klasie pochodnej pozwala nam zmienić *nieodpowiednie zachowanie* obiektów klasy bazowej.
- Czasem stosuje się dziedziczenie *wyłącznie* po to, by zmienić *nieodpowiadające nam zachowanie* zdefiniowane w klasie bazowej.

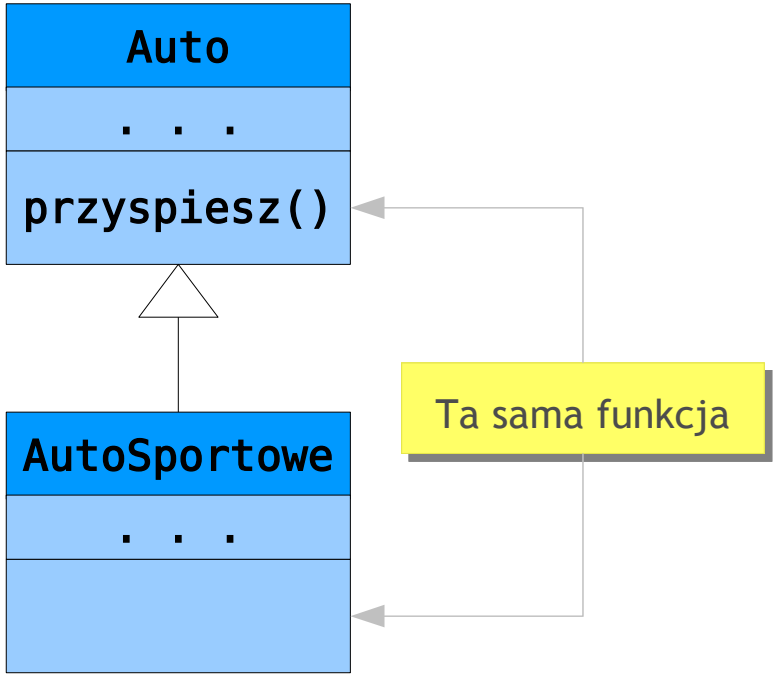


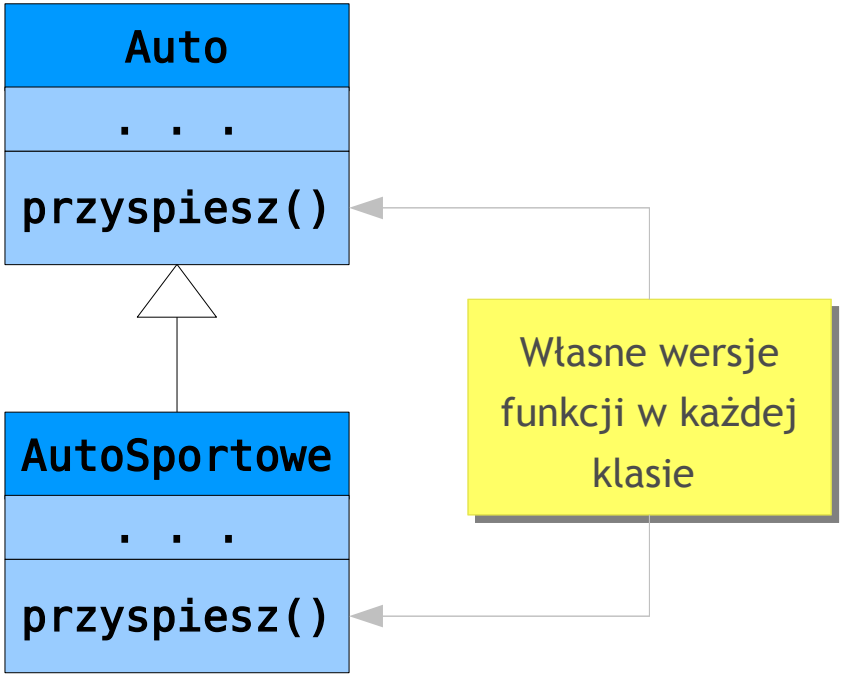

```
Sphere::area()
{
  return 4 * Circle::area();
}
```

```
Circle::area()
{
  return M_PI * radius * radius;
}
```

```
cylinder::area()
{
  return 2 * Circle::area() + height
         * 2 * M_PI * getRadius();
}
```

Redefiniowanie funkcji składowych a diagramy UML



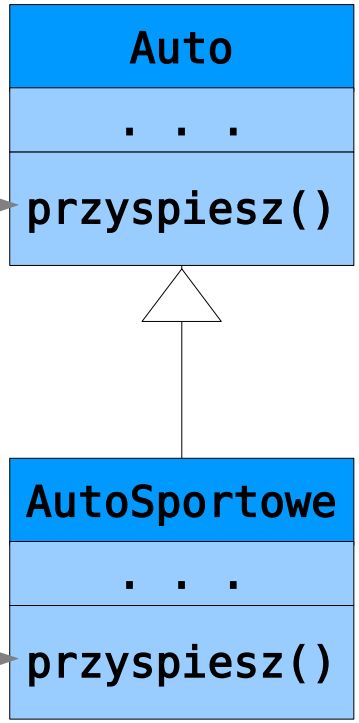


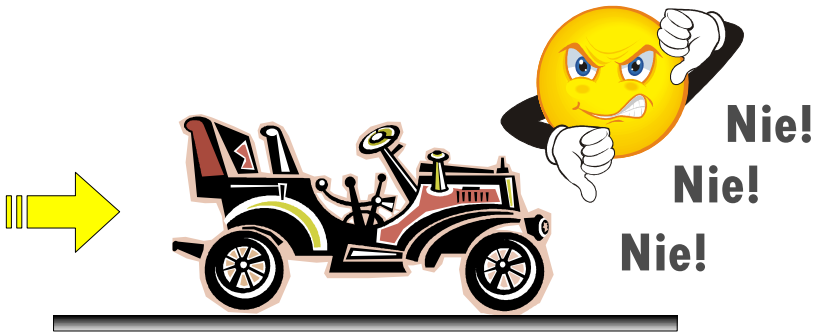
Klasa AutoSportowe wykorzystuje odziedziczoną funkcję *przyspiesz* klasy Auto.

Klasa *AutoSportowe* definiuje własną funkcję *przyspiesz*, nastąpiła jej ***redefinicja***.

Co daje redefiniowanie funkcji składowych?

```
Auto mojeAutko;
mojeAutko.przyspiesz();
```






```
AutoSportowe mojeAutko;
mojeAutko.przyspiesz();
```

Redefinicja funkcji składowej pozwala na uzyskanie *innego zachowania* obiektu klasy pochodnej.

Jeżeli nie odpowiada nam zachowanie obiektu klasy bazowej, stosujemy *dziedziczenie* i dokonujemy *redefinicji* funkcji, których działanie nam nie odpowiada.

Dzięki temu, nie trzeba *modyfikować klasy bazowej*, należy utworzyć *nową klasę pochodną*, spełniającą nasze oczekiwania — osiągamy to stosując redefinicję nieodpowiadających nam funkcji składowych.

Odwołania do prywatnych pól klasy bazowej

- Odwołania do prywatnych pól klasy bazowej są niewygodne — pola nie są bezpośrednio dostępne, trzeba wykorzystywać funkcje dostępowe, np:

```
class Pixel : public Point
{
  . . .
  void display() const
  {
      putpixel( getX(), getY(), color );
  }
};
```

```
class Point
{
  . . .
  private:
    int  x, y;
};
```

```
class ScreenPixel : public Pixel
{
  . . .
  void setX( int newX )
  {
    Point::setX( ( newX >= minX && newX <= maxX ) ? newX : getX() );
  }
};
```

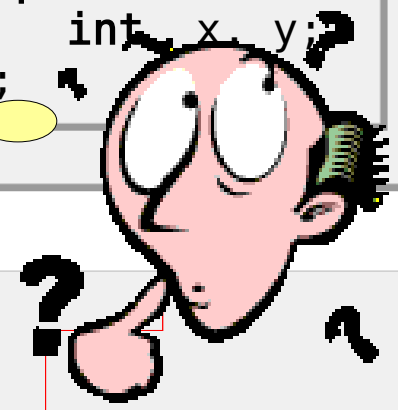
Dlaczego nie można odwoływać się bezpośrednio?

- Odwołania do prywatnych pól klasy bazowej są niewygodne — pola nie są bezpośrednio dostępne, trzeba wykorzystywać funkcje dostępowe, np:

```
class Pixel : public Point
{
  . . .
  void d
  {

  }
};
```

```
class Point
{
  . . .
  private:
    int x, y;
};
```

Dlaczego nie można odwoływać się bezpośrednio? Przecież te klasy tworzą rodzinę?



```
class
{
  . . .
  void setX( int newX )
  {
    Point::setX( ( newX >= minX && newX <= maxX ) ? newX : getX() );
  }
};
```

Protected — składowe chronione

- Przewidując, że pewna klasa będzie wykorzystywana jako klasa bazowa, można zadeklarować jej składowe (pola i metody) jako ***chronione — protected***.

```
class Point
{
  public:

    . . .

  protected:
    int  x, y;
};
```

- Składowe zadeklarowane jako *protected* są *dostępne* dla obiektów *wszystkich klas pochodnych* (tak jak składowe public).
- Składowe zadeklarowane jako *protected* są *niedostępne* dla obiektów *innych, niezaprzyjaźnionych klas* (tak jak składowe *private*).
- Zatem specyfikator *protected* działa jak *private* dla „obcych” a jak *public* dla „swoich”, przy czym „swoi” to klasy pochodne.

Protected – składowe chronione

```
class Point
{
  public:
    . . .
  protected:
    int  x, y;
};
```

```
class Pixel : public Point
{
  . . .
  void display() const
  {
      putpixel( x, y, color );
  }
};
```

```
class ScreenPixel : public Pixel
{
  . . .
  void setX( int newX )
  {
    Point::setX( ( newX >= minX && newX <= maxX ) ? newX : x );
  }
};
```

Dziedziczenie wielobazowe

Dziedziczenie wielobazowe — koncepcja

- Dziedziczenie *wielobazowe* pozwala na tworzenie klas potomnych, dziedziczących z więcej niż jednej klasy bazowej.
- Klasa pochodna dziedziczy pola i funkcje składowe każdej z klas bazowych.

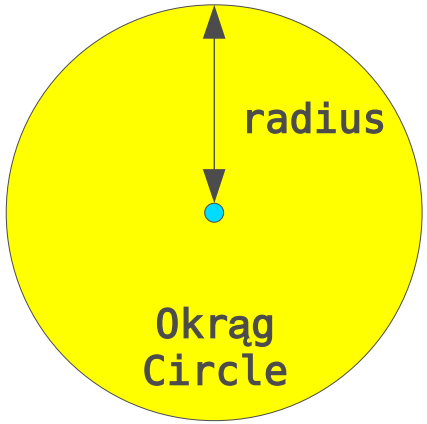
Za i przeciw dziedziczeniu wielobazowemu:

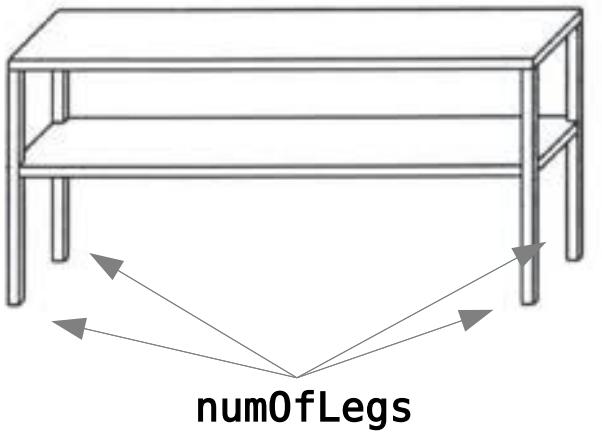
- *Tak* — pozwala programiście budować elastyczne i dobrze odwzorowujące rzeczywistość hierarchię klas.
- *Nie* — zmusza programistę do większej uwagi, zwiększa złożoność języka oraz kompilatora.
- W wielu językach programowania zrezygnowano z dziedziczenia wielobazowego:
  - jest np. w językach *C++*, *Pyton*, *Eiffel*, *Pearl*;
  - nie ma np. w językach: *Ada*, *Ruby*, *Objective-C*, *PHP*, *C#*, *Object Pascal*, *Java*.

Dziedziczenie wielobazowe — przykład

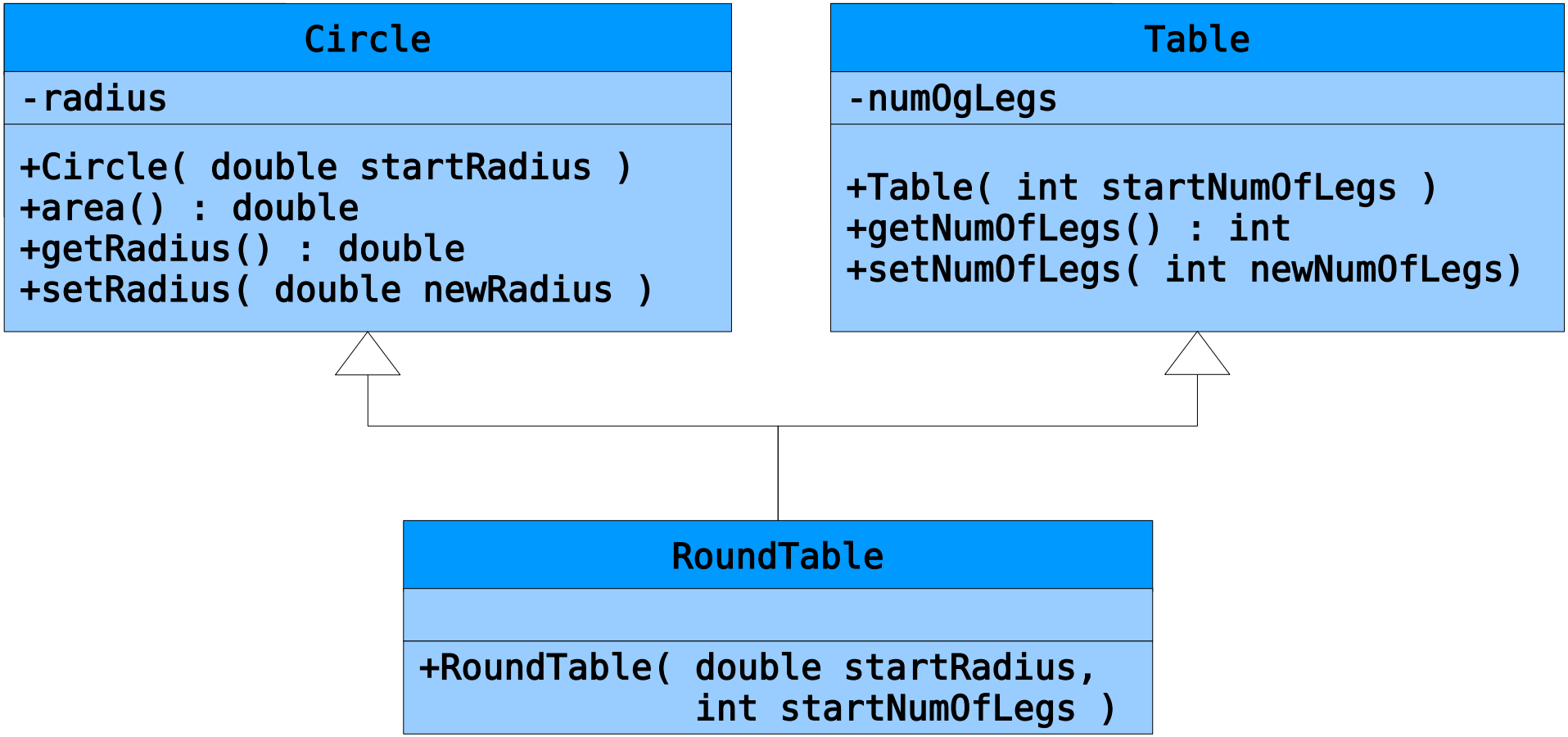
Circle
-radius
+Circle(double startRadius) +area() : double +getRadius() : double +setRadius(double newRadius)

Table
-numOgLegs
+Table(int startNumOfLegs) +getNumOfLegs() : int +setNumOfLegs(int newNumOfLegs)





Dziedziczenie wielobazowe — przykład



Dziedziczenie wielobazowe – przykład

```
class Circle
{
  public:
    Circle( double startRadius = 0 ) : radius( startRadius ) {}
    double getRadius() const { return radius; }
    void setRadius( double newRadius ) { radius = newRadius; }
    double area() const { return M_PI * radius * radius; }
    double perimeter() const { return 2 * M_PI * radius; }

  private:
    double radius;
};
```

```
class Table
{
  public:
    Table( int startNumOfLegs = 3 ) : numOfLegs( startNumOfLegs ) {}
    int getNumOfLegs() const { return numOfLegs; }
    void setNumOfLegs( int newNumOfLegs ) { numOfLegs = newNumOfLegs; }

  private:
    int numOfLegs;
};
```

Dziedziczenie wielobazowe – przykład

```
class RoundTable : public Circle, public Table
{
  public:
    RoundTable( double startRadius = 0, int startNumOfLegs = 3 )
    : Circle( startRadius ), Table( startNumOfLegs )
    {
    }
};
```

```
int main()
{
  RoundTable kingArturTable( 10, 12 );

  cout << "\nPromien stolu: " << kingArturTable.getRadius();
  cout << "\nLb. nog stolu: " << kingArturTable.getNumOfLegs();
  cout << "\nPole    stolu: " << kingArturTable.area();
  cout << "\nObwod   stolu: " << kingArturTable.perimeter();

  return EXIT_SUCCESS;
}
```

```
Promien stolu: 10
Lb. nog stolu: 12
Pole    stolu: 314.159
Obwod   stolu: 62.8319
```

Dziedziczenie wielobazowe — kolejność aktywowania konstruktorów

- *Kolejność aktywowania konstruktorów* dla obiektu klasy pochodnej wynika z kolejności występowania nazw klas bazowych w *deklaracji* tej klasy.
- *Nie jest istotna* kolejność ich umieszczenia na *liście inicjalizacyjnej* konstruktora klasy pochodnej.
- Specjalnie traktowane są wirtualne klasy bazowe — ale o tym później.
- Kolejność aktywowania destruktorów jest *odwrotna* do kolejności stosowanej w przypadku konstruktorów.

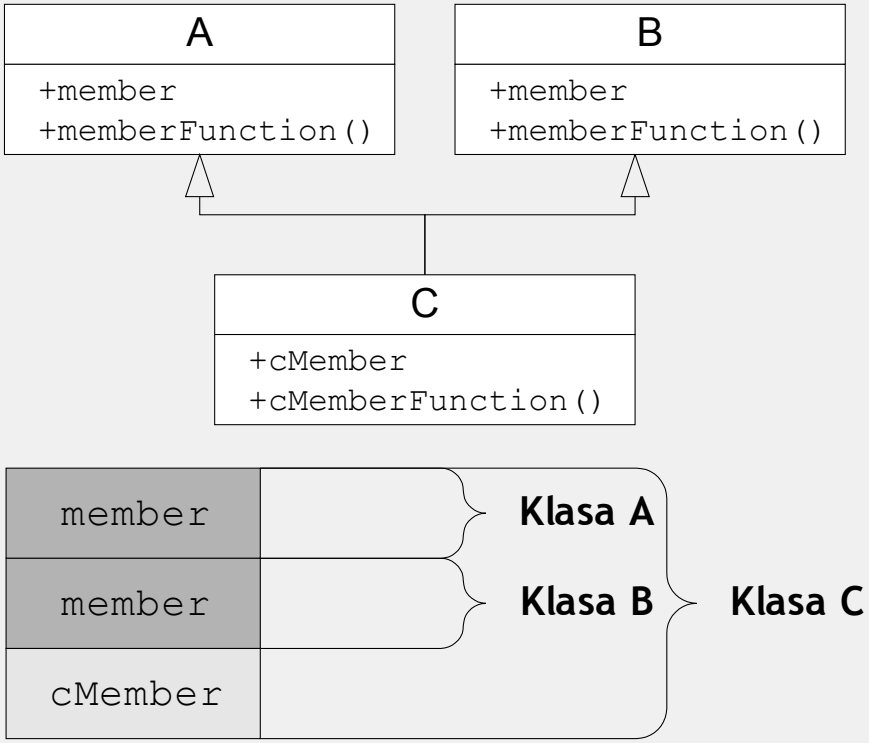
Kolejność aktywowania konstruktorów – przykład

```
class A
{
  public:
    A()  : aMember( 0 ) { cout << "Konstruktor: A()" << endl; }
    ~A() { cout << "Destruktor: ~A()" << endl; }
    int aMember;
};

class B
{
  public:
    B()  : bMember( 0 ) { cout << "Konstruktor: B()" << endl; }
    ~B() { cout << "Destruktor: ~B()" << endl; }
    int bMember;
};

class C : public A, public B
{
  public:
    C()  : A(), B(), cMember( 0 ) { cout << "Konstruktor: C()" << endl; }
    ~C() { cout << "Destruktor: ~C()" << endl; }
    int cMember;
};
. . .
C obj;
```

```
Konstruktor: A()
Konstruktor: B()
Konstruktor: C()
Destruktor: ~C()
Destruktor: ~B()
Destruktor: ~A()
```

Składowe o tej samej nazwie

```
class A
{
  public:
    int  member;
    void memberFunction() {};
};

class B
{
  public:
    int  member;
    void memberFunction() {};
};

class C : public A, public B
{
  public:
    int  cMember;
    void cMemberFunction();
};

. . .
C obj;
```



obj : tylko pola obiektu

Składowe o tej samej nazwie – problem

- Odwołania jednoznaczne:

```
obj.cMember = 10;
obj.cMemberFunction();
```

- Odwołanie niejednoznaczne:

```
obj.member = 100;
obj.memberFunction();
```

- Błąd kompilacji (np. GCC):

```
Request for member 'member' is ambiguous, candidates are:
int B::member is int A::member
```

- Błąd kompilacji (np. Borlanc C++):

```
Member is ambiguous: 'A::member' and 'B::member'
Member is ambiguous: 'A::memberFunction' and 'B::memberFunction'
```

Składowe o tej samej nazwie, wykorzystanie operatora zasięgu ::

- Odwołania jednoznaczne:

```
obj.A::member = 100;
obj.B::member = 200;

cout << obj.A::member << endl;
obj.A::memberFunction();

cout << obj.B::member << endl;
obj.B::memberFunction();
```

```
C * ptr;

ptr = new C;

ptr->A::member = 100;
ptr->B::member = 200;

cout << ptr->A::member << endl;
ptr->A::memberFunction();

cout << ptr->B::member << endl;
ptr->B::memberFunction();

delete ptr;
```

Wykorzystanie duplikatów składowych klas

```
class Vehicle
{
  public:
    Vehicle( int maxSpeed = 0, int numOfWheels = 0 )
    : maxSpeed( maxSpeed ), numOfWheels( numOfWheels )
    {
    }

  protected:
    int maxSpeed;
    int numOfWheels;
};

class Boat
{
  public:
    Boat( int maxSpeed = 0, int displacement = 0 )
    : maxSpeed( maxSpeed ), displacement( displacement )
    {
    }

  protected:
    int maxSpeed;
    int displacement;
};
```

Wykorzystanie duplikatów składowych klas

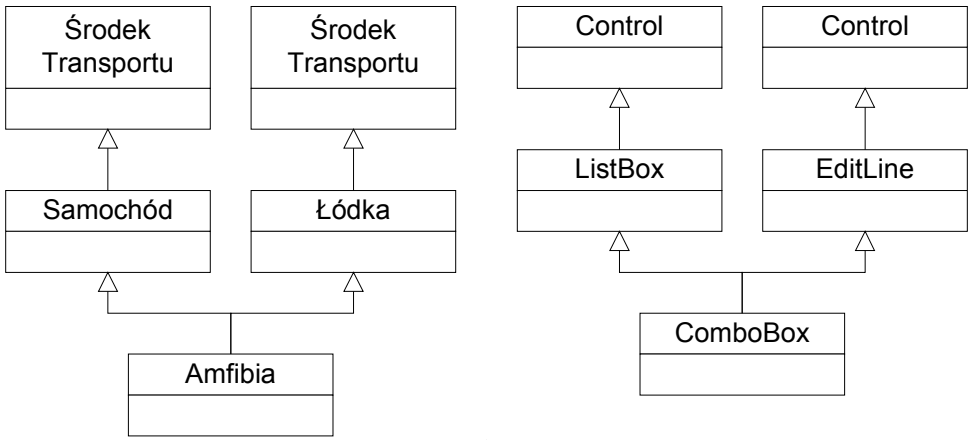
```
class Amphibian : public Vehicle, public Boat
{
  public:
    Amphibian( int maxLandSpeed = 0, int maxWaterSpeed = 0,
               int numOfWheels = 0, int displacement = 0 )
    : Vehicle( maxLandSpeed, numOfWheels ),
      Boat( maxWaterSpeed, displacement )
    {
    }

    int getMaxLandSpeed() const  { return Vehicle::maxSpeed; }
    int getMaxWaterSpeed() const { return Boat::maxSpeed; }
    int getNumOfWheels() const   { return numOfWheels; }
    int getDisplacement() const  { return displacement; }
};

. . .

Amphibian a( 120, 20, 4, 1500 );

cout << endl << "Maks. predk. na ladzie : " << a.getMaxLandSpeed();
cout << endl << "Maks. predk. na wodzie : " << a.getMaxWaterSpeed();
cout << endl << "Liczba kol : " << a.getNumOfWheels();
cout << endl << "Wypornosc  : " << a.getDisplacement();
```

Dziedziczenie bazowe — sporo problemów, różne ich rozwiązania

Problemów z dziedziczeniem wielobazowym jest *wiele*.

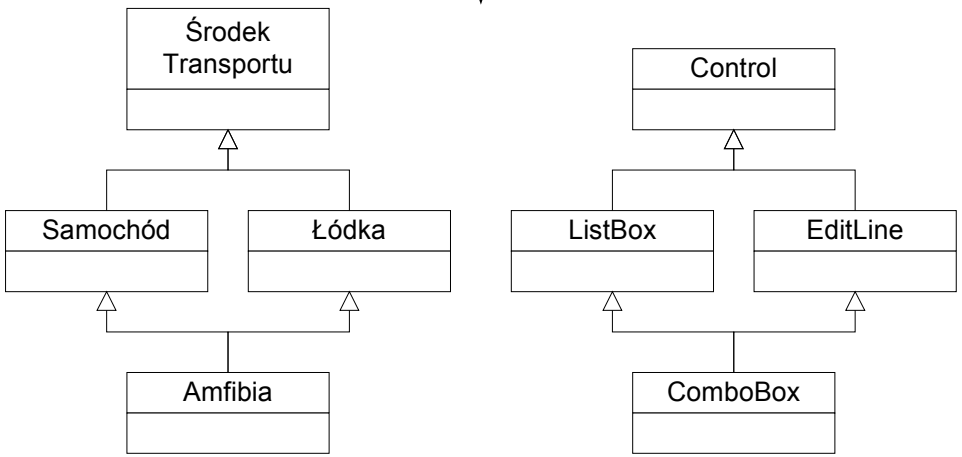
Omówienie tych zagadnień *wykracza poza ramy tego wykładu*.

W wielu językach obiektowych zrezygnowano z tego rodzaju dziedziczenia, wykorzystując tylko dziedziczenie *jednobazowe*.



1 Wspólna, niebezpośrednia klasa bazowa a problem duplikatów składowych

2 Rozwiązanie problemu 1 powoduje nowy problem — *diamond problem*



Ćwiczenie 1

Zdefiniować klasę *Data* zawierającą pola prywatne: *int dzien*, *int miesiac*, *int rok*. Klasa powinna zawierać funkcje dostępowe, pobierające zawartość poszczególnych pól (np. *podajDzien* itd.), oraz funkcje pozwalające na ustalanie zawartości pól (np. *ustawDzien*, itd.).

Funkcje ustawiania wartości mogą w tej wersji nie kontrolować poprawności parametrów wstawianych do pól.

Klasa ma posiadać konstruktor domyślny, zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących dzień, miesiąc, rok. Proszę zdefiniować klasę pochodną klasy *Data* — *DataZKontrolą*. W klasie tej należy dokonać redefinicji funkcji ustawiających wartości pól (*ustawDzien*, itd.), tak aby dokonywały kontroli wartości parametrów, adekwatnie dla charakteru pola.

Klasa *DataZKontrolą* powinna posiadać konstruktor domyślny zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących godzinę, minutę, sekundę. Proszę pamiętać o aktywowaniu konstruktorów klasy bazowej.

Ćwiczenie 2

Zdefiniować klasę *Czas* zawierającą pola prywatne: *int godzina*, *int minuta*, *int sekunda*. Klasa powinna zawierać funkcje dostępowe, pobierające zawartość poszczególnych pól (np. *podajGodzine* itd.), oraz funkcje pozwalające na ustalanie zawartości pól (np. *ustawGodzine*, itd.). Klasa ma posiadać konstruktor domyślny zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących godzinę, minutę, sekundę.

Funkcje ustawiania wartości mogą w tej wersji nie kontrolować poprawności parametrów wstawianych do pól.

Proszę zdefiniować klasę pochodną klasy *Czas* — *CzasZKontrolą*. W klasie tej należy dokonać redefinicji funkcji ustawiających wartości pól (*ustawGodzine*, itd.), tak aby dokonywały kontroli wartości parametrów, adekwatnie dla charakteru pola.

Klasa *CzasZKontrolą* powinna posiadać konstruktor domyślny zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących godzinę, minutę, sekundę. Proszę pamiętać o aktywowaniu konstruktorów klasy bazowej.

Ćwiczenie 3

Zdefiniować klasę *Osoba* zawierającą informację o numerze PESEL, imieniu i nazwisku osoby. Mają to być odpowiednio prywatne pola *int PESEL*, *char imie*[*N*], *char nazwisko*[*N*], gdzie *N* oznacza pewną stałą adekwatną dla maksymalnej długości imienia i nazwiska — np. *N*=25. Do ustawiania oraz pobierania ich wartości należy zdefiniować odpowiednie funkcje składowe. Należy zdefiniować również odpowiedni konstruktor (konstruktory). Klasa powinna również posiadać funkcje składową *char kimJestes*(), której rezultatem powinien być napis „*Osoba*”.

Wykorzystując dziedziczenie i klasę *Osoba*, należy zbudować klasę pochodną *Student*, rozbudowując ją o prywatne pole *int numerAlbumu*, definiując jednocześnie odpowiednie funkcje dostępowe oraz konstruktor (konstruktory).

Analogicznie proszę zbudować klasę pochodną klasy *Student* o nazwie *Absolwent*, rozbudowując ją o prywatne pole *int numerDyplomu*, definiując jednocześnie odpowiednie funkcje dostępowe oraz konstruktor (konstruktory).

Klasy *Student* i *Absolwent* powinna redefiniować funkcje składową *char kimJestes*(), której rezultatem powinien być odpowiednio napis „S*tudent*” i „*Absolwent*”.

Dodatek specjalny

Z ostatniej chwili — najnowsze doniesienia na temat Runnerów

Tylko u nas — ujawniamy sensacyjne informacje na temat tajemnic pochodzenia rodziny Runnerów!

Środa
14 grudnia 2011

Redaktor Prowadzący
Roman Simiński

Fakty IT

gazeta codzienna

Niecodzienny dodatek *Gazety Codziennej*

Dziś poznamy sensacyjne informacje o pochodzeniu znanego wszystkim osobnika *R*, przedstawiciela rodziny *Runnerów*.

Rodzina Runnerów ujawnia swoje tajemnice

Ekskluzywny wywiad z R!

R — witamy Cię ponownie

— Witam, dziękuję za zaproszenie.

Pojawiły się sensacyjne pogłoski na temat Twojego pochodzenia, czy możesz coś na ten temat powiedzieć?

— Zaraz tam sensacje. Prosta sprawa. Ja również mam swojego przodka. Jak każdy. A mój przodek, też miał swojego. I tyle.

Kim jest Twój przodek?

— Musisz zrozumieć jedną rzecz — u nas, gwiazd programów jest trochę inaczej niż u Was w realu. Przypomnę Ci — jestem *R*, jestem reprezentantem klasy *Runner*. Obiektem jestem. Takich jak ja może być więcej. Wszystko co wiemy, i wszystko co umiemy, zostało zdefiniowane w klasie *Runner*.

gazeta codzienna

Co to oznacza?

— To oznacza, ze aby nas poznać, trzeba poznać klasę, do której należymy. Zatem trzeba się dokładnie przyjrzeć klasie *Runner*.

I co zobaczymy?

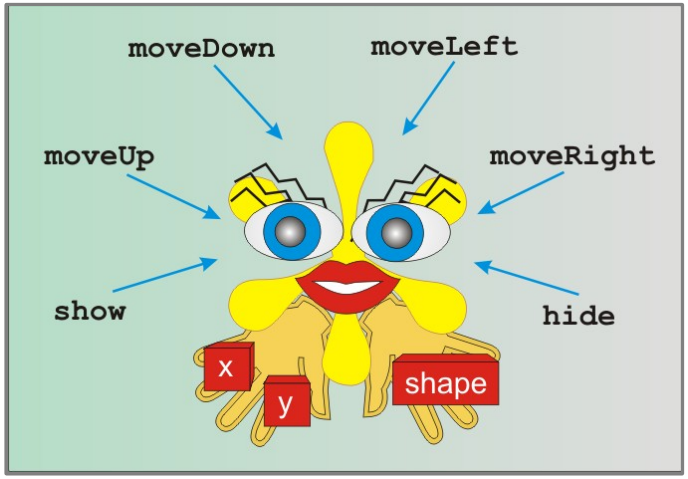
— To zależy jak się patrzy. Pamiętasz coś z naszego ostatniego wywiadu redaktorku?

No tak, pamiętam...

— Sprawdźmy: kim tak naprawdę jestem?

Jesteś ruchomym punktem ekranowym. Znasz swoją pozycję, znak jakim się wizualizujesz i potrafisz się przesuwać po ekranie i nie wylecisz poza.

— Nieprawdopodobne, redaktorek coś pamięta! Dobrze! To co powiedziałeś wystarczy do zrozumienia historii Runnerów. Zaczynamy opowieść?



Zamieniam się w słuch...

— Tak jak sam powiedziałeś jestem punktem. I w naszym świecie wszystko się od tego zaczęło. Na początku był punkt, a klasa jego zwała się *Point*. Obiekty jej klasy były takie głupiutkie..., wiedziały tylko gdzie są! Rozumiesz, miały pola *x* i *y*, w których pamiętały współrzędne na płaszczyźnie 2D. I tyle.

gazeta codzienna

A co te obiekty umiały?

— Właściwie nic. Ponieważ informacje o położeniu były prywatne, umiały tylko przyjąć polecenia ustalenia nowej pozycji — *setX*(*noweX*) i *setY*(*noweY*), oraz polecenia podania tej pozycji — *getX*() i *getY*().

To rzeczywiście nie wiele...

— No mówiłem, że obiekty klasy *Point* to głupiutkie były. I wiesz — one w ogóle nie wyglądały! Nie było wiadomo czy są fajne, czy nie, bo nie mogłeś ich zobaczyć.

Kiepsko, i co dalej?

— Programista stwierdził, że potrzebuje obiektów, które wyglądają i potrafią się pokazać. I jeszcze, żeby wiedziały, że są na ekranie i pozwoliły się ustawić poza nim. Tym głupiutkim obiektom klasy *Point* mogłeś ustawić każdą pozycję a one grzecznie to przyjmowały.

Rozumiem, czyli programista przerobił klasę *Point*...

— Nie, nie nie! Zostawił ją w spokoju. Taka klasa zawsze się może przydać, zawsze warto mieć trochę głupiutkich obiektów. Programista stworzył nową klasę o nazwie *ScreenPoint* — punkt ekranowy, ale zrobił to sprytnie. Uczynił klasę *Point* ojcem klasy *ScreenPoint* — zastosował *dziedziczenie*. Klasa *ScreenPoint* przejęła cały materiał genetyczny z klasy *Point*.

A po co to dziedziczenie, nie mógł napisać jej od nowa?

— A po co pisać od nowa, jak już jest? Lepiej wykorzystać to, co jest, i rozszerzyć o to, czego brakuje.

gazeta codzienna

Niby tak...

— No właśnie. W klasie *ScreenPoint* pojawiło się nowe pole *shape* do przechowywania znaku, jakim ma wyglądać punkt ekranowy. Pojawiły się funkcje do ustawiania i pobierania tego pola, bo ono jest prywatne. Ale najważniejsze jest to, że pojawiły się funkcje: *show*() do pokazywania punktu i *hide*() do jego ukrywania.

Czyli obiekty tej klasy zaczęły wyglądać...

— Zaczęły wyglądać i uzyskały możliwość pokazywania się na ekranie i znikania z niego. No i jeszcze jedno. W tej klasie pojawiły się jeszcze dwie funkcje, takie pomocnicze, ale ważne: *isXOnScreen*(*jakiśX*) i *isYOnScreen*(*jakiśY*). Dzięki nim w klasie *ScreenPoint* można było wzmocnić kontrolę współrzędnych podawanych obiektom.

To obiekty klasy *ScreenPoint* się mądrzejsze zrobiły...

— Tak, całkiem mądre. Wiedzą gdzie są — bo wiedzą i potrafią tyle co obiekty klasy *Point*. Wyglądają i potrafią się pokazać i zniknąć. Potrafią również sprawdzić, jakie współrzędne są poprawne dla ekranu.

Gdyby tę klasę przerobić, to obiekty umiały by tyle co Ty...

— Nie, nie nie! Nie przerabiamy! Budujemy nową kasę o rozszerzonych możliwościach! Zrozum wreszcie! Jeżeli to zrozumiesz, zrozumiesz jak powstałem. Obiekty klasy *ScreenPoint* potrafią wyglądać, pokazywać się i znikać. Ale nie potrafią się poruszać. Ale wystarczy jeszcze raz zastosować dziedziczenie...

gazeta codzienna

I z klasy *ScreenPoint* zbudować klasę *Runner*...

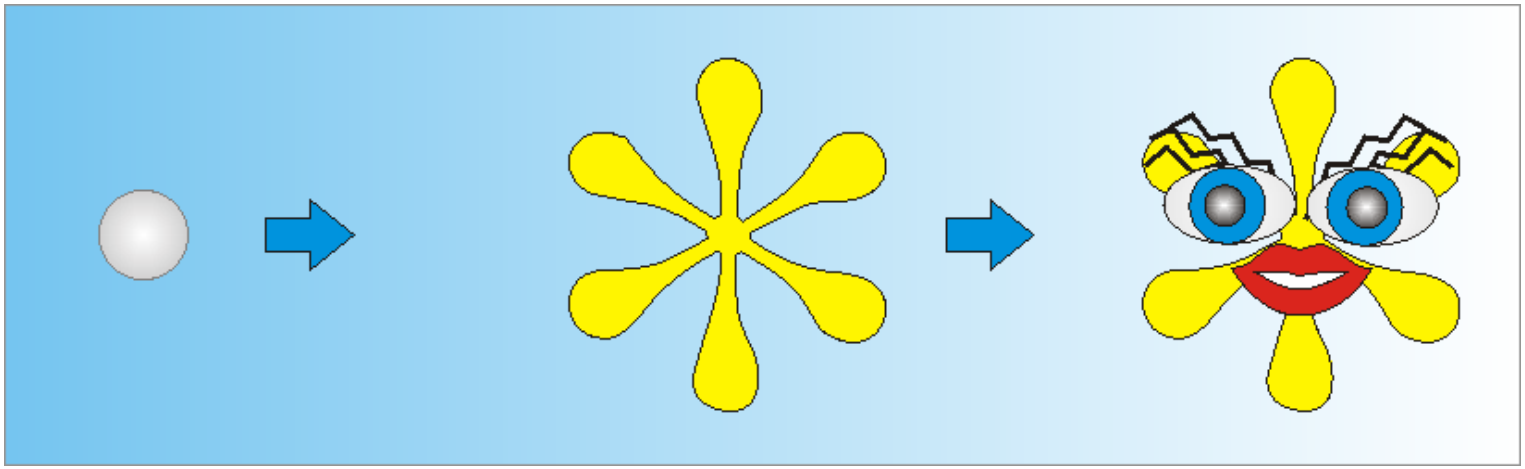
— Yes! Yes! Yes! Nareszcie zrozumiałeś. Ojcem klasy *Runner* jest właśnie *ScreenPoint*. Umiem wszystko to, co każdy obiekt tej klasy. Ale umiem więcej, bo w mojej klasę zdefiniowano nowe funkcje: *moveUp*(), *moveDown*(), *moveLeft*(), *moveRigt*().

Zatem pochodzisz od punktu, który jest Twoim dziadkiem, ale Twoim ojcem jest punkt ekranowy?

— Dokładnie tak. Takie jest moje pochodzenie i się tego nie wstydzę.

Ale dlaczego tak to właśnie zrobiono?

— Spytaj programistę, on jest od programowania. Ja Ci tylko powiedziałem co wiem o swoim pochodzeniu, idę pozasuwać po ekranie, cześć.



Klasa Point raz jeszcze

```
class Point
{
  public:
    Point( int startX = 0, int startY = 0 );

    void setX( int newX );
    void setY( int newY );

    int  getX() const;
    int  getY() const;

  private:
    int  x, y;
};

Point::Point( int startX, int startY ) : x( startX ), y( startY )
{
}

inline void Point::setX( int newX )
{
  x = newX;
}

inline void Point::setY( int newY )
{
  y = newY;
}
```

Klasa Point raz jeszcze

```
inline int Point::getX() const
{
  return x;
}

inline int Point::getY() const
{
  return y;
}
```

Klasa ScreenPoint

```
class ScreenPoint : public Point
{
  public:
    ScreenPoint( int startX = 0, int startY = 0, char startShape = ' ' );

    void setX( int newX );
    void setY( int newY );

    void setShape( char newShape );
    char getShape() const;

    void show();
    void hide();

    bool isXOnScreen( int x );
    bool isYOnScreen( int y );

  private:
    char shape;
};

ScreenPoint::ScreenPoint( int startX, int startY, char startShape )
: Point( startX, startY ), shape( startShape )
{
}
```

Klasa ScreenPoint

```
void ScreenPoint::setX( int newX )
{
  if( isXOnScreen( newX ) )
    Point::setX( newX );
}

void ScreenPoint::setY( int newY )
{
  if( isYOnScreen( newY ) )
    Point::setY( newY );
}

void ScreenPoint::setShape( char newShape )
{
  if( newShape > 32 && newShape < 255 )
    shape = newShape;
}

char ScreenPoint::getShape() const
{
  return shape;
}

void ScreenPoint::show()
{
  con.writeCharXY( getX(), getY(), shape );
}
```

Klasa ScreenPoint

```
void ScreenPoint::hide()
{
  con.writeCharXY( getX(), getY(), ' ' );
}

bool ScreenPoint::isXOnScreen( int newX )
{
  return ( newX >= 1 && newX <= 80 );
}

bool ScreenPoint::isYOnScreen( int newY )
{
  return ( newY >= 1 && newY <= 24 );
}
```

Klasa Runner raz jeszcze, ale w nowej wersji

```
class Runner : public ScreenPoint
{
  public:
    Runner( int startX = 0 , int startY = 0, char startShape = ' ' );

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();
};

Runner::Runner( int startX, int startY, char startShape )
: ScreenPoint( startX, startY, startShape )
{
}

void Runner::moveUp()
{
  if( isYOnScreen( getY() - 1 ) )
  {
    hide();
    setY( getY() - 1 );
    show();
  }
}
```

Klasa Runner raz jeszcze, ale w nowej wersji

```
void Runner::moveDown()
{
  if( isYOnScreen( getY() + 1 ) )
  {
    hide();
    setY( getY() + 1 );
    show();
  }
}

void Runner::moveLeft()
{
  if( isXOnScreen( getX() - 1 ) )
  {
    hide();
    setX( getX() - 1 );
    show();
  }
}

void Runner::moveRight()
{
  if( isXOnScreen( getX() + 1 ) )
  {
    hide();
    setX( getX() + 1 );
    show();
  }
}
```