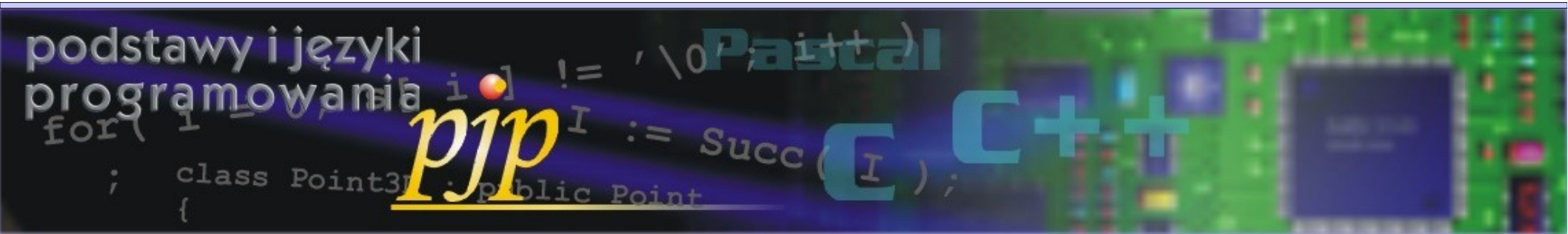
Projektowanie obiektowe

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

Obiektowość w systemach informatycznych



Dlaczego projektowanie jest ważne

Fakt 1: Wzrost złożoności oprogramowania

Złożoność oprogramowania ciągle rośnie. Jest to wynikiem:

rozwoju technicznego informatyki (nowe procesory, peryferia, nowe algorytmy)

oraz

wzrostu oczekiwań użytkowników (łatwiej, szybciej, wygodniej, inteligentniej).

Programming in the small $\rightarrow$ Programming in the large

Fakt 2: Wzrost nakładów na oprogramowanie

Nakłady finansowe na tworzenie, rozwój i pielęgnacje oprogramowania rosną

Dlaczego projektowanie jest ważne, cd.

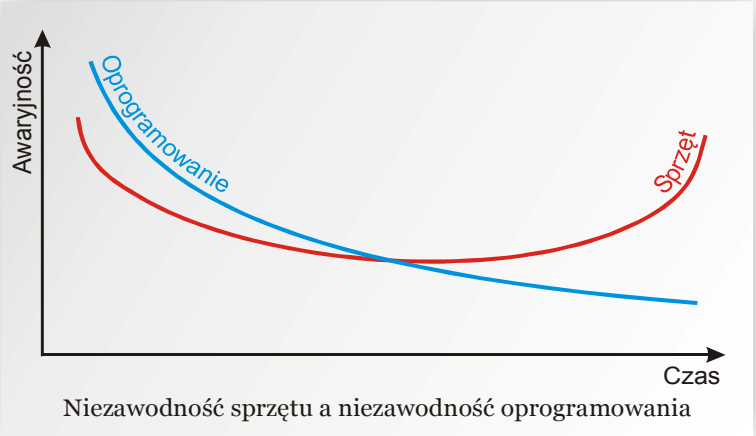
Fakt 3: Pogorszenie jakości i niezawodności oprogramowania

Nowe systemu tworzone są pośpiesznie w warunkach radykalnego wzrostu oczekiwań użytkowników.

Dotychczas sprawdzone metody i techniki programowania okazują się niewystarczające.

Fakt 4: Nowe wersje oprogramowania pojawiają się w coraz krótszych odstępach czasu

Nowe wersje są zbyt krótko i pobieżnie testowane oraz muszą być — ze względów komercyjnych — coraz bogatsze.

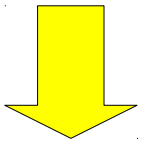


Niezawodność sprzętu a niezawodność oprogramowania

Podejścio obiektowe

Potrzebne jest takie podejście do programowania, które:

- lepiej uwzględnia i odwzorowuje oczekiwania użytkownika,
- w sposób konsekwentny zachowuje terminologię dziedziny problemu,
- pozwala zarządzać dużym i złożonym kodem źródłowym,
- jest w stanie sprostać rozwojowi technicznemu informatyki,
- pozwala na wielokrotne wykorzystanie tego samego kodu.



Obiektowe podejście do projektowania systemów informatycznych:

Analiza obiektowa — OOA (ang. *Object Oriented Analysis*),

Projektowanie obiektowe — OOD (ang. *Object Oriented Design*),

Programowanie obiektowe — OOP (ang. *Object Oriented Programming*).

OOA, OOD, OOP

Rola obiektu

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

Analiza obiektowa

Celem analizy obiektowej jest stworzenie modelu, w którym każdy element świata rzeczywistego odwzorowany jest odpowiednim obiektem (lub ich grupą), należącym do określonej klasy. Interakcja pomiędzy obiektami jest odbiciem procesów zachodzących w rzeczywistości.

Projektowanie obiektowe

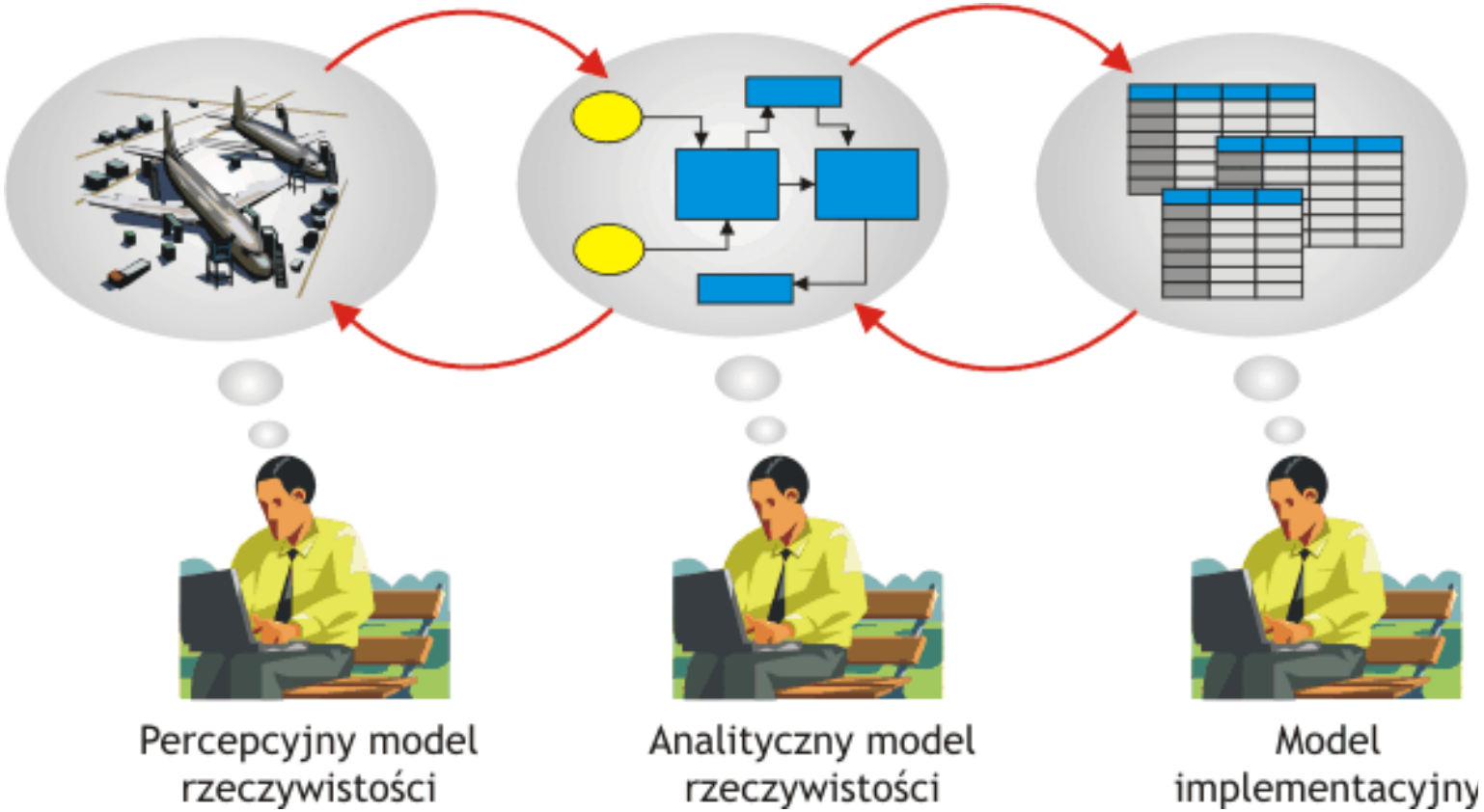
Celem projektowania obiektowego jest opisanie struktury projektowanego systemu w kategoriach obiektów i klas, do których należą, stosując odpowiednie notacje, odwzorowujące logiczne i fizyczne właściwości obiektów oraz opisujące ich właściwości statyczne i dynamiczne.

Programowanie obiektowe

Programowanie obiektowe zakłada organizację programu jako zbioru współpracujących ze sobą obiektów, będących reprezentantami pewnych klas. Klasy te należą do hierarchii klas, opisanej zunifikowanymi sposobami odwzorowania relacji występujących pomiędzy klasami.

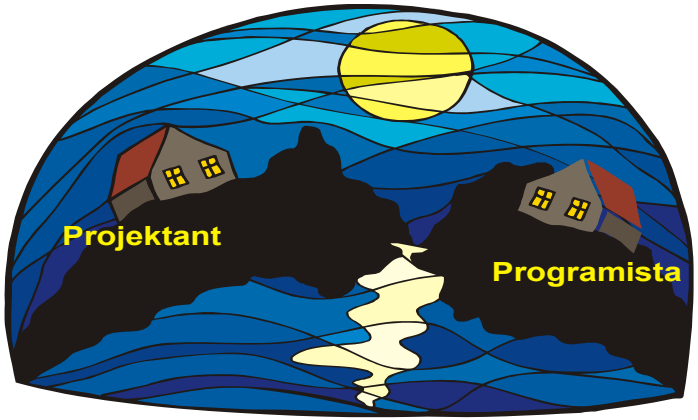
Projektowanie jako opracowywanie modelu

Celem modelowania systemów informatycznych jest stworzenie dokładnego opisu systemu, na który składa się opis wymagań i oczekiwań użytkowników implikujący opis zakresu systemu, jego funkcji, interakcji z otoczeniem.



Dlaczego projektowanie jest ważne

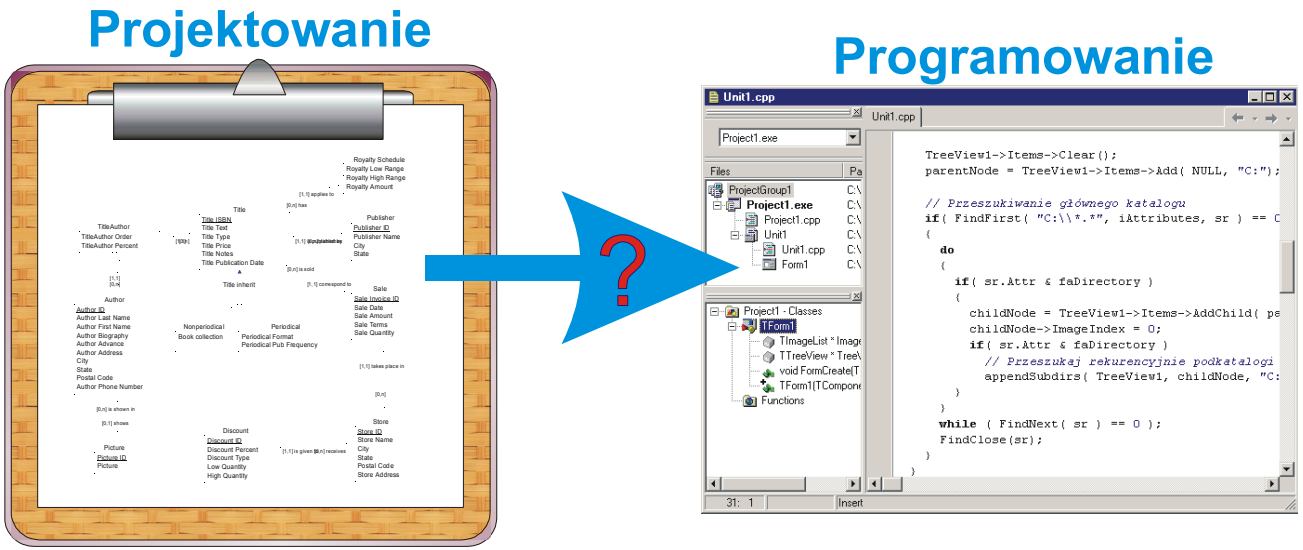
Wielki kanion



Świat projektanta:
modele, procesy, przepływy, encje, związki, stany, przejścia, diagramy, schematy, relacje, ...

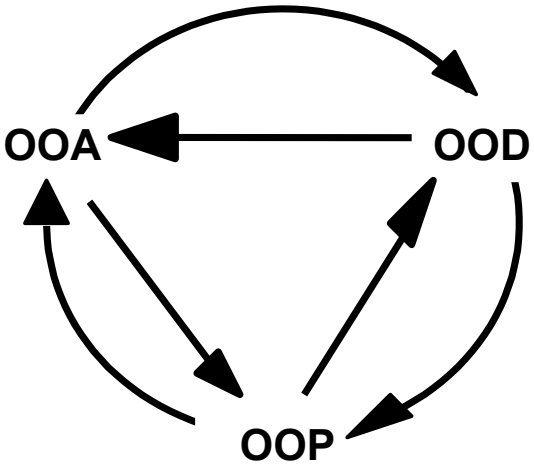
Świat programisty:
moduły, funkcje, procedury, typy zmienne, instrukcje, tabele. rekordy, klucze, ...

Zmiana specyfikacji



Dlaczego projektowanie jest ważne

Ciągłość specyfikacji



Model piłeczki baseball'owej

Podejści obiektowe

Bazuje na obiektach a nie algorytmach — podstawowymi blokami logicznymi programu są zatem *obiekty* a nie *podprogramy*.

Każdy obiekt jest reprezentantem pewnej *klasy*, klasa określa *właściwości* i *możliwości* każdego obiektu.

Klasy są ze sobą powiązane tworząc pewną *hierarchię*, podstawą do jej budowania relacje pomiędzy klasami, główną jest *dziedziczenie* właściwości klas oraz związki *całość-część*.

Programowanie obiektowe — trochę historii

- Programowanie obiektowe nie jest nowością ostatnich lat — obiektowość w programowaniu pojawia się mniej lub bardziej jawnie od wielu lat.
- Pierwszym znanym językiem *wykorzystującym obiekty* był język *Simula67*, opracowany w latach 60-tych w Norweskim Ośrodku Obliczeniowym w Oslo przez O-J. Dahla, B. Myhrhauga i K. Nygaarda.
- Termin *object oriented programming* pojawił się w latach 70-tych wraz z językiem *Smalltalk*, wywodzącym się z Xerox Palo Alto Research Center.
- Język *Smalltalk,* inspirowany językiem *Simula 67*, posiada szereg interesujących właściwości. Oprócz cech obiektowych posiada rozbudowane biblioteki, środowisko programistyczne, działa z wykorzystaniem maszyny wirtualnej.
- Mimo swojego nowatorstwa jak i zaawansowania technologicznego *Smalltalk* nie zdobywa popularności, staje się jednak *źródłem inspiracji* dla wielu innych języków obiektowych.

Programowanie obiektowe — trochę historii, cd...

- Koncepcja obiektowości w programowaniu znajduje wielu zwolenników. Powstają obiektowe rozszerzenia istniejących języków programowania (np. *Object Pascal*, *C++*, *Objective-C*).
- Powstają nowe języki, rdzennie posiadające wbudowane mechanizmy obiektowe. Są to zarówno języki ortodoksyjnie obiektowe (np. *Oberon*, *Ruby*), jak i języki wywodzące się z nurtu hybrydowego (np. *Java*, *JavaScript*, *C#*, *PHP*).
- Obiektowość w programowaniu jest jednolita na poziomie koncepcji, jednak w zakresie rozwiązań szczegółowych i implementacyjnych występują różnice.
- Popularne aktualnie języki programowania, takie jak *Java*, *C#*, *PHP*, *JavaScript*, *ActionScript*, wykorzystują (w różnym co prawda stopniu) koncepcję programowania obiektowego wywodzącą się z języka *C++*.

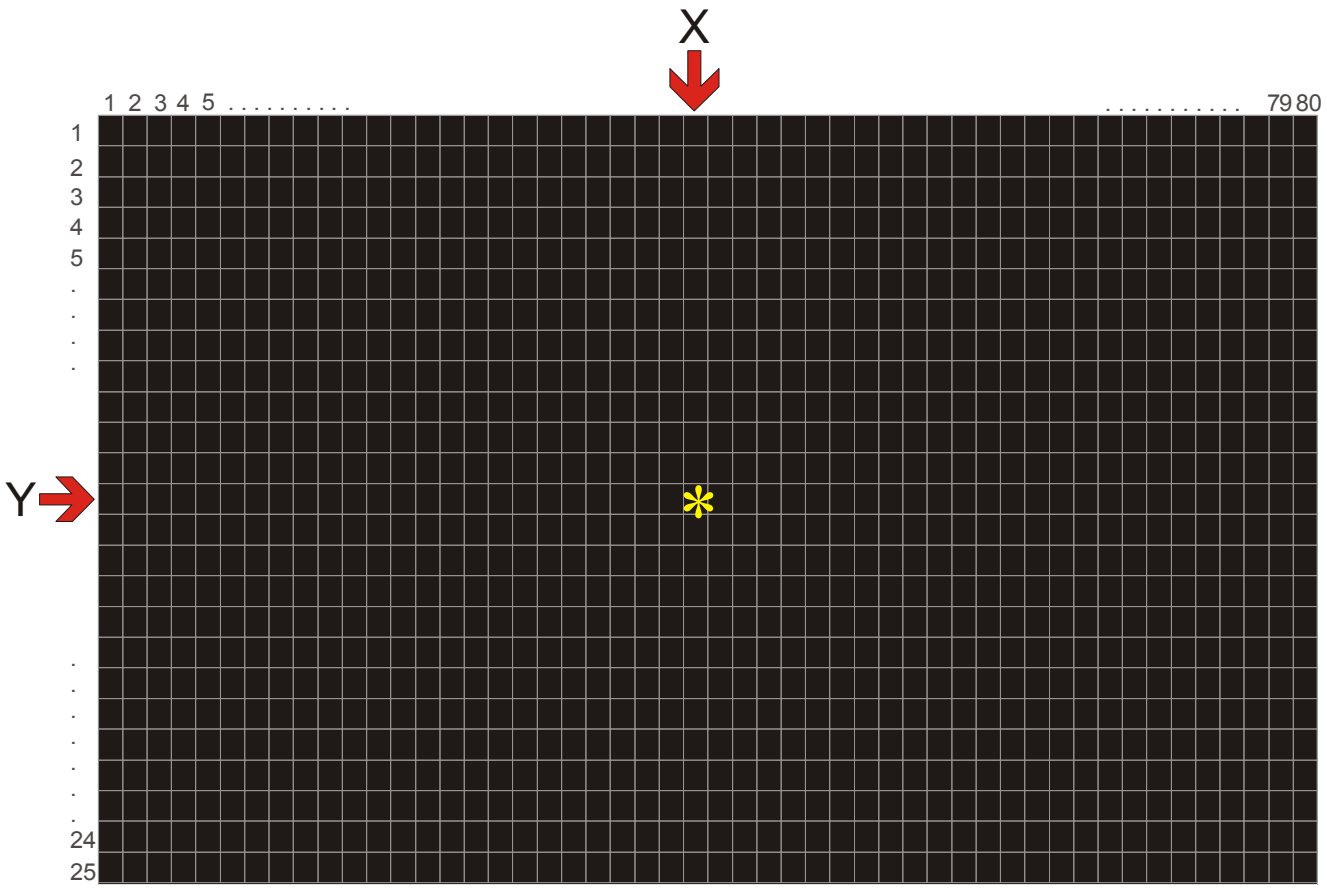
Programowanie obiektowe — ogólna koncepcja

- Co to jest obiekt?
- Jak jest zbudowany?
- Jak działa?

Koncepcja programowania obiektowego jest prosta i naturalna. Aby ją sugestywnie przedstawić, proponuję rozważenie pewnego przykładu, zrealizowanego najpierw strukturalnie a potem obiektowo.

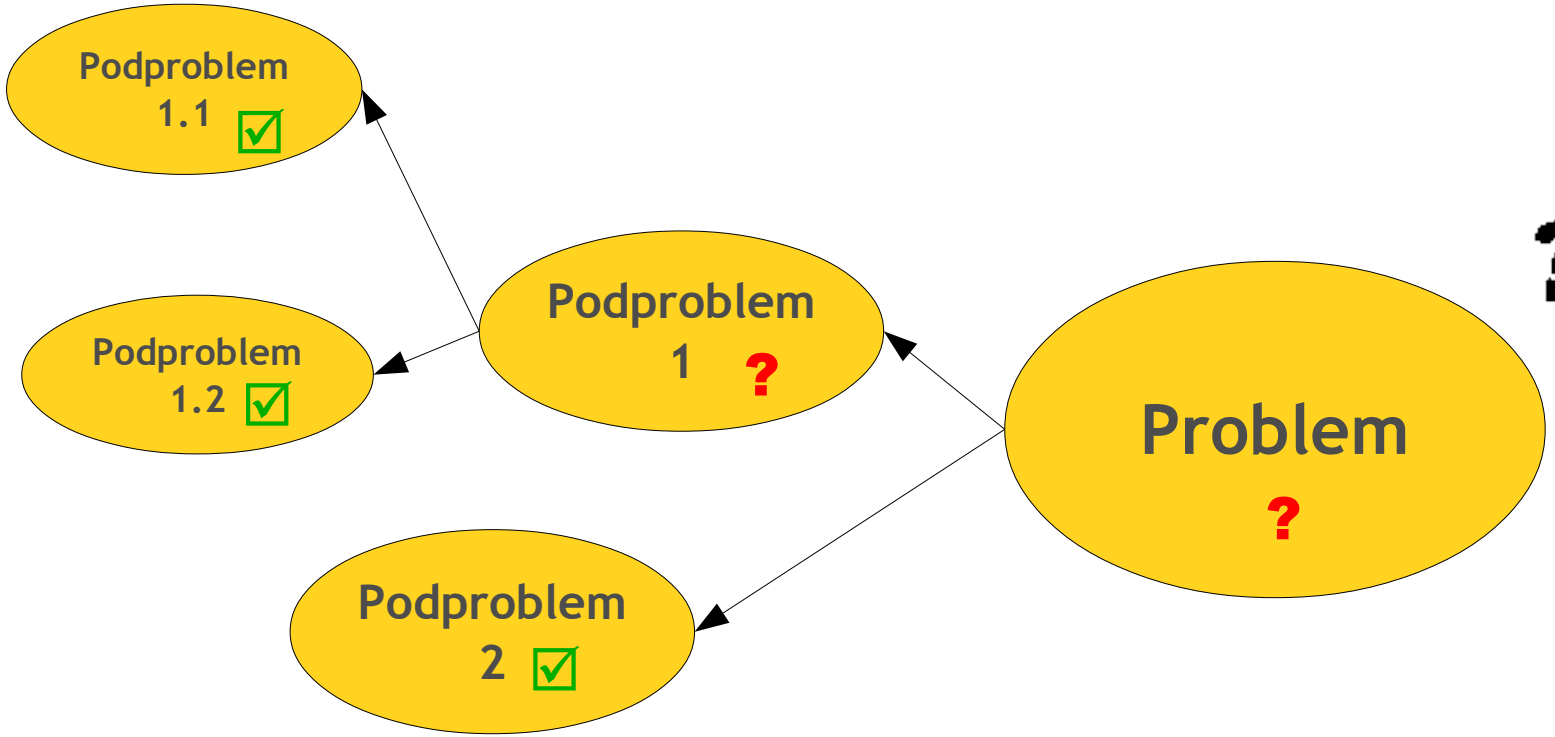
Przykład

Napisać program wyświetlający na pustym ekranie tekstowym zadany znak (np. '*'), pozwalający na przesuwanie tego znaku w górę, dół, lewo i prawo, przy użyciu klawiszy sterowania kursorem.



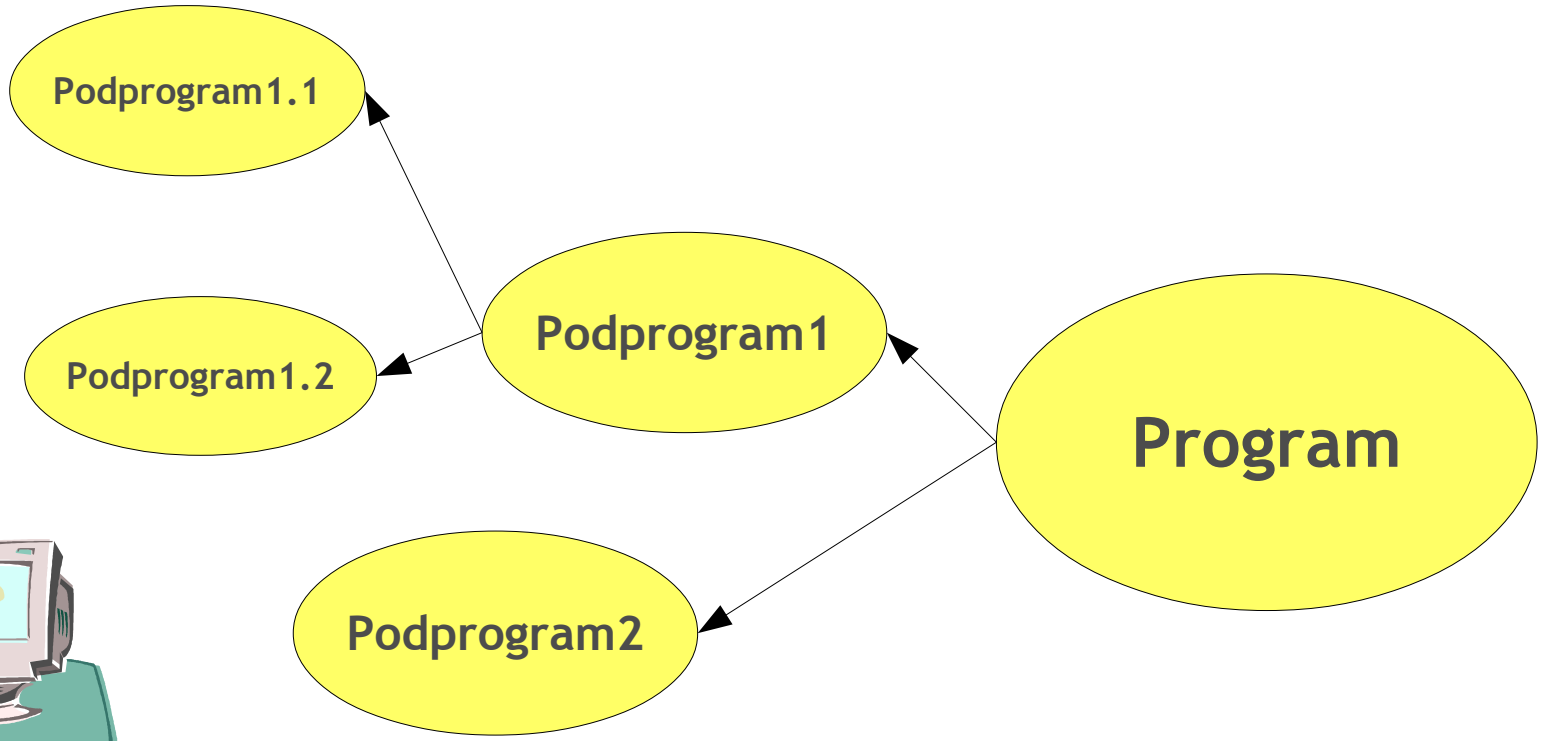
Dekompozycja problemu — z mniejszym być może łatwiej walczyć

- Rozwiązanie zadanego problemu w programowaniu polega na wykonaniu sekwencji pewnych *czynności*, zgodnie z algorytmem rozwiązania tego problemu.
- *Dekompozycyjna strategia rozwiązywania problemów* polega na dzieleniu potencjalnie *dużego i złożonego problemu na podproblemy* tak długo, aż wyodrębnione podproblemy staną się proste do rozwiązania.



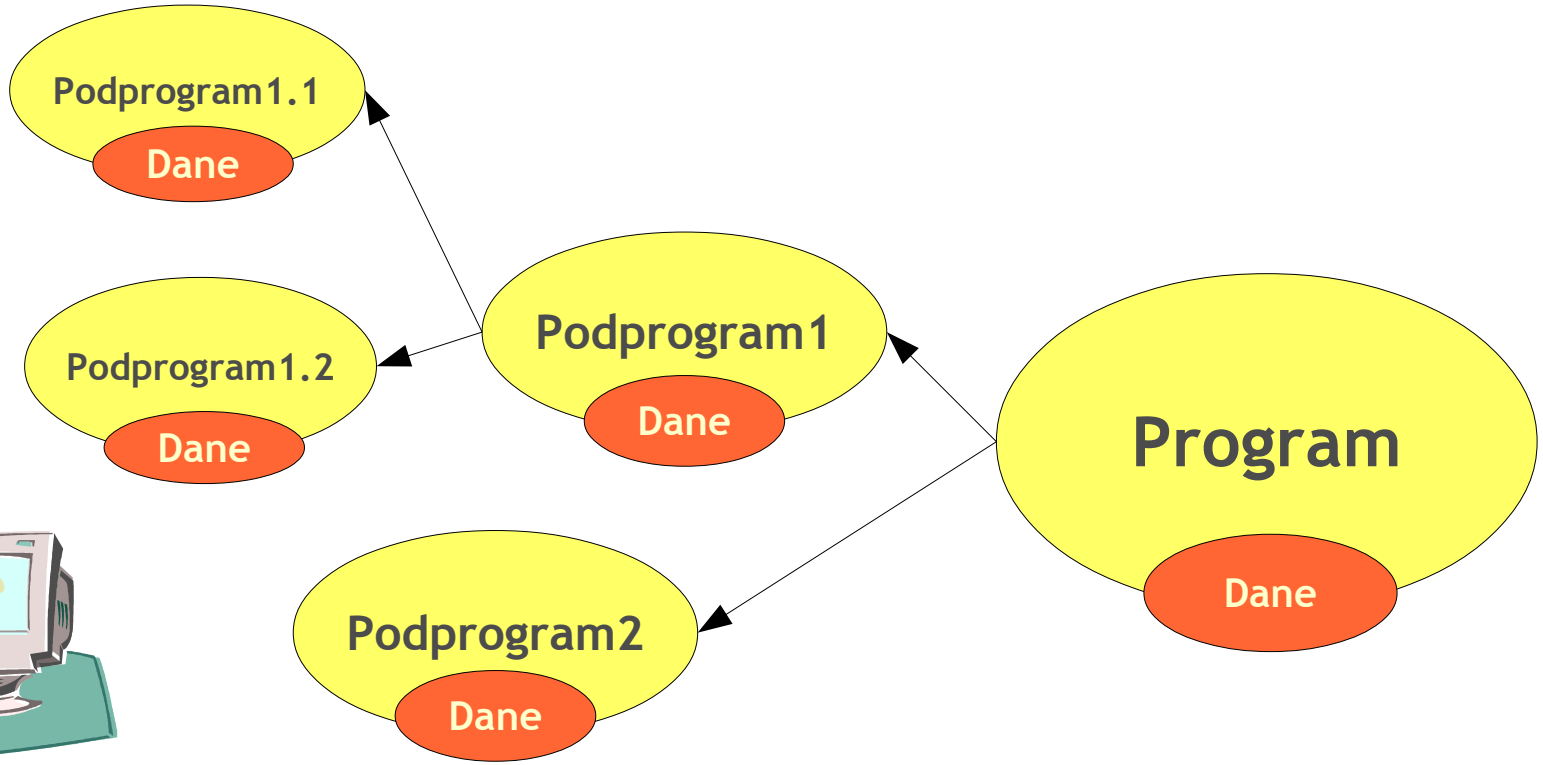
Podejście proceduralne

- *Proceduralne podejście* do programowania zakłada, że złożone czynności niezbędne do rozwiązania problemu będą opisane *procedurami (podprogramami),* rozbijanymi na kolejne *podprocedury* tak długo, aż uzyskane zostaną czynności proste, łatwe do zaprogramowania.



Podejście strukturalne

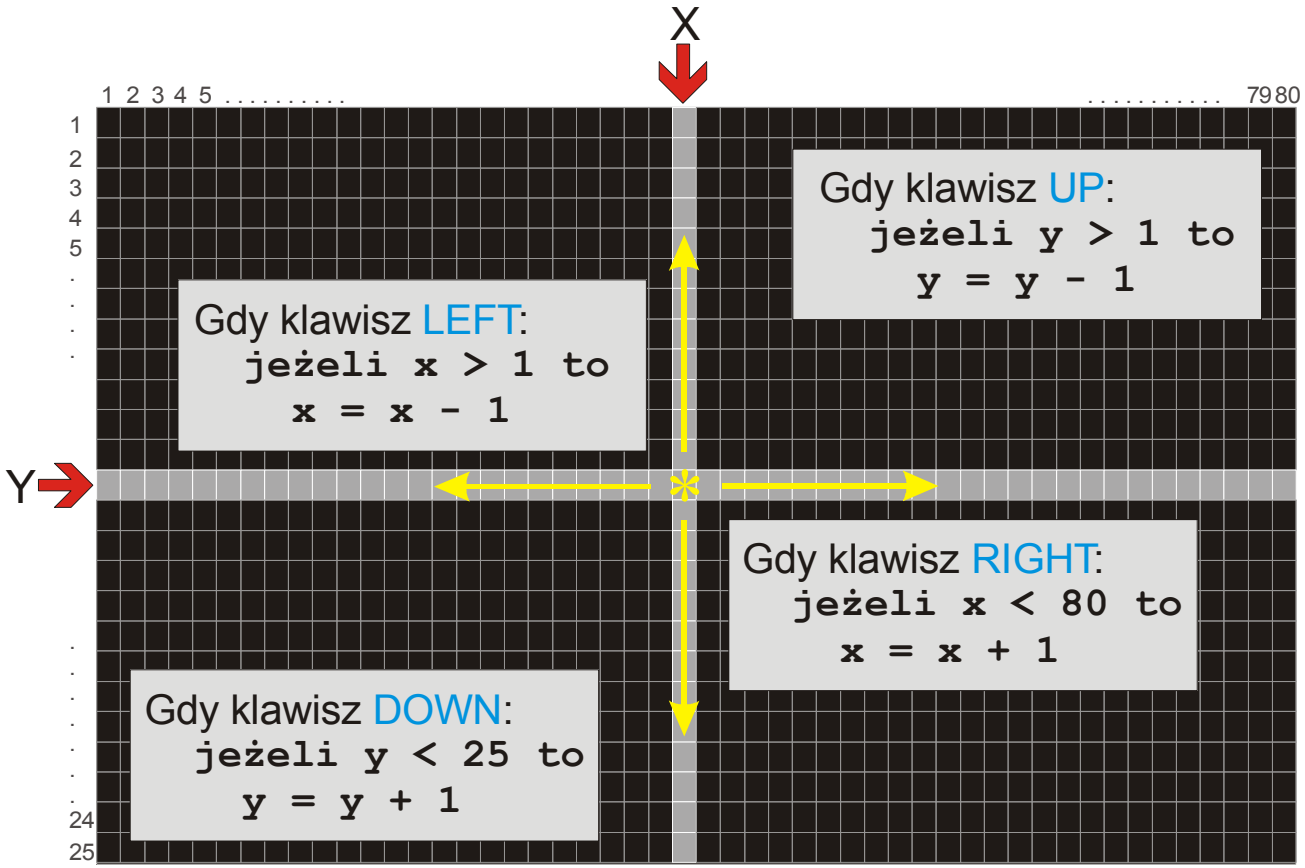
- *Podejście strukturalne* rozwija ideę programowania proceduralnego, zakładając tworzeniu programów w taki sposób, że blok główny programu stanowi szkielet, określający *kolejność realizacji czynności*, zapisanych w postaci odpowiednich procedur i/lub funkcji.
- Podprogramy wykorzystują *globalne* i *lokalne* struktury danych.



Analiza — dane i czynności związane z przesuwaniem punktu

- Pozycja przesuwanego znaku będzie opisana parą zmiennych **`x`** i **`y`**, kod tego znaku (wygląd) będzie opisany zmienną **`shape`**.

- Przesuwanie znaku polega na wymazywaniu znaku na pozycji poprzedniej i ponownym wyświetlaniu na pozycji nowej, o ile jest ona prawidłowa.

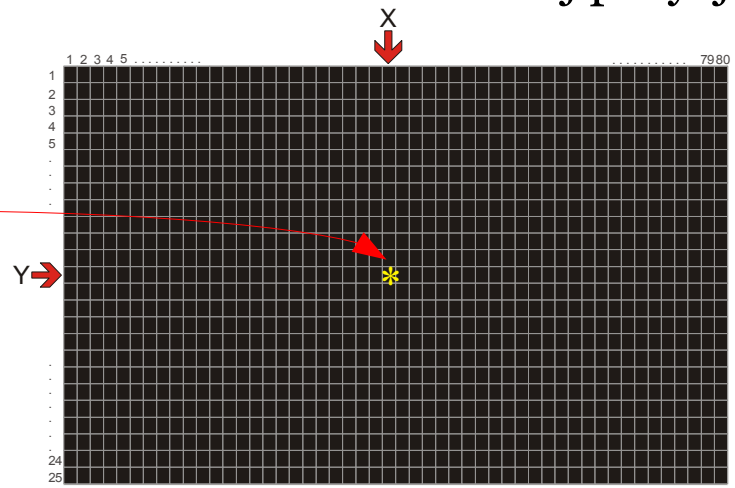


Analiza — wymagane funkcje obsługi konsoli

Zakładamy, że mamy do dyspozycji funkcje obsługi ekranu konsoli:

- Funkcję **`clearScreen`**, czyszczącą ekran tekstowy.
- Funkcję **`writeCharXY`**, wyświetlającą na ekranie znak na określonej pozycji.

```
writeCharXY( 40, 12, '*' );
```



Analiza — wymagane funkcje obsługi konsoli

Zakładamy, że mamy do dyspozycji obsługę klawiatury konsoli:

- Funkcję **getKey**, której rezultatem jest kod klawisza, w tym tzw. *klawiszy sterujących kursorem* (czytanie znaku bez echa na ekranie i bez "potwierdzania" Enterem).
- Stałe określające kody klawiszy, wtym klawiszy sterujących kursorem (dla funkcji **getKey**): **KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT, KEY_ESC.**

```
int  key;
key = getKey(); /* Czekaj na naciśnięcie klawisza */
switch( key )
{
  case KEY_UP    : /* Obsługa strzałki w gorę  */
                   break;
  case KEY_DOWN  : /* Obsługa strzałki w dół   */
                   break;
  case KEY_LEFT  : /* Obsługa strzałki w lewo  */
                   break;
  case KEY_RIGHT : /* Obsługa strzałki w prawo */
                   break;
}
```

Przykładową implementację funkcji **getKey** zawiera suplement umieszczony na końcu tego dokumentu.

Realizacja strukturalna — najprostsza wersja

```
int  key, x = 40, y = 12;
char shape = '*';

clearScreen();

do
{
  writeCharXY( x, y, shape );   /* Wyświetl znak na zadanej pozycji */
  key = getKey();               /* Czekaj na naciśnięcie klawisza   */
  writeCharXY( x, y, ' ' );     /* Wymaż znak, potencjalny ruch     */

  switch( key )
  {
    case KEY_UP    : if( y > 1 ) --y;   /* Przeskocz wyżej   */
                     break;
    case KEY_DOWN  : if( y < 24 ) ++y;  /* Przeskocz niżej   */
                     break;
    case KEY_LEFT  : if( x > 1 ) --x;   /* Przeskocz w lewo  */
                     break;
    case KEY_RIGHT : if( x < 80 ) ++x;  /* Przeskocz w prawo */
                     break;
  }
}
while( key != KEY_ESC );
```

Realizacja strukturalna — wady

Wady techniczne

- Brak jawnego powiązania pomiędzy zmiennymi `x`, `y` i `shape`. A przecież te zmienne opisują jeden obiekt!
- Brak jawnego powiązania pomiędzy fragmentem wyświetlającym (ukrywającym) punkt oraz zmiennymi go opisującymi.
- Siermiężne testowanie poprawności współrzędnych.

Wady metodyczne

- Z kodu programu nie wynika bezpośrednio, jakie jest jego przeznaczenie, brak powiązań pomiędzy wykonywanymi akcjami a danymi.
- Sam algorytm jest prosty, lecz jego zapis nie wspomaga zrozumienia jaka jest istota rozwiązywanego problemu.

Od programowanie proceduralnego do obiektowego

Podejdźmy do problemu inaczej — spróbujmy zrozumieć jego istotę

Zapraszamy na:

Krótki wywiad z głównym bohaterem programu!

Czwartek
24 maja 2011

Redaktor Prowadzący
Roman Simiński

Fakty IT

gazeta codzienna

Niecodzienny dodatek *Gazety Codziennej*

Bohaterem dzisiejszego wywiadu jest ruchomy punkt ekranowy. Dzięki temu wyjątkowemu wywiadowi poznamy unikatowe fakty z życia przedstawicieli tego gatunku.

Tylko u nas, ekskluzywny wywiad z głównym bohaterem Programu

Jak się nazywasz?

— No, wiesz, na imię mam nieciekawie ... jestem *R*, po prostu *R*.

Kim jesteś?

— Jestem ruchomym punktem ekranowym. Ponieważ zasuwam po całym ekranie, można o mnie powiedzieć z angielska, że jestem jednym z Runner'ów.

Czyli może Was być więcej?

— Takich jak ja może być więcej. W tym programie potrzeba tylko jednego Runner'a, więc oto jestem.

Powiedz nam coś o sobie...

— Co tu dużo gadać. Jak trzeba to mnie widać, jak nie trzeba to znikam. Potrafię się przesunąć o jedno pole w górę, w dół, w lewo lub w prawo. Nie wylecę poza ekran — to pewne — dbam o to. Możesz mi podać pozycję ekranową oraz znak, jakim mam być.

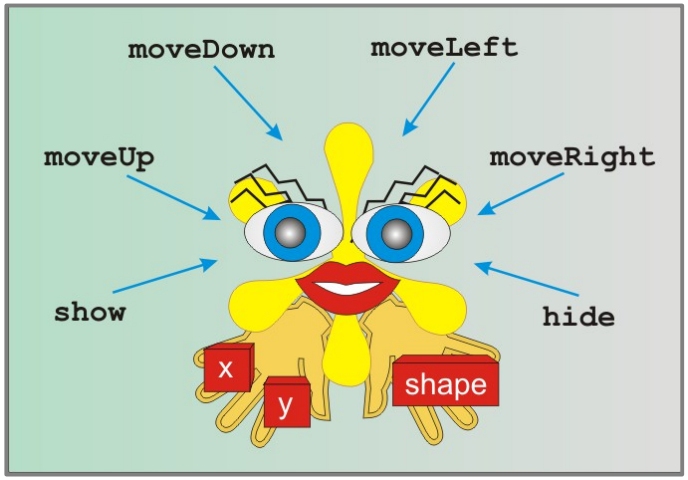
gazeta codzienna

Jak to „możesz mi podać”?

— Co jak to... zwyczajnie. Przecież wiesz jak mam na imię, to mnie zawołaj, ustal gdzie mam być na ekranie i jak mam wyglądać. W ogóle to możesz mną sterować, zawołaj mnie i prześlij mi *komunikat*, rozumiesz, takie polecenie co mam zrobić.

Jak niby mam Ci przesyłać komunikat?

— Jaki ty jesteś... no, niedoinformowany. Przecież już Ci mówiłem, co potrafię. Dokładnie komunikaty nazywają się: *show*, *hide*, *moveUp*, *moveDown*, *moveLeft*, *moveRigt*. Dodatkowo mam swoją wewnętrzną pamięć i trzymam tam wszystko, co powinienem wiedzieć. Pozycję przechowuję w komórkach o nazwie *x* i *y*, a znak jakim mam być, znaczy, wyglądać, w szufladce *shape*. Chcesz, to mi coś tam wstaw. Co łaska, byleby to było legalne. Wiesz, my gwiazdy programów musimy uważać ... .



A skąd Ty to wszystko wiesz i potrafisz?

— Nauczył mnie tego programista. Wymyślił, że będą Runner’y, określił, co każdy Runner powinien wiedzieć, co powinien robić. A potem nauczył nas, Runner’ów jak to wszystko mamy robić. Ja jestem Runner'em, to umiem to wszystko. Każdy Runner umie to wszystko. No, przydałoby się, żeby nauczyła nas jeszcze kilku rzeczy..., no wiesz... ale nie nauczył :(.

gazeta codzienna

A skoro już tu jesteś... mogę trochę Ciebie poużywać?

— Taaak, Wy to, jak zwykle, zawsze o jednym. A dobra masz mieć, nie jestem mydełko, nie wymydlę się. Ale ostrożnie, najpierw Ci wszystko wyjaśnię. Jeżeli chcesz, żebym wiedział, że mam się pojawić np. w 40-stej kolumnie 12-go wiersza, to piszesz w swoim programie:

```
R.x = 40;
R.y = 12;
```

— Zauważ, że piszesz najpierw moje imię, potem po kropce nazwę miejsca, gdzie przechowuję dane. To miejsce nazywam *polem*. Możesz do tego pola przypisać wartość np.: 40, 12. To tak, jakbyś mi wkładał tą wartość do kieszeni.

Jeżeli chcesz, żebym pokazał się na ekranie piszesz:

```
R.show();
```

— I znowu, piszesz moje imię, potem po kropce nazwę polecenia, jakie mam wykonać. To tak jakbyś do mnie wysyłał *komunikat*. To wygląda trochę jak wywołanie funkcji w języku C..., i rzeczywiście, na Twój *komunikat* odpowiadam wywołaniem funkcji, którą mam w siebie wbudowaną. Taka funkcja nazywa się *funkcją składową*.

gazeta codzienna

— Jeżeli mam się przesunąć o jedno pole do góry, potem o jedno w lewo i jeszcze raz o jedno do góry, to piszesz:

```
R.moveUp();
R.moveLeft();
R.moveUp();
```

— Jeżeli chcesz, żebym przeszedł po kwadracie: pięć kroków w prawo, pięć w dół, pięć w lewo i pięć w górę, to napiszesz np.:

```
int steps;

R.x = 1;  // Powiedz mi, skąd mam wystartować
R.y = 1;

R.show(); // Powiedz żebym się pokazał

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w prawo
  R.moveRight(); // Krok w prawo

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w dół
  R.moveDown();  // Krok w dół

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w lewo
  R.moveLeft();  // Krok w lewo

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w górę
  R.moveUp();    // Krok w górę
```

Czy możesz naszym czytelnikom wyjaśnić jak są zbudowane funkcje składowe, np. *show*, *hide*?

— Mogę..., ale mi się nie chce. Zapytaj o to programistę. Ja jestem od zasuwania na ekranie a nie od programowania.

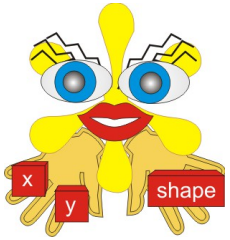
Szkoda..., ale i tak serdecznie dziękujemy za rozmowę.

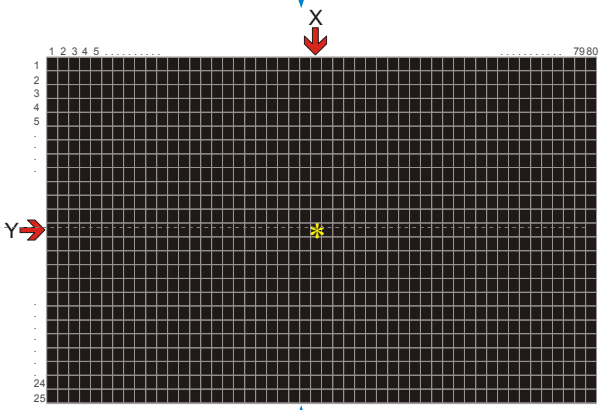
Realizacja obiektowa — najprostsza wersja

```
Runner R; // Utworzenie obiektu o nazwie R i typie Runner

R.x = 40; // Ustalenie pozycji i wyglądu Runnera'a R
R.y = 12;
R.shape = '*';

clearScreen();
R.show();  // Niech R się pokaże
do
{
  key = getKey();
  switch( key )
  {
    case KEY_UP    : R.moveUp();    // Niech R się przesunie w górę
                     break;
    case KEY_DOWN  : R.moveDown();  // Niech R się przesunie w dół
                     break;
    case KEY_LEFT  : R.moveLeft();  // Niech R się przesunie w lewo
                     break;
    case KEY_RIGHT : R.moveRight(); // Niech R się przesunie w prawo
                     break;
  }
}
while( key != KEY_ESC );
```

Obiektowo vs. strukturalnie

```
R.x = 40;
R.y = 12;
R.shape = '*';

R.show();
. . .
R.moveUp();
. . .
R.moveDown();
. . .
R.hide();
. . .
```





Obiektowo

Strukturalnie

```
int key, x = 40, y = 12;
int shape = '*';

clearScreen();

writeCharXY( x, y, shape );
key = getKey();
writeCharXY( x, y, ' ' );
. . .
if( y > 1 ) --y;
. . .
if( y < 24 ) ++y;
. . .
```

Obiektowo vs. strukturalnie

Obiektowo

- Wyodrębnij najważniesze *obiekty* w rozważanym problemie, ustal co powinny *wiedzieć* oraz co powinny *umieć*. *Naucz* je tego wszystkiego.
- Jeżeli problem tego wymaga, połącz obiekty w *grupy*, ustal *szefa grupy* — jemu będziesz przekazywała *polecenia*, on będzie *zarządzał* pracą grupy.
- Utwórz potrzebne *obiekty*, przekaż im konieczne *informacje* i każ im *pracować* wysyłając odpowiednie *komunikaty*, tak zby rozwiązać rozważany problem.

Strukturalnie

- Ustal jakie *dane* są istotne dla rozważanego problemu, dobierz odpowiednie *struktury* do ich przechowywania.
- Ustal jakie *czynności* powinny być wykonane na *danych*, jaka powinna być ich *kolejność*. Jeżeli ustalone czynności są *złożone*, dziel je na *podczynności* tak długo, aż będą łatwe do zaprogramowania. Zbuduj odpowiednie podprogramy.
- Uaktywnij zaplanowane *czynności* w ustalonej *kolejności*, tak aby rozwiązać rozważany problem.

Programming in the small → programming in the large.

- Programowanie *proceduralne* jest zorientowane na *czynności* wykonywane na *strukturach danych*.
- Czynności są realizowane przez *podprogramy* (*procedury* i *funkcje*) grupowane zwykle w *moduły*.
- Programowanie *proceduralne* wraz ze *wzrostem rozmiaru programu* zaczyna być coraz *trudniejsze*.
- Coraz więcej *podprogramów*, coraz więcej *modułów*, coraz więcej nieintuicyjnych i niejasnych *powiązań* pomiędzy podprogramami a strukturami danych.

Potrzebujemy ***porządku*** i ***hierarchii*** — trzeba jasno określić *kto* i *za co* jest odpowiedzialny.

W programie muszą się pojawić *kompetentne podmioty*, zarówno te *zarządzające* (kadra „managerska”) jak i te *zarządzane*.

W programowaniu *podmiot*, czyli *ktoś* to *obiekt*.

Obiektowość — analiza, projektowanie, programowanie

- Obiektowość jest strategią pozwalającą zapanować na złożonym projektem.
- Może być stosowana w każdej z faz realizacji systemu, zapewniając spójny i jednolity opis systemu w każdej z tych faz.
- W wymiarze czysto programistycznym obiektowość jest strategią organizacji kodu potencjalnie *złożonego* oprogramowania.
- Programowanie obiektowe *nie stoi w sprzeczności* z ideą programowania strukturalnego, stanowi raczej jego *nadbudowę*.
- Zazwyczaj realizacja *metod obiektów* w systemie przebiega zgodnie z koncepcją programowania *strukturalnego*.

Programowanie obiektowe — *strategiczne* narzędzie realizacji złożonych systemów.

Programowanie proceduralne — *taktyczne* narzędzie realizacji mniejszych systemów, bądź *wybranych elementów* systemów złożonych.

Obiektowość — przepis na program obiektowy raz jeszcze

- Wyodrębnij najważniesze *obiekty* w rozważanym problemie, ustal co powinny *wiedzieć* oraz co powinny *umieć*.
- Jeżeli problem tego wymaga, połącz obiekty w grupy, ustal szefów grup — im będziesz przekazywała *polecenia,* oni będą *zarządzali* pracą grupy.
- Sprawdź czy istnieją odpowiadające Ci *gotowe szablony* dla potrzebnych obiektów. Jeżeli tak to *użyj ich*, jeżeli nie to zbuduj *własne szablony*.
- Utwórz potrzebne *obiekty*, przekaż im konieczne *informacje* i każ im *pracować* wysyłając odpowiednie *komunikaty*, tak zby rozwiązać rozważany problem.

Wyodrębniliśmy już najważniejszy obiekt w naszym problemie, wiadomo co powinien *wiedzieć* i co powinien *umieć*. Jak go tego nauczyć?

Obiekt — krótka definicja

Istnieje wiele, różniących się od siebie definicji pojęcia *obiekt* i *klasa*. Tutaj przyjmiemy następującą definicje:

Obiekt jest programowym tworem, stanowiącym *model* pewnego konkretnego bytu ze świata rzeczywistego, reprezentujący *rzecz* (obiekt fizyczny), *pojęcie* (obiekt konceptualny) lub obiekt jest pewnym *bytem programistycznym* nie mający swojego odpowiednika w rzeczywistości.

- W sensie programistycznym *obiekt* jest *zmienną złożoną*, zawierającą dane różnego typu. Jest zatem podobny do *rekordu*.
- Od rekordu różni go *zdolność do działania*, polegająca na możliwości przesłania do obiektu *komunikatu*, na który reaguje on *uruchomieniem* wbudowanej weń *funkcji składowej* (*metody*).

Klasa — krótka definicja

Klasa to szablon określający jak obiekt jest *zbudowany* i *co potrafi*.

W szczególności klasa zawiera informacje o nazwach, typie i reprezentacji pól, deklaracje i definicje funkcji składowych oraz informacje dotyczące relacji i powiązań z innymi klasami.

- W sensie programistycznym *klasa* jest dla *obiektu* tym, czym *typ* dla *zmiennej*.
- W języku Object Pascal mówi się: *zmienna obiektowa* i *typ obiektowy*, w C++ i pokrewnych językach obiektowych mówi się po prostu *obiekt* i *klasa*.
- Klasa ma charakter pierwotny — aby *utworzyć obiekt*, najpierw należy *zdefiniować jego klasę*.

Trzy podstawowe cechy obiektu

- Tożsamość (ang. *identity*).
- Stan (ang. *state*).
- Działanie, zachowanie (ang. *behavior*).

Tożsamość obiektu

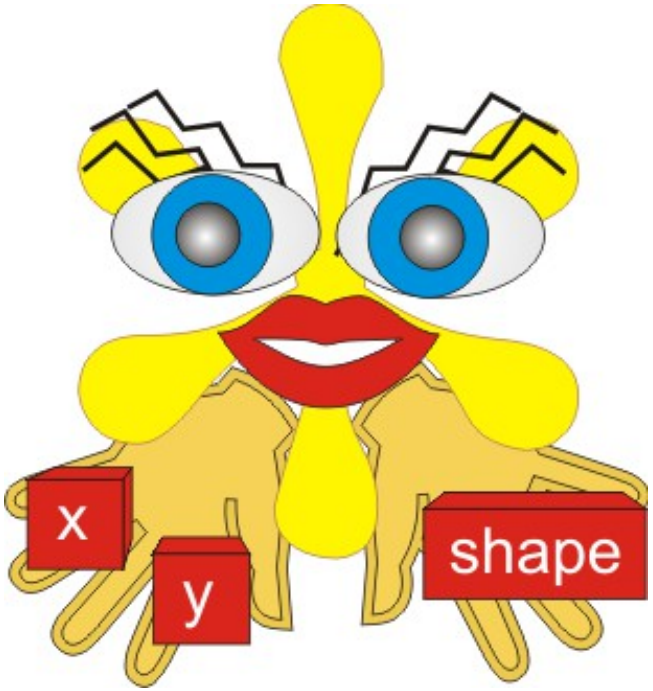
- *Tożsamość* (ang. *identity*), pozwala na odróżnienie danego obiektu od każdego innego.
- Tożsamość jest potrzebna, aby można było obiekt *zlokalizować* i *wykorzystać*.
- Programista przypisuje obiektowi *unikatową nazwę* lub włącza go do *identyfikowalnej kolekcji obiektów*, lub odwołuje się do obiektu posługując się *informacją o jego adresie* w pamięci operacyjnej.

```
Runner R;
. . .
R.show();
. . .

Runner duchy[ 3 ];
for( int i = 0; i < 3; i++ )
  duchy[ i ].show();
. . .

Runner * wsk = new Runner;
. . .
wsk->show();
. . .
```

Stan obiektu

- *Stan* (ang. *state*) obiektu w danym momencie opisany jest wartością jego pól.
- Pola obiektu nazywane są również *atrybutami* lub *składowymi danych*.
- Wartości pól oczywiście mogą się zmieniać w czasie wykonania programu, oznacza to, że obiekt zmienia swój stan.

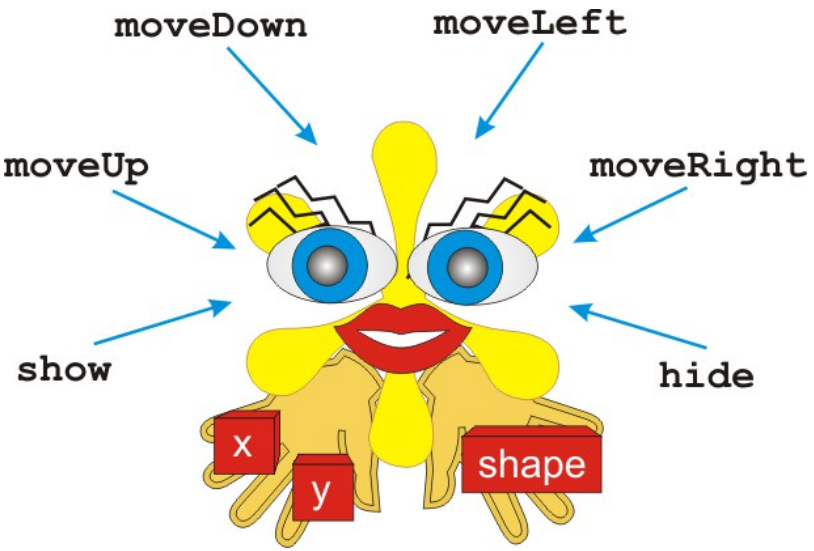
```
Runner R;

R.x = 1;
R.y = 1;
R.shape = '@';
R.show();
. . .

R.x = 5;
R.y = 5;
R.shape = '$';
R.show();
. . .
```



Zdolność obiektu do działania

- *Zdolność do działania* (ang. *behavior*) opisuje *operacje*, które obiekt może *wykonać*.
- Operacje są realizowane poprzez wykonanie *metod* zwanych w C++ *funkcjami składowymi*.
- Metody są częścią obiektu i mogą być aktywowane wtedy, gdy obiekt otrzyma *komunikat*, zgodny z nazwą metody, która ma być wykonana.
- Wykonanie metod zwykle *zmienia stan* obiektu poprzez zmianą *wartości jego pól*.

```
Runner R;
. . .
R.show();
R.moveLeft();
R.moveLeft();
R.moveUp();
. . .
R.hide();
```



Najważniejsze cechy obiektowego podejścia do programowania

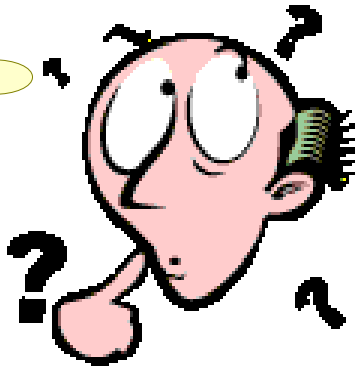
- Abstrakcja (ang. *abstraction*).
- Hermetyzacja (ang. *encapsulation*).
- Dziedziczenie (ang. *inheritance*).
- Polimorfizm (ang. *polymorphism*)

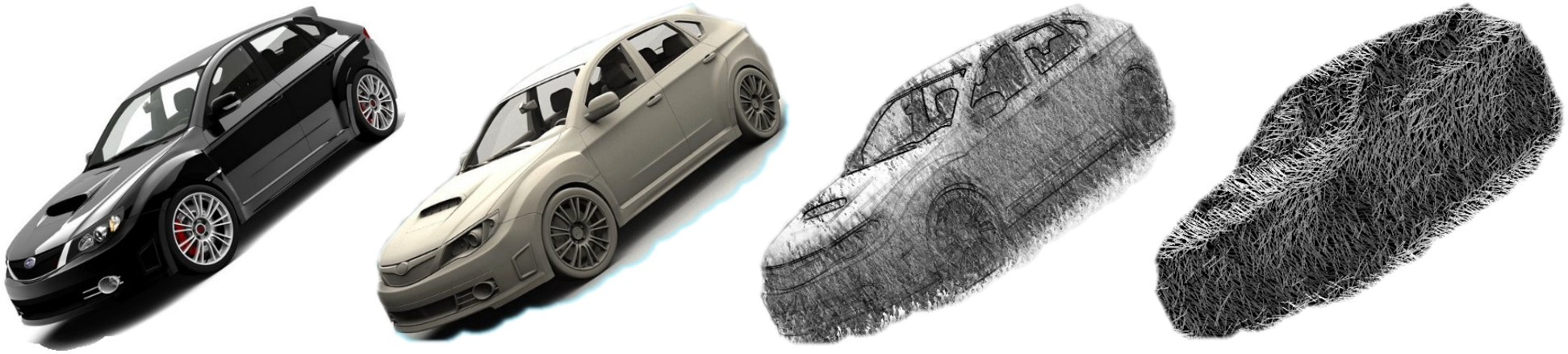
Abstrakcja

Abstrakcja — w sztuce to *abstrakcjonizm*..., po co w programowaniu?

Czy
muszę wiedzieć wszystko?

Czy potrzebne są wszystkie
szczegóły?

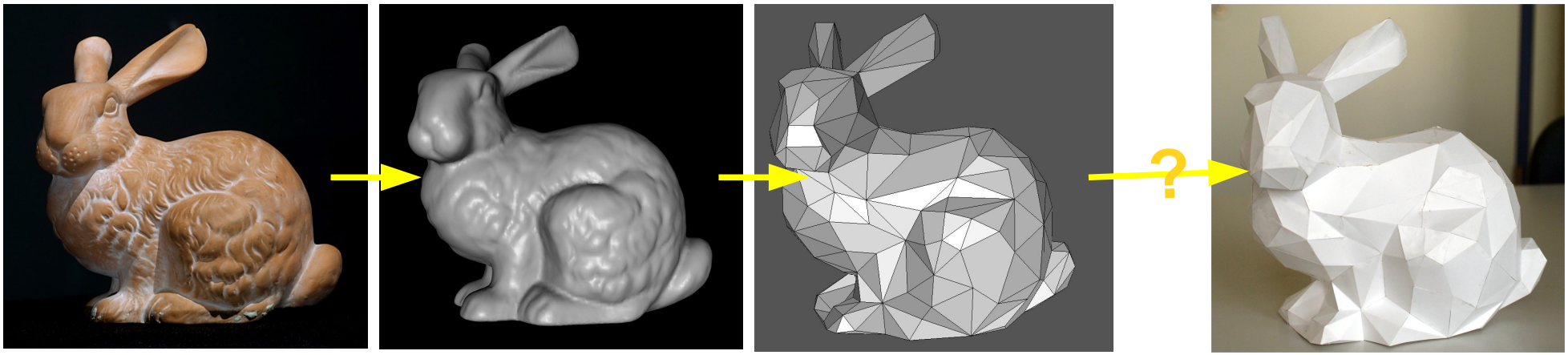




Rysunki: http://www.turbosquid.com

Abstrakcja — tworzenie komputerowych modeli

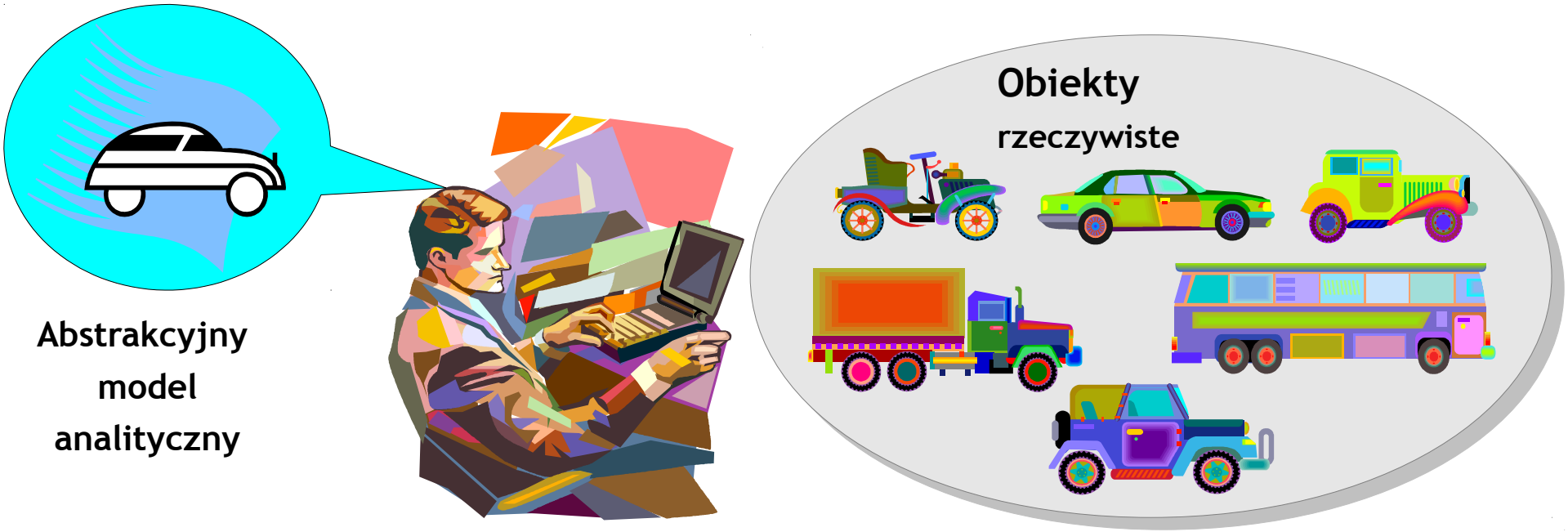
- Obiekt jest zwykle *programową abstrakcją* pewnego konkretnego bytu ze świata rzeczywistego, reprezentującego *rzecz* (obiekt fizyczny) lub *pojęcie* (obiekt konceptualny).
- W programowaniu obiekt stanowi zatem *komputerowy model* jakiegoś obiektu ze świta rzeczywistego.
- Obiekt-model zwykle jest *uproszczony* w stosunku do rzeczywistego pierwowzoru. Jest tak, ponieważ trudno *wiernie odwzorować rzeczywistość*, lub to wierne odwzorowanie *nie jest wcale potrzebne*.



The "Stanford Bunny"

Abstrakcja — tylko to co istotne

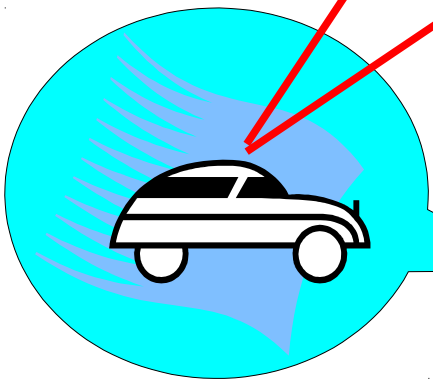
- *Abstrakcja* pozwala postrzegać modelowaną rzeczywistość *bez rozważania wszystkich szczegółów* tej rzeczywistości oraz *bez wnikania w jej wewnętrzną strukturę*.
- Stosowanie *abstrakcji* w programowaniu obiektowym prowadzi do tego, że obiekt powiniem posiadać tylko te elementy, które są *istotne dla tworzonego systemu*, elementy w danym momencie nieistotne *powinny zostać pominięte*.



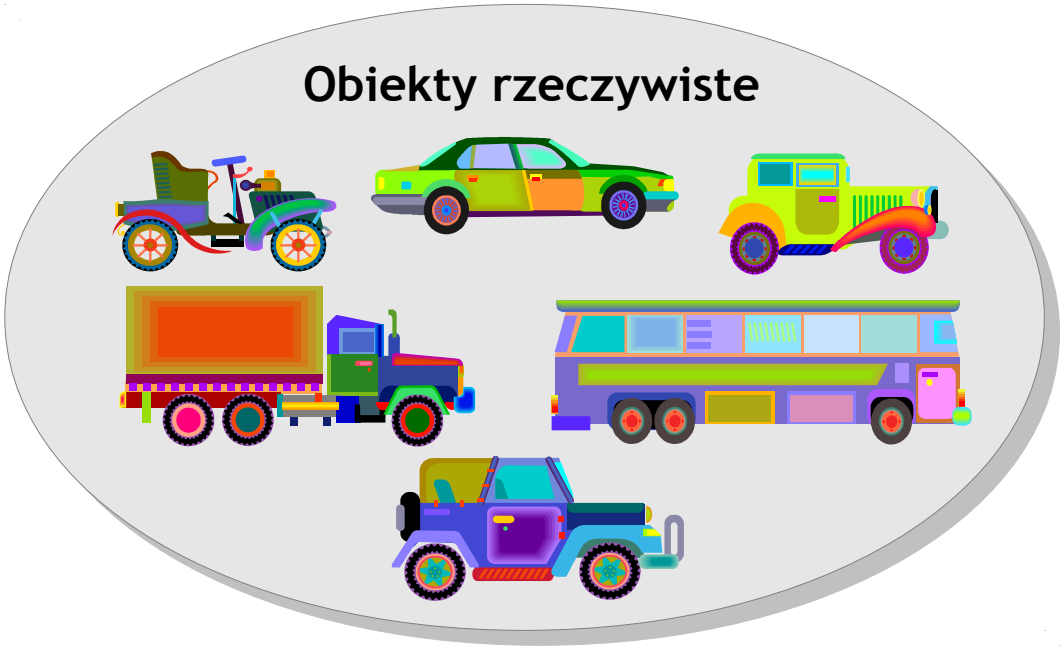
Abstrakcja — obiekt w rzeczywistości → model → obiekt w programie

Cechy	Marka Model Rok produkcji Nr rejestracyjny Kolor nadwozia Nr silnika . . .
Akcje	Podaj średnie spalanie Podaj przebieg . . .

Reprezentacja obiektowa



Abstrakcyjny model analityczny



Abstrakcja — *prostota* ale nie *prostactwo*

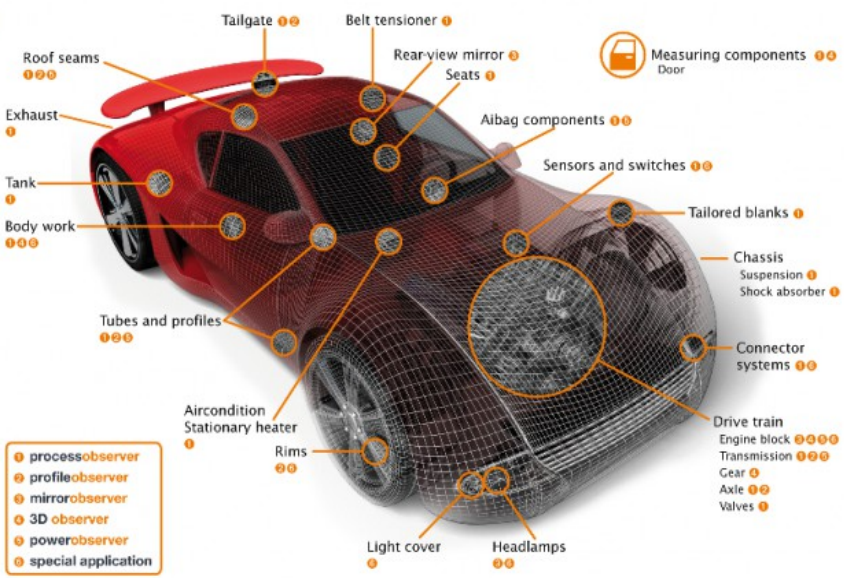
- Abstrakcja ma dostarczyć maksymalnie *prostego* i *adekwatnego* dla problemu *opisu obiektu*.
- Obiekt ma być *prosty*, lecz nie *prostacki*.
- Stosowaniu abstrakcji w programowaniu ma towarzyszyć *antycypacja*, czyli *przewidywanie i wyprzedzenie* potencjalnych zmian *wymagań*.
- *Antycypacja* i *abstrakacja* mają dostarczać *prostego* opisu obiektu, jednocześnie *elastycznego* i *podatnego na rozwój*.

Hermetyzacja

Hermetyzacja — czyli: *mam to w ...*, po co w programowaniu?

Czy muszę
wiedzieć co jest w środku
aby tego używać?

Czy muszę wiedzieć jak to działa
aby tego używać?



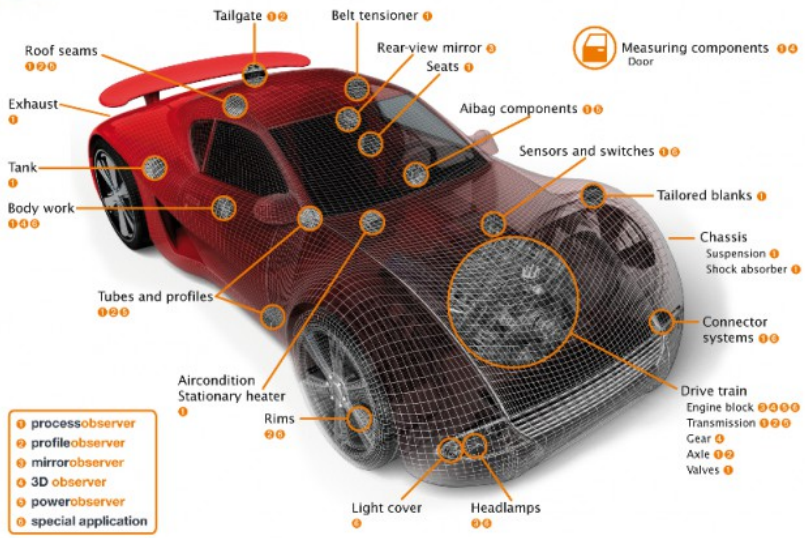
Hermetyzacja — totalny zez, czyli: *spójrz z dwóch stron…*

Nie wiem jak to działa,
i nie chcę wiedzieć.

Chcę tego używać!

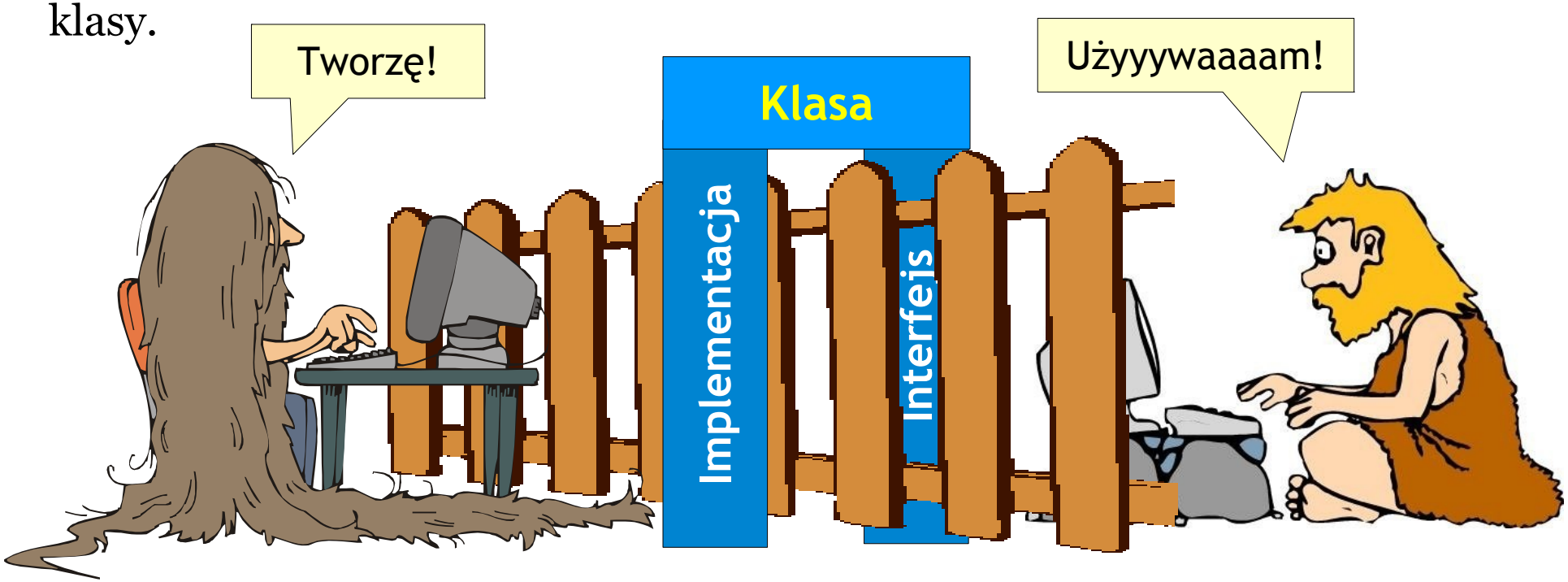
Wiem jak to działa,
ale Ty nie musisz tego wiedzieć
aby używać.

Tak, dbam o swoje interesy!



Hermetyzacja w programowaniu

- Bardzo często programista *wykorzystujący* obiekt pewnej klasy *nie pisze* jej samodzielnie, wykorzystując klasy przygotowane przez innych.
- Klasa powinna mieć jasno zdefiniowany *interfejs*, który wykorzystuje programista używający klasy.
- Szczegóły reprezentacji pól oraz implementacji metod powinny zostać przez twórcę klasy *oddzielone od interfejsu* i *niewidoczne* dla programisty używającego klasy.



Hermetyzacja

Klasa bez hermetyzacji

```
class Runner
{
  public:
    int  x, y;
    char shape;

    void show();
    void hide();

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

- Hermetyzację osiąga się to poprzez ukrycie (ang. *information hiding*) pól obiektu oraz niektórych funkcji składowych w sekcjach *prywatnych* lub *chronionych*.

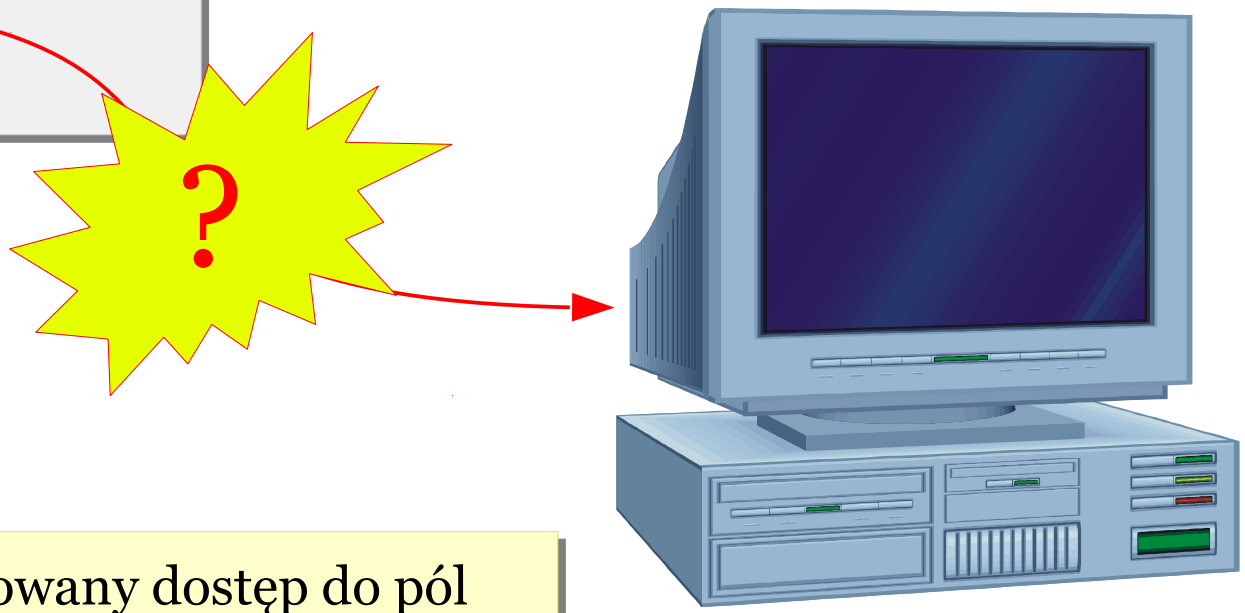
- Dzięki hermetyzacji można modyfikować wewnętrzną reprezentację klasy, sposób jej działania w sposób przeźroczysty dla programisty używającego klasy.

Niebezpieczeństwa braku hermetyzacji

```
Runner R;

R.x = -400;
R.y = 1200;

R.shape = ' ';

. . .

R.show();
```

Błędna pozycja ekranowa

Nieszczęśliwie dobrany znak reprezentujący obiekt



Bezpośredni i niekontrolowany dostęp do pól obiektu nie pozwala na sprawdzenie poprawności informacji przekazywanych obiektowi.

Ukrycie pól — dostęp za pomocą funkcji składowych

```
Runner R;

R.setX( -400 );
R.setY( 1200 );

R.setShape( ' ' );
. . .

R.show();
```

Ustalanie pozycji via funkcje składowe obiektu

Ustalanie znaku via funkcje składowe obiektu

Funkcje składowe zignorowały niepoprawne parametry i ustaliły wartości domyślne:
x = 1, y = 1, shape = '*'

Funkcje składowe mogą kontrolować poprawność danych przekazywanych obiektowi, odrzucając wartości niepoprawne, przyjmując ustalone arbitralnie wartości domyślne dla pól.

Publiczne i prywatne elementy klasy

```
class Runner
{
  public :
    void setX( int newX );
    void setY( int newY );
    void setShape( char newShape );

    int  getX();
    int  getY();
    char getShape();

    void show();
    void hide();

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

  private:
    int  x, y;
    char shape;

    bool isXOnScreen( int newX );
    bool isYOnScreen( int newY );
};
```

```
Runner R;

R.x = 10;
R.y = 20;
R.shape = '#';
R.isXOnScreen( 100 );
```

Interfejs klasy — sekcja publiczna

Tylko te elementy mogą być wykorzystywane przez programistę używającego klasę Runner.

```
R.setX( 10 );
R.setY( 20 );
R.setShape('#');
```

Wewnętrzne elementy klasy — sekcja prywatna

Te elementy mogą być wykorzystywane wyłącznie przez funkcje składowe klasy.

Podział publicznych funkcji składowych

```
class Runner
{
  public :
    void setX( int newX );
    void setY( int newY );
    void setShape( char newShape );

    int  getX();
    int  getY();
    char getShape();

    void show();
    void hide();

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

  private:
    int  x, y;
    char shape;

    bool isXOnScreen( int newX );
    bool isYOnScreen( int newY );
};
```

Modyfikatory — funkcje dokonujące modyfikacji wartości pól, zapewne w sposób kontrolowany.

Akcesory — funkcje umożliwiające kontrolowane pobieranie wartości pól.

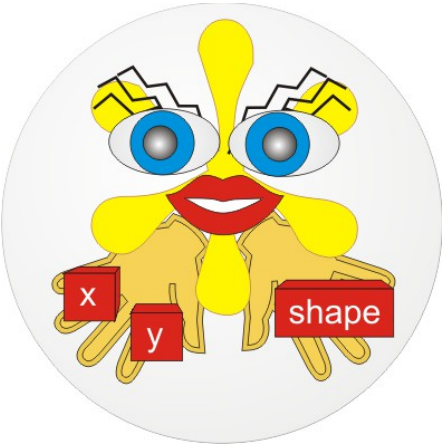
Realizatory — funkcje realizujące właściwe dla klasy usługi.

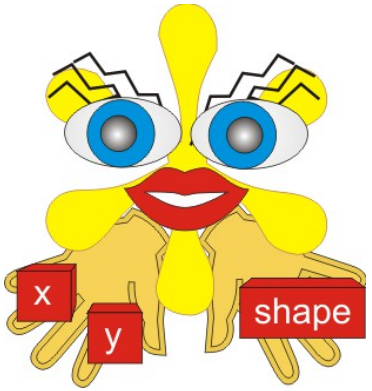
Modyfikatory i *akcesory* mają charakter administracyjno-kontrolny, *realizatory* to esencja działania klasy.

Z hermetyzacją i bez, czyli jak z polem siłowym i bez niego

```
class Runner
{
  public :
    void setX( int newX );
    void setY( int newY );
    void setShape( char newShape );

    int  getX();
    int  getY();
    char getShape();

    void show();
    void hide();

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

  private:
    int  x, y;
    char shape;

    bool isXOnScreen( int newX );
    bool isYOnScreen( int newY );
};
```

```
class Runner
{
  public:
    int x, y, shape;

    void show();
    void hide();

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





Dziedziczenie

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.

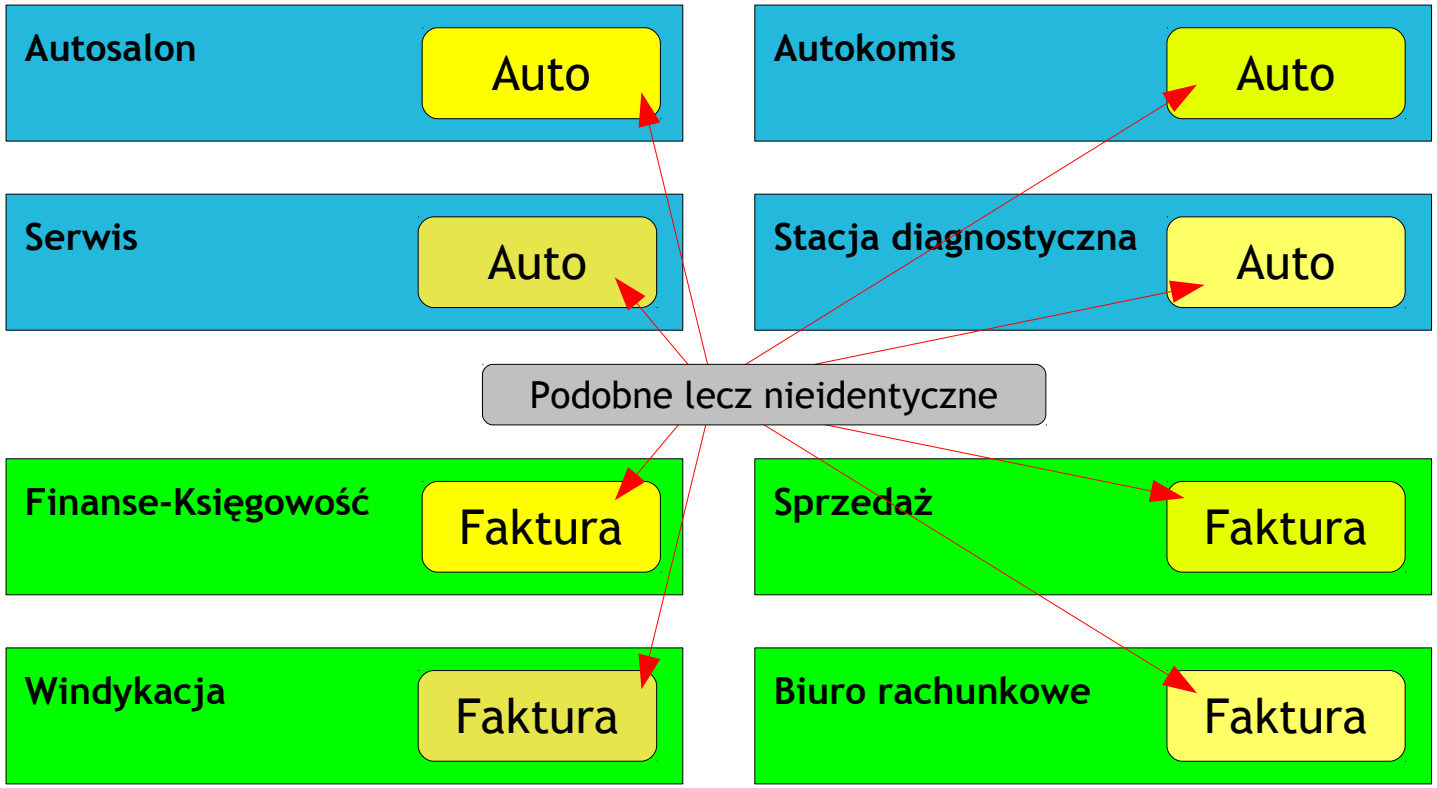
Ponowne wykorzystanie kodu w programowaniu obiektowym

- 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*).

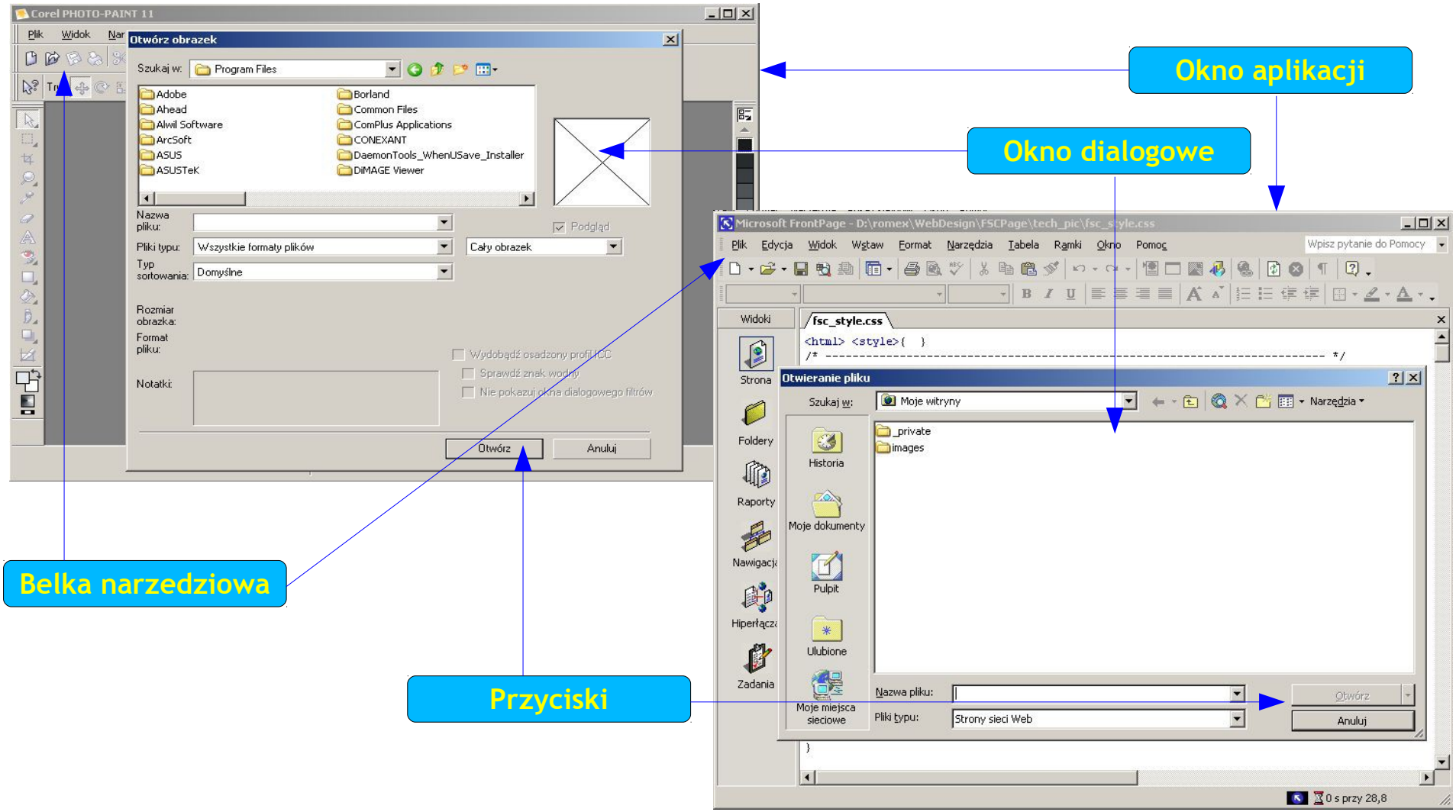
Podobne ale nie identyczne

- 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*.



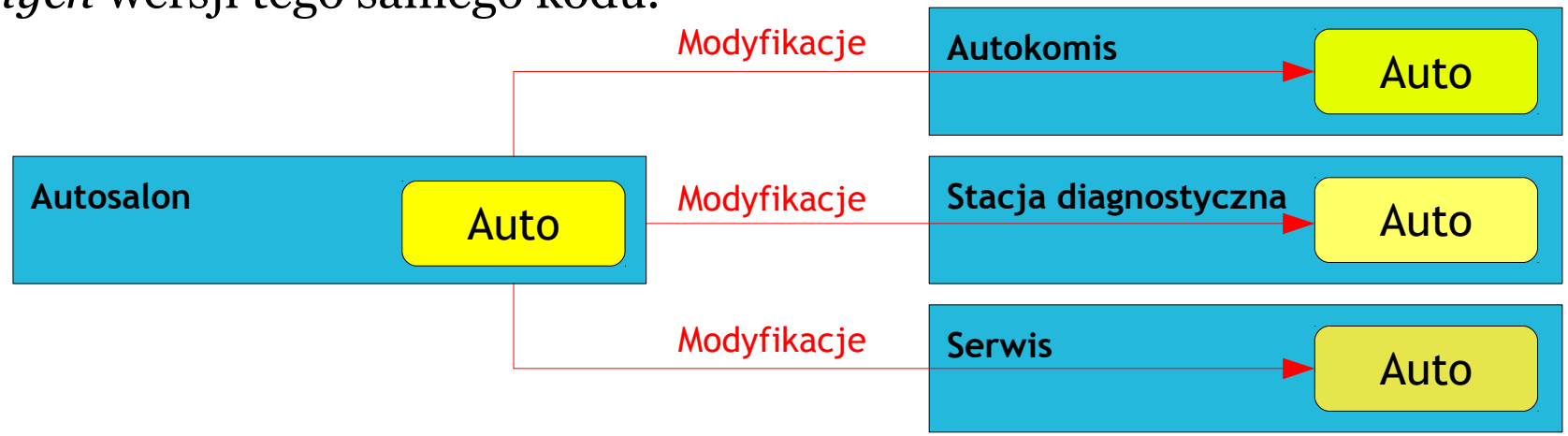
Podobne ale nie identyczne

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



Ponowne wykorzystanie kodu — problemy

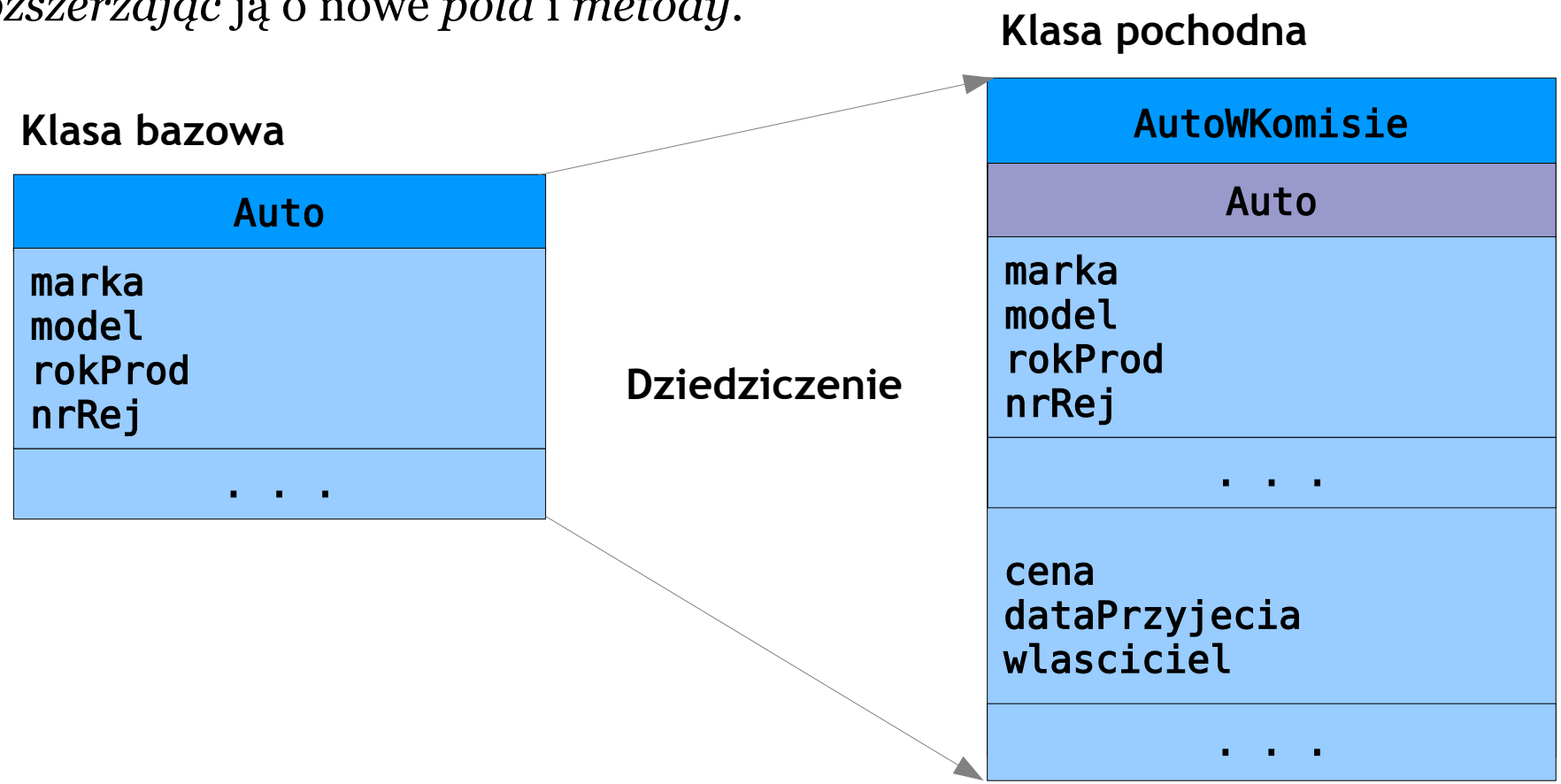
- Powtarzalność nieidentycznych obiektów powoduje to, że musimy *modyfikować istniejący kod*, dostosowując go do *specyfiki projektu*.
- Modyfikacje potrzebne z punktu widzenia jednego systemu mogą być *nieużyteczne* lub *nieakceptowalne* z punktu widzenia innego systemu.
- Modyfikacje dostosowujące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

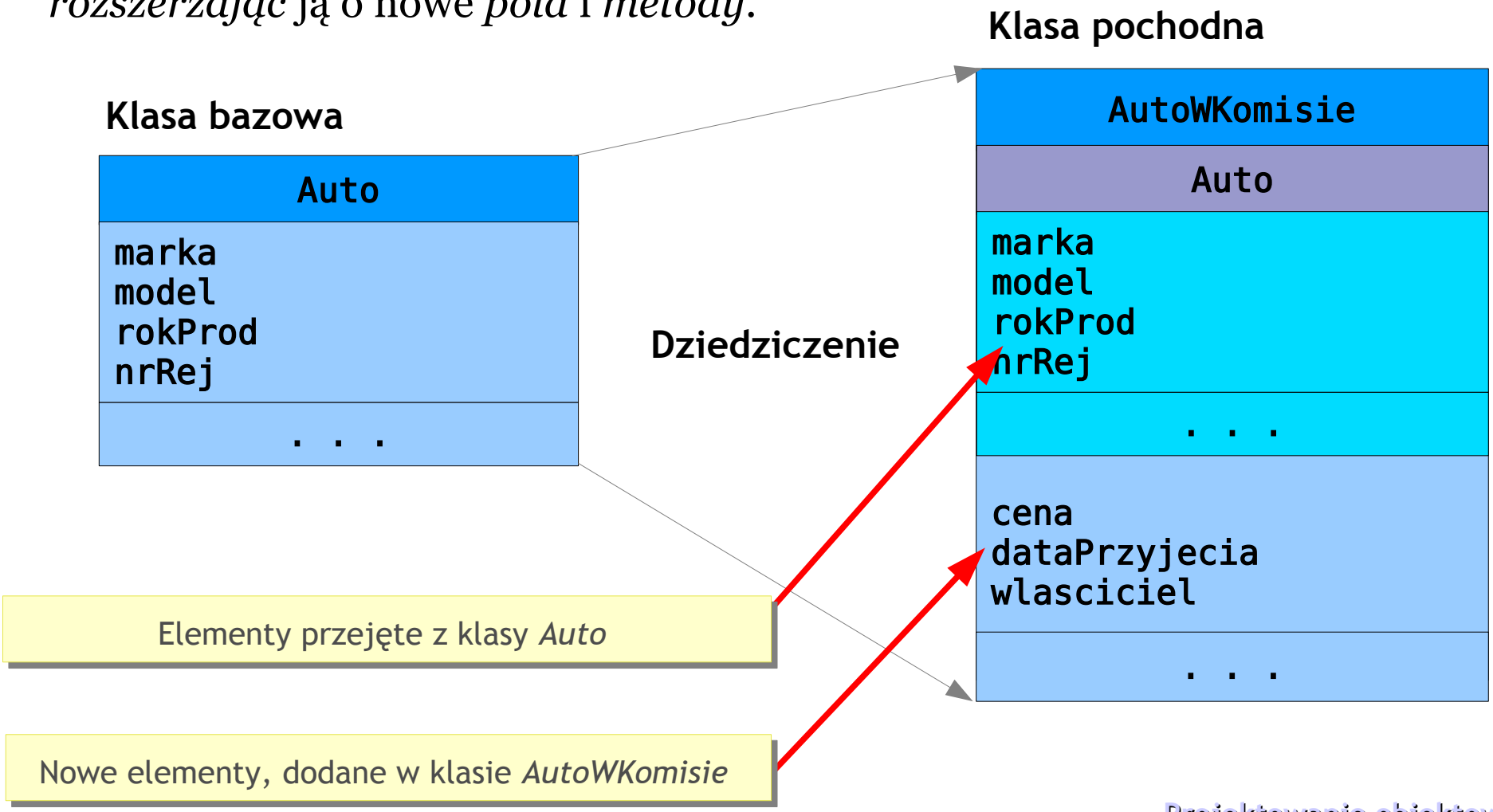
- ***Dziedziczenie*** polega na budowaniu *nowych klas*, zwanych klasami *potomnymi*, na podstawie klas już istniejących, zwanych klasami *bazowymi*.
- Każda klasa *pochodna* ***dziedziczy*** wszystkie właściwości klasy bazowej, być może *rozszerzając* ją o nowe *pola* i *metody*.



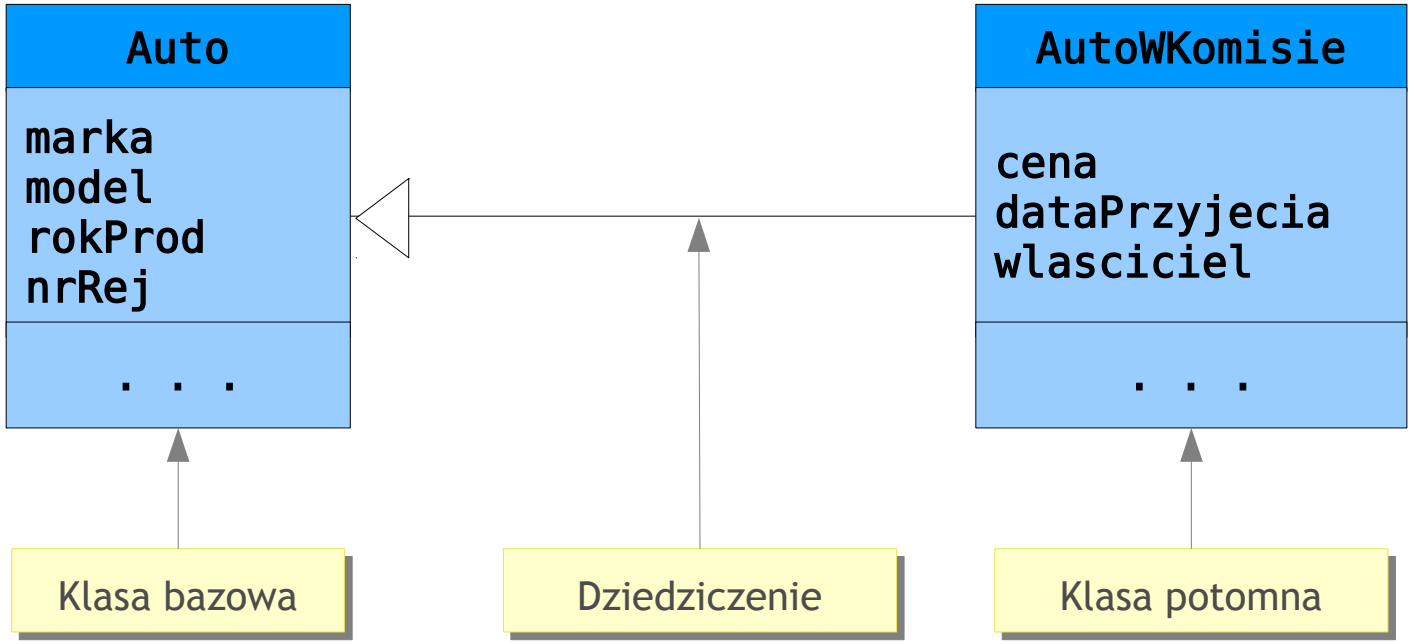
Uwaga — chwilowo na przedstawianych diagramach klas nie będą stosowane oznaczenia widoczności pól: (+) dla pól publicznych i (-) dla pól prywatnych..

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*.

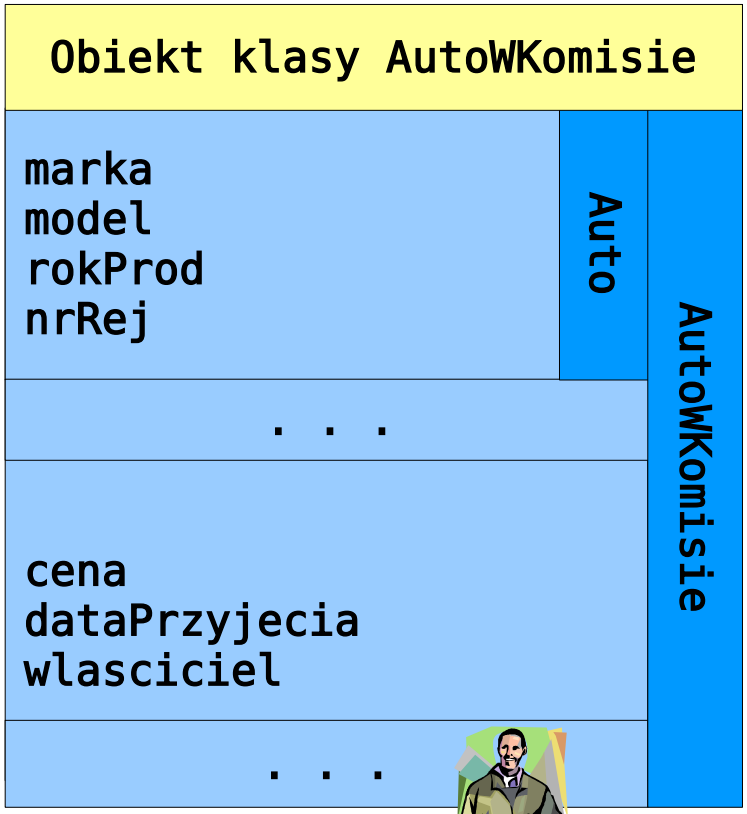


Klasa Auto jako klasa bazowa klasy AutoWKomisie — notacja UML



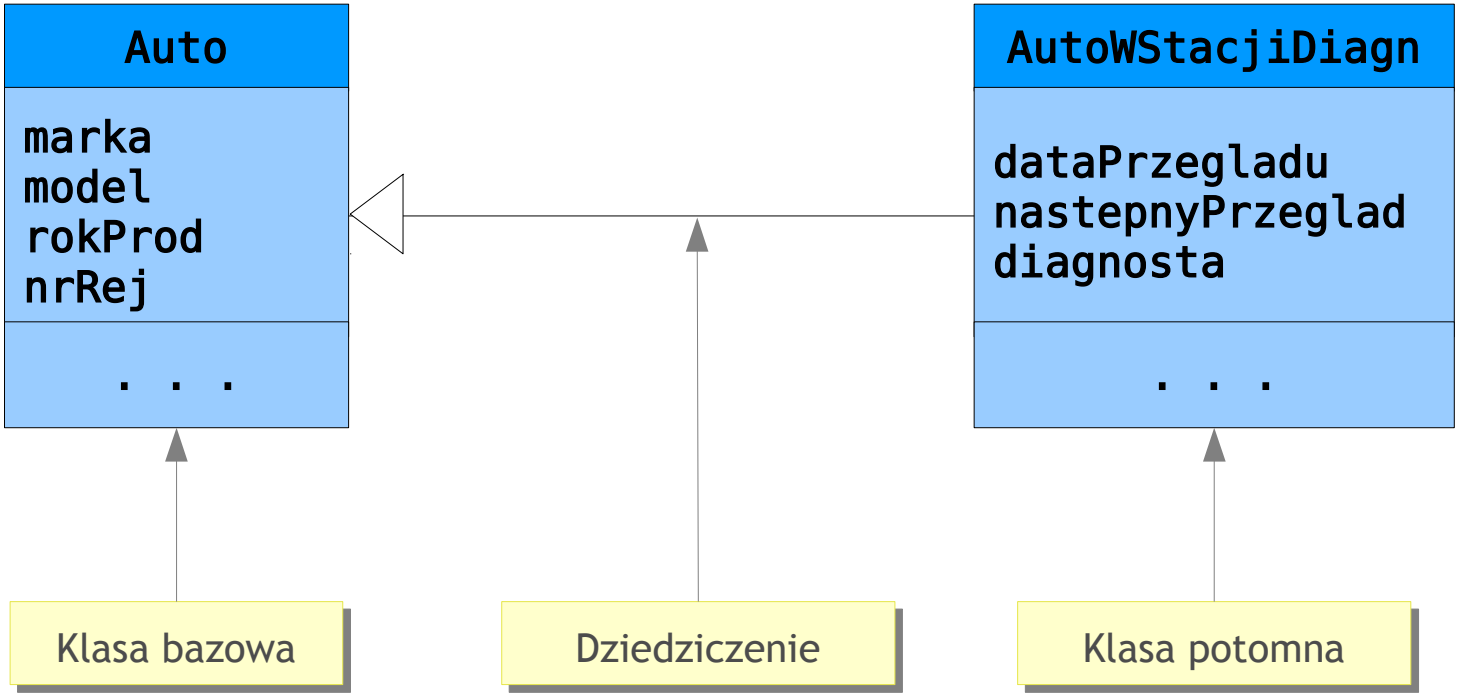
Obiekt klasy pochodnej otrzymuje wszystkie pola i metody klasy bazowej.
100% elementów klasy bazowej występuje w klasie pochodnej.

Obiekt klasy pochodnej a obiekt klasy bazowej

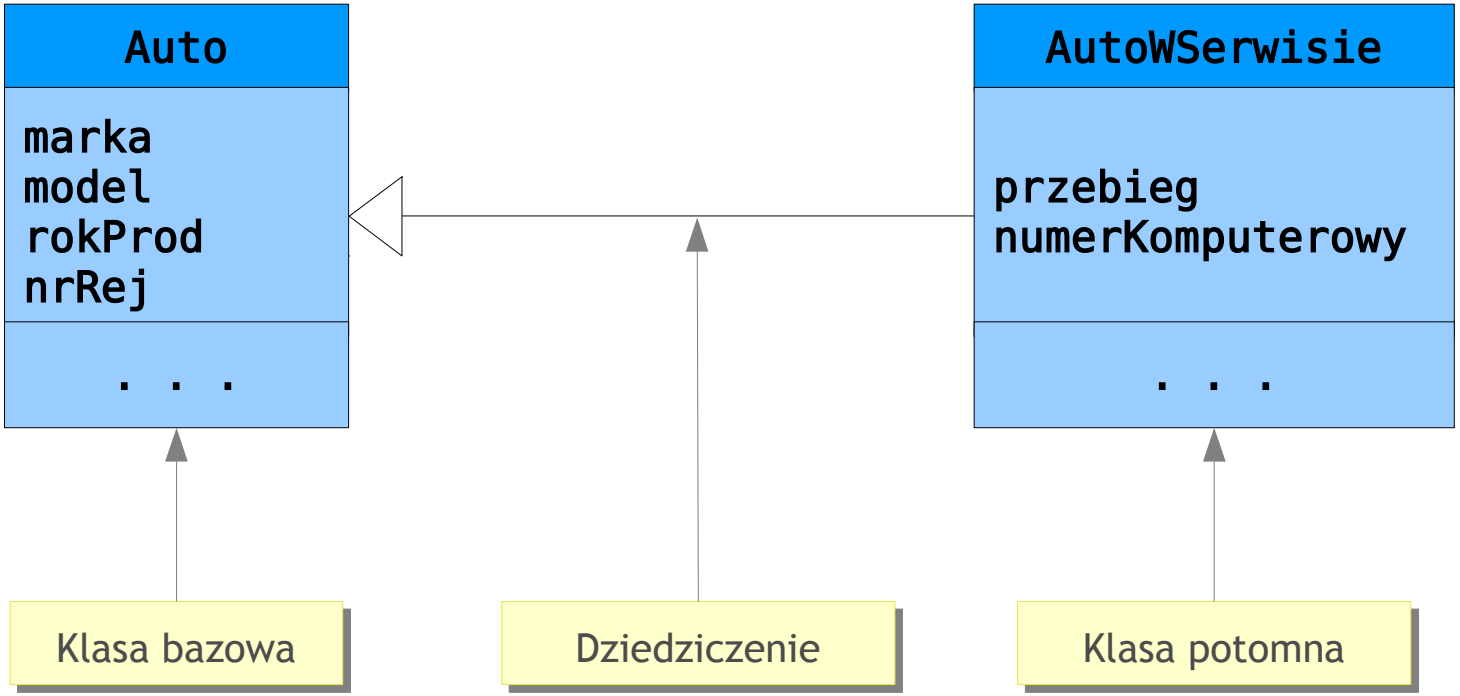


Dziedziczenie pozwala na rozszerzanie klasy o *nowe informacje* i/lub *nowe zachowanie*. Można również *zmieniać* dotychczasowe *zachowanie*.

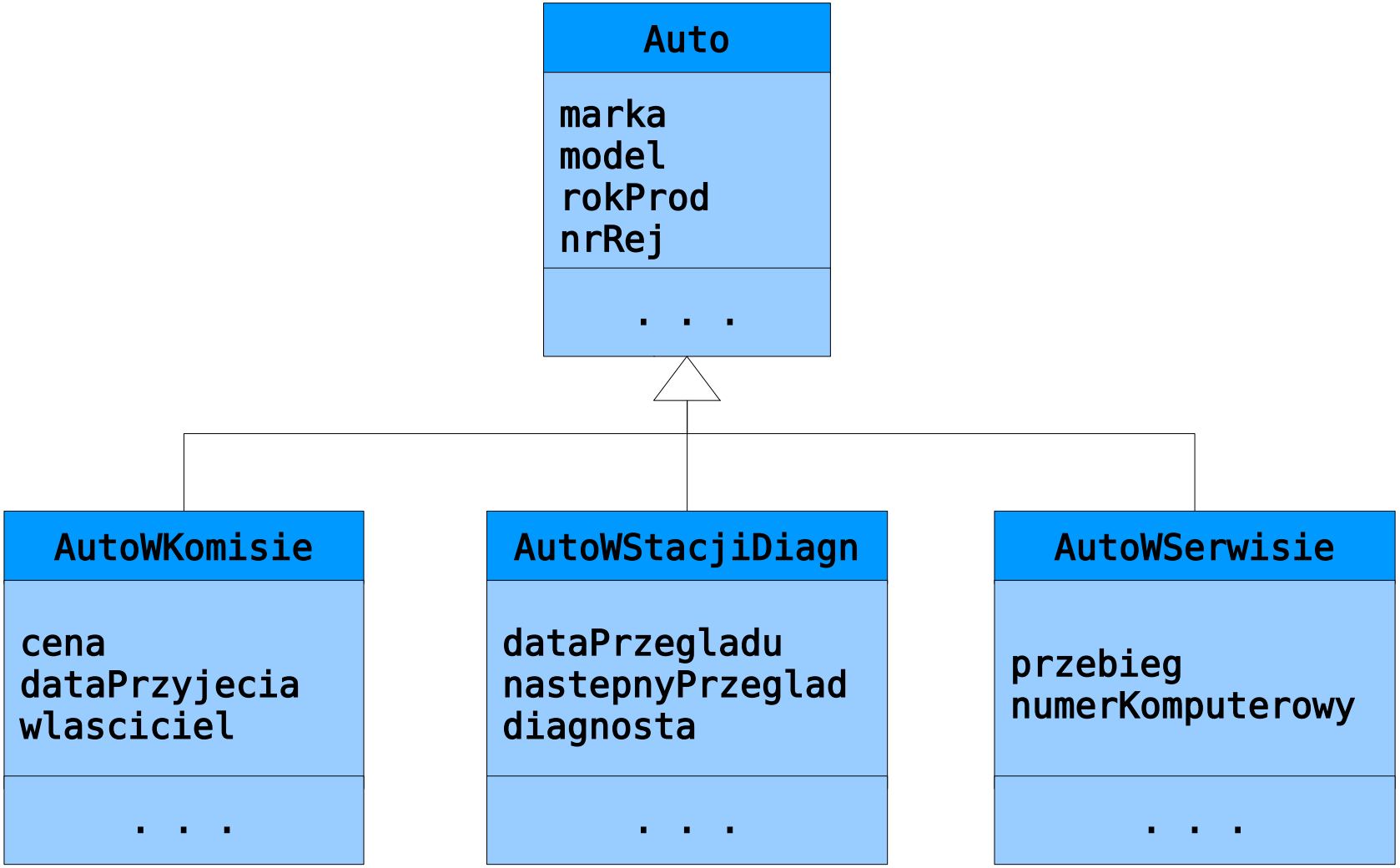
Klasa Auto jako klasa bazowa klasy AutoWStacjiDiagn



Klasa Auto jako klasa bazowa klasy AutoWSerwisie

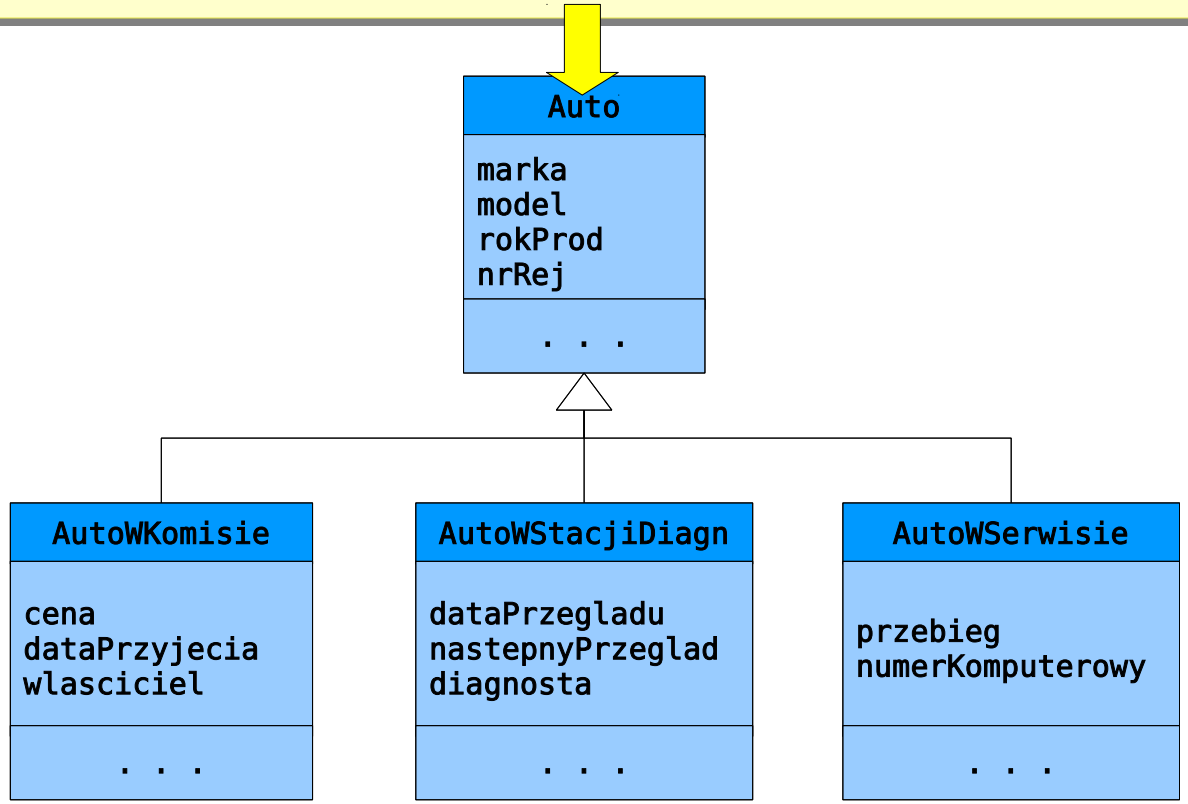


Klasa bazowa i klasy potomne tworzą hierarchię klas

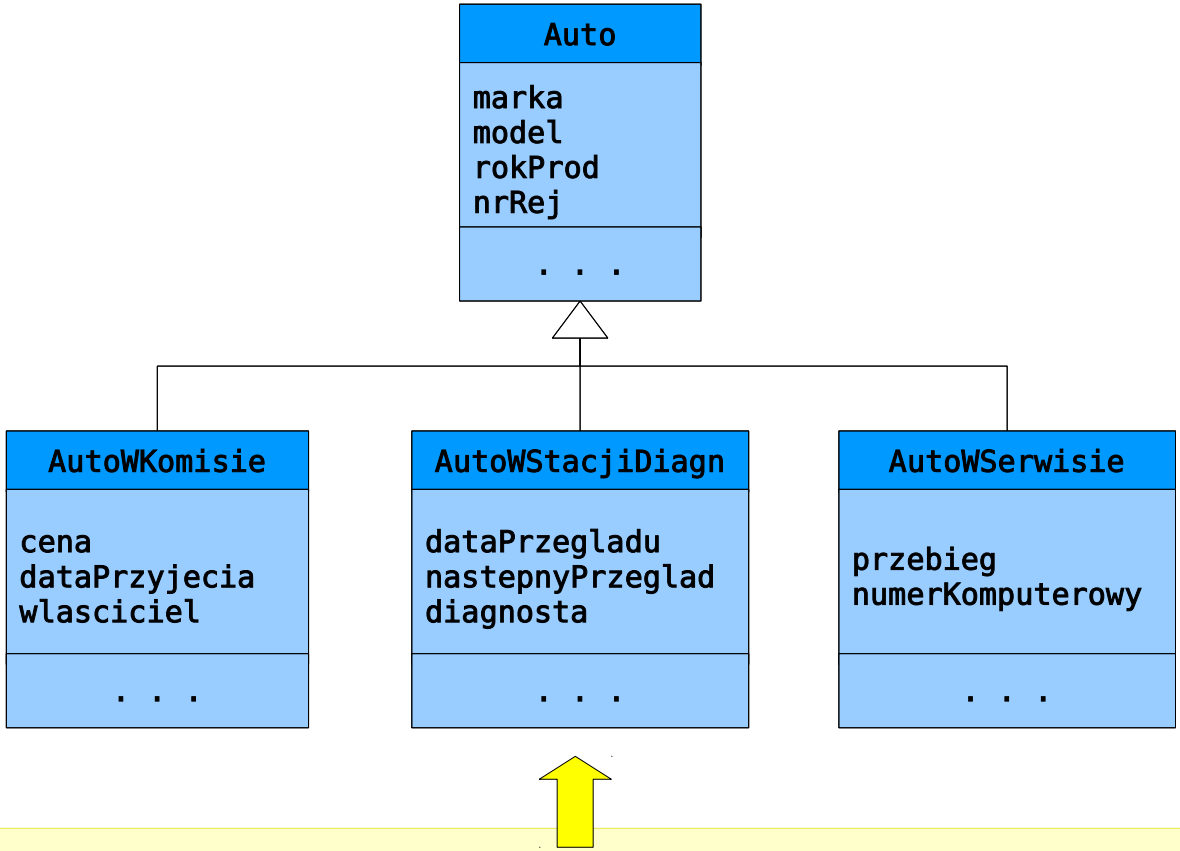


Klasa bazowa jako klasa ogólna

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



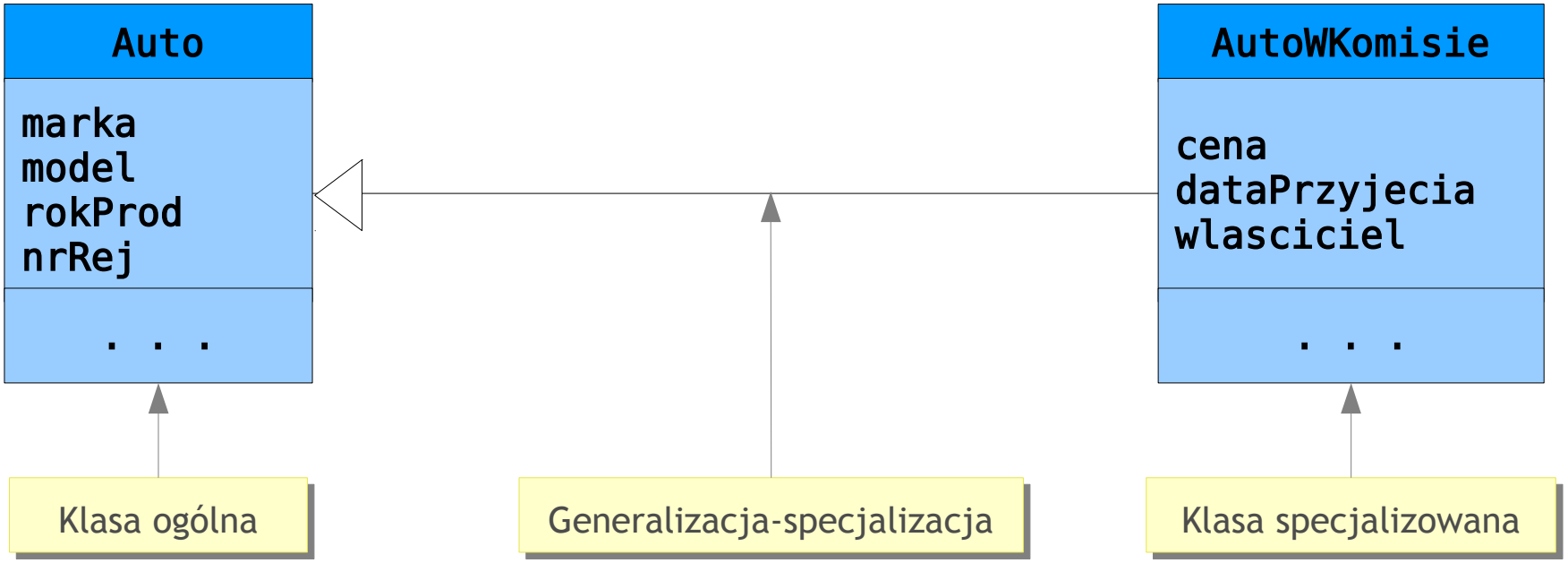
Klasa bazowa jako klasa ogólna



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?

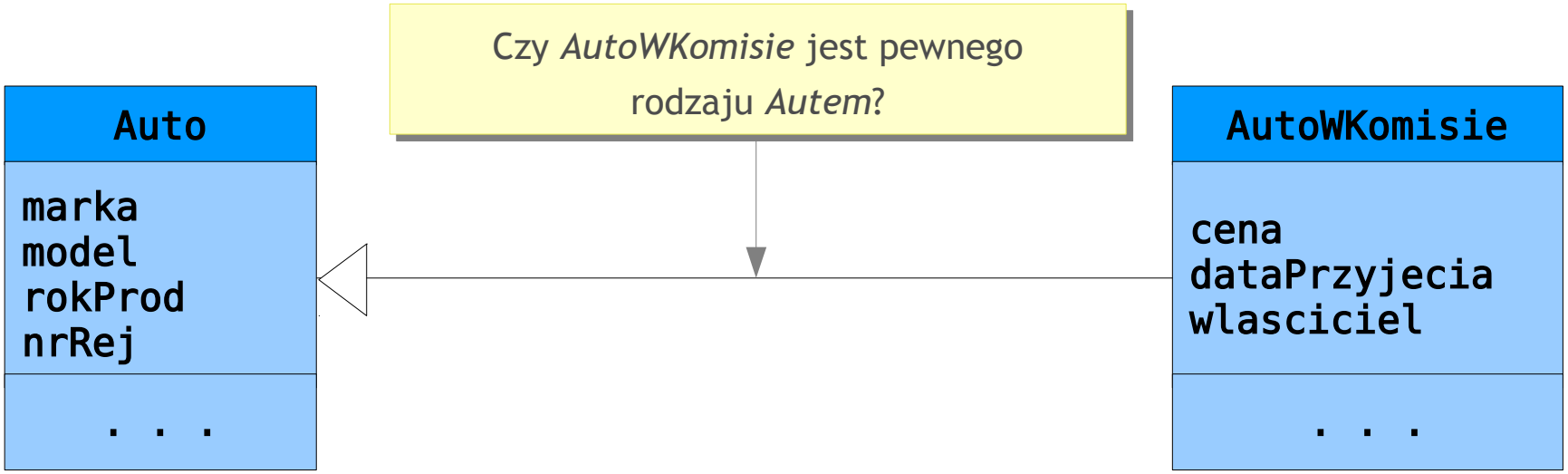
Reguła ***is-a*** (ang. *is a kind of*) pozwala na sprawdzenie czy zachodzi związek dziedziczenia (specjalizacji–generalizacji) pomiędzy klasami.

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*).

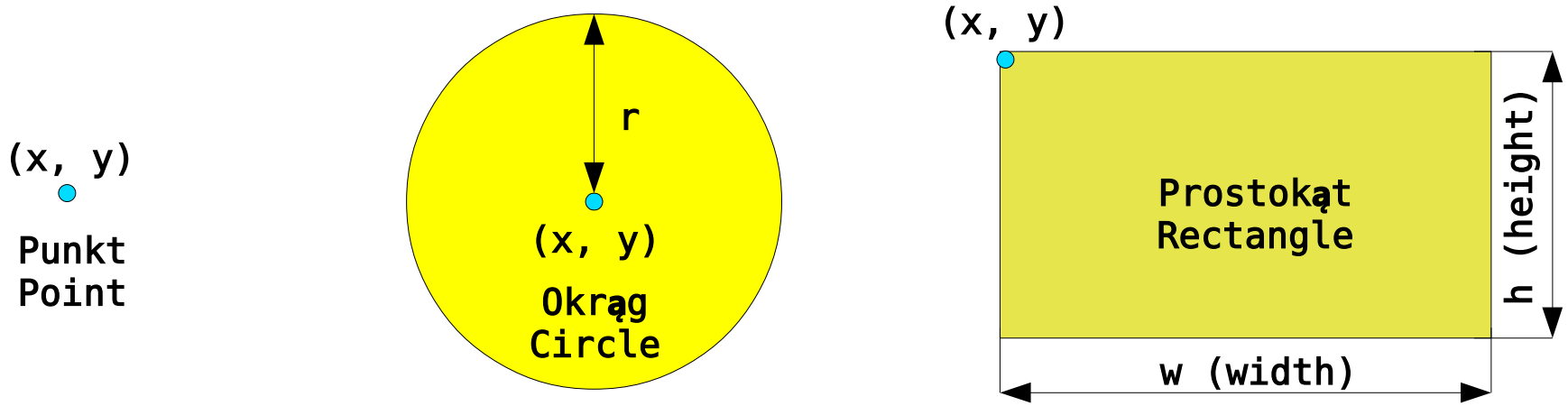
Przykład weryfikacji poprawności dziedziczenia

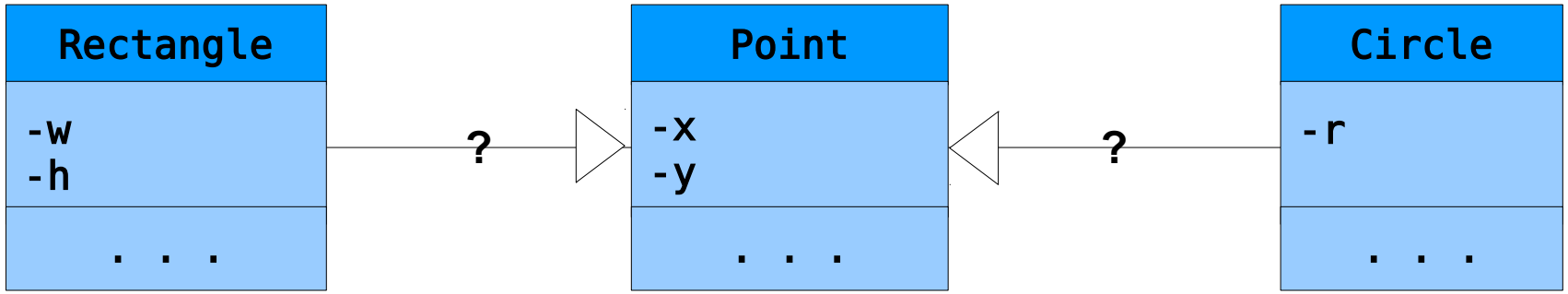
Czy klasa **`AutoWKomisie`** jest klasą pochodną klasy **`Auto`**:

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

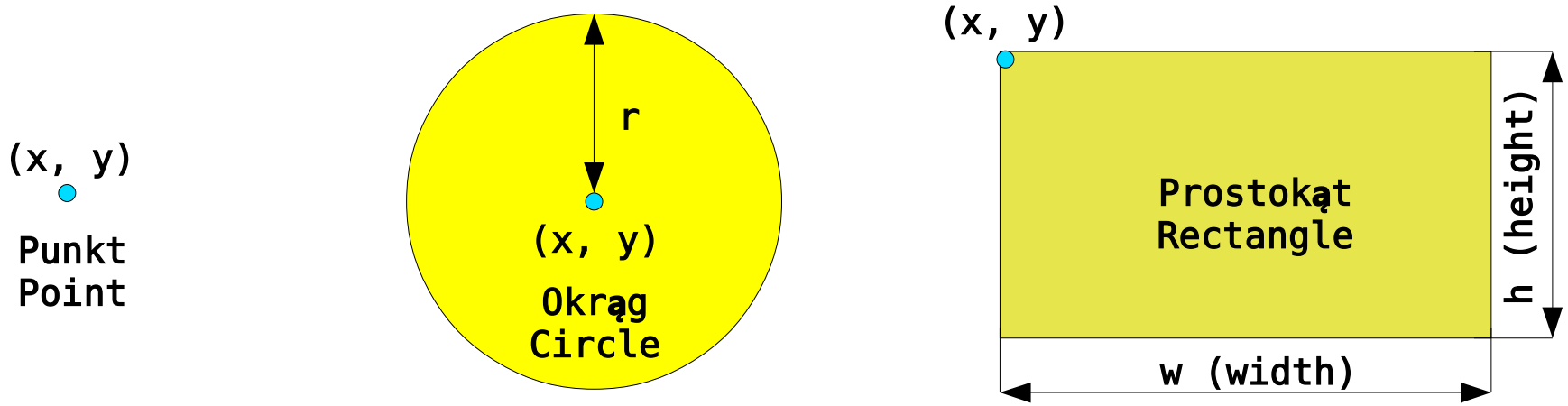
A jak ten test przejdą standardowe przykłady dziedziczenia z popularnych książek o programowaniu obiektowym?

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`**:

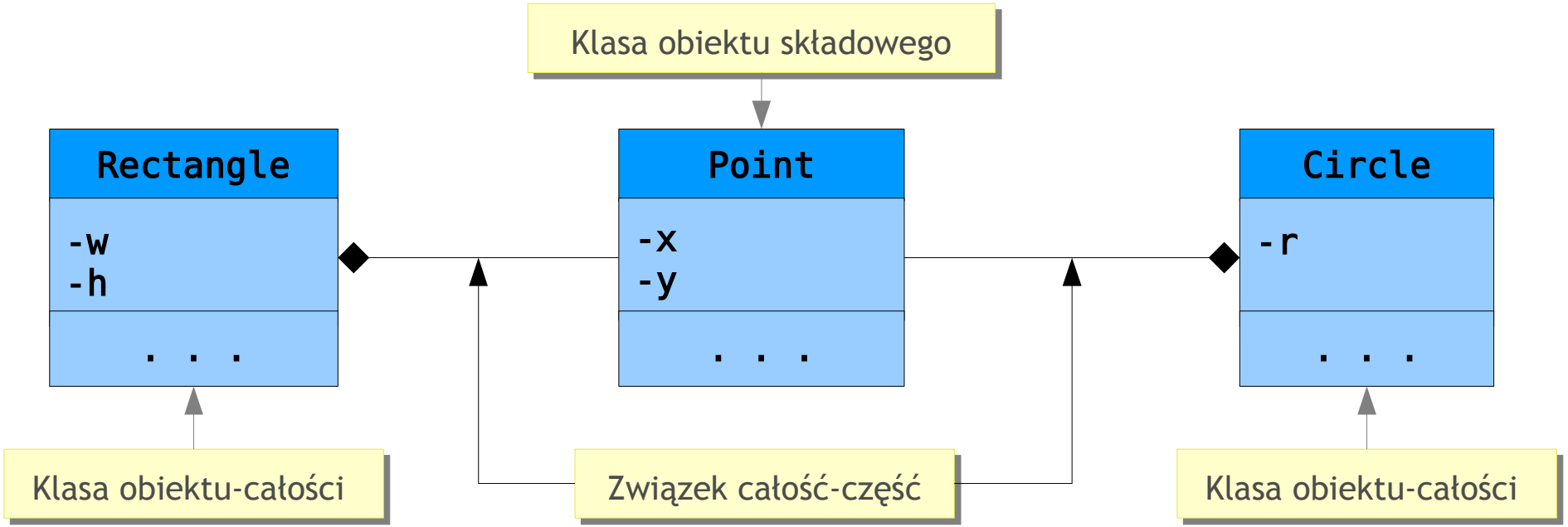
- czy obiekt klasy **`Rectangle`** jest pewnego rodzaju obiektem klasy **`Point`**? ← **?**
- czy obiekt klasy **`Point`** jest pewnego rodzaju obiektem klasy **`Rectangle`**? ← **?**

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:

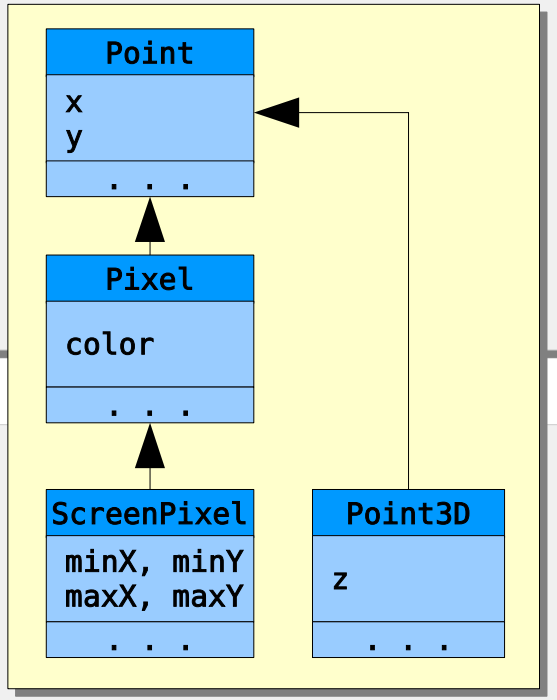
- czy obiekt klasy **A** jest częścią obiektu klasy **B**?
- czy obiekt klasy **B** zawiera w sobie obiekt klasy **A**?



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

Nowość nr 2 — redefinicja metody w klasie pochodnej

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



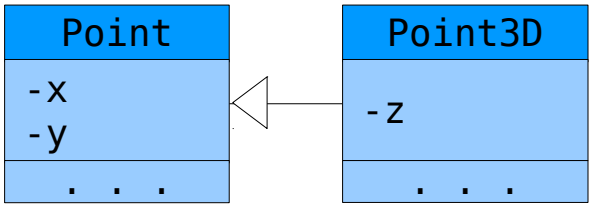

```
class ScreenPixel : public Pixel
{
  public :
    . . .
    void setX( int newX )
    {
      Point::setX( ( newX >= minX && newX <= maxX ) ? newX : 0 );
    }
    . . .
};
```

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

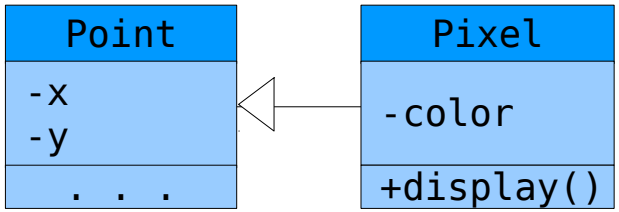
Dziedziczenie
Krótkie podsumowanie

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*.



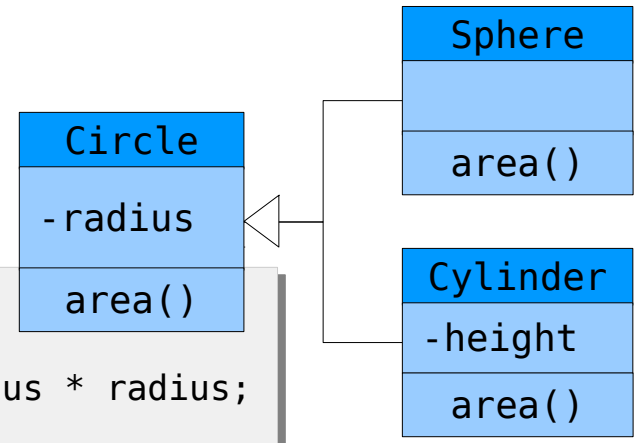
- Klasa pochodna zazwyczaj posiada *potrafi więcej niż klasa bazowa* — umiejętności te są wynikiem zdefiniowania *nowych funkcji składowych klasy pochodnej*.



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

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.

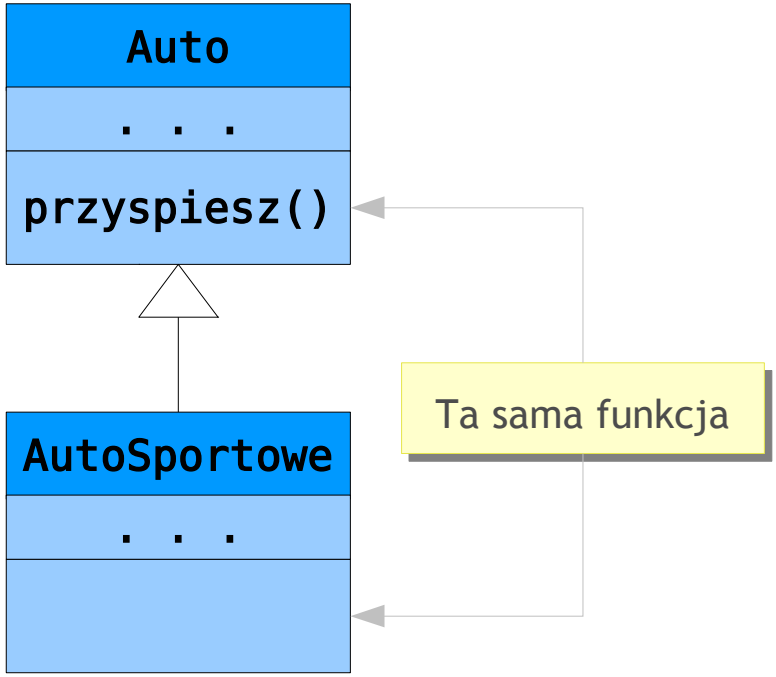


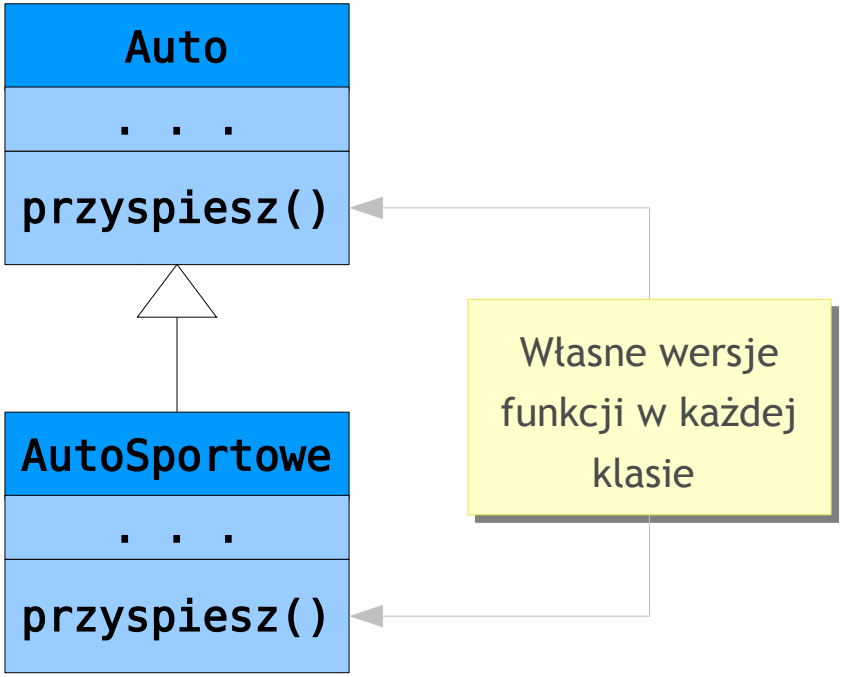

```
Sphere::area()
{
  return 4 * Circle::area();
}
```

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

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

Redefiniowanie funkcji składowych a diagramy UML



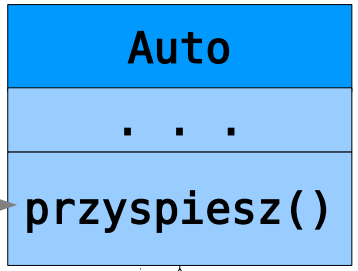


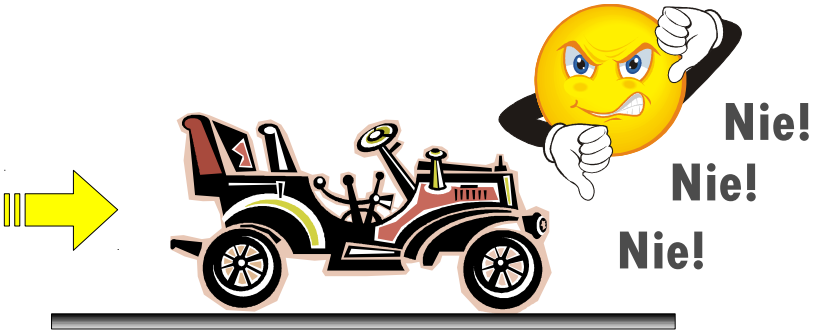
Klasa AutoSportowe wykorzystuje odziedziczoną funkcję *przyspiesz* klasy Auto.

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

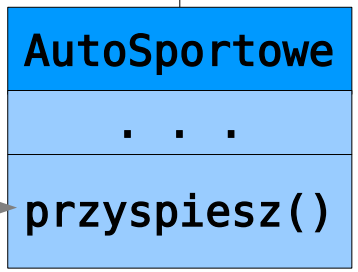
Co daje redefiniowanie funkcji składowych?

```
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.

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.

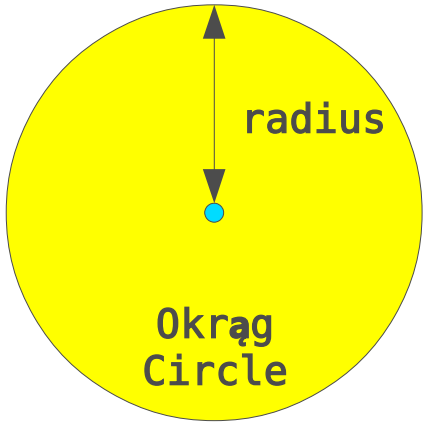
Za i przeciw dziedziczeniu wielobazowemu:

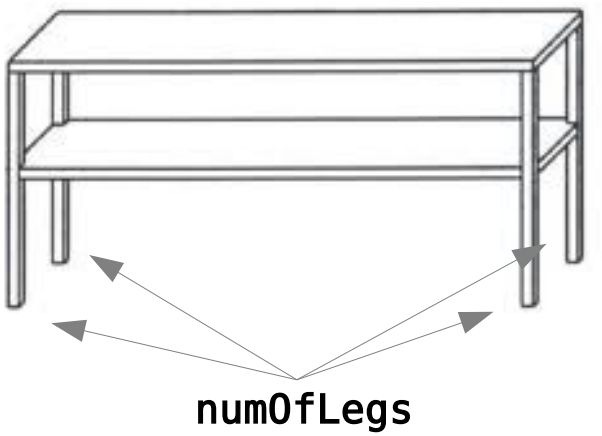
- *Tak* — pozwala programiście budować elastyczne i dobrze odwzorowujące rzeczywistość hierarchię klas.
- *Nie* — zmusza programistę do większej uwagi, zwiększa złożoność języka oraz kompilatora.
- W wielu językach programowania zrezygnowano z dziedziczenia wielobazowego:
  - jest np. w językach *C++*,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

Dziedziczenie wielobazowe — przykład

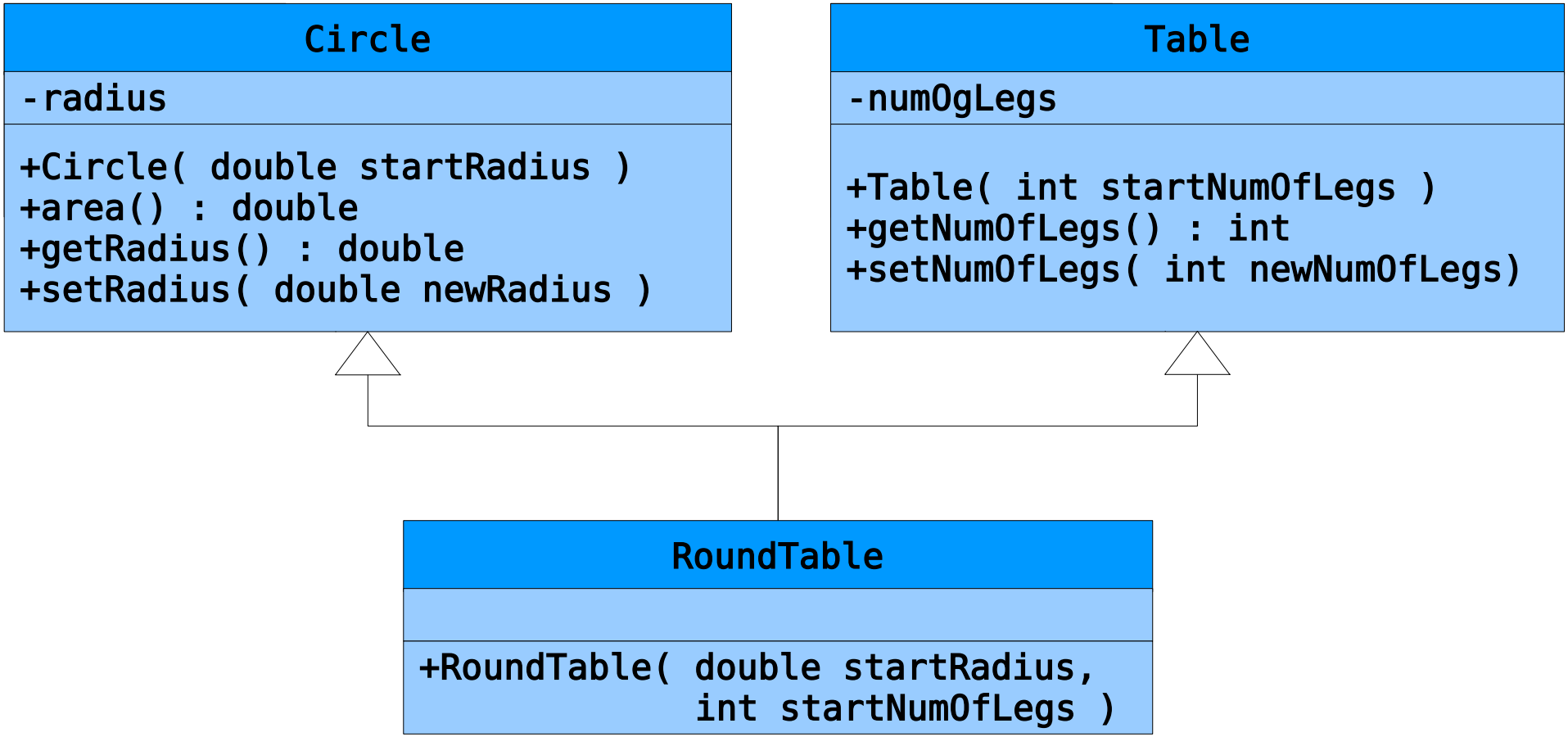
Circle
-radius
+Circle(double startRadius) +area() : double +getRadius() : double +setRadius(double newRadius)

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
```

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

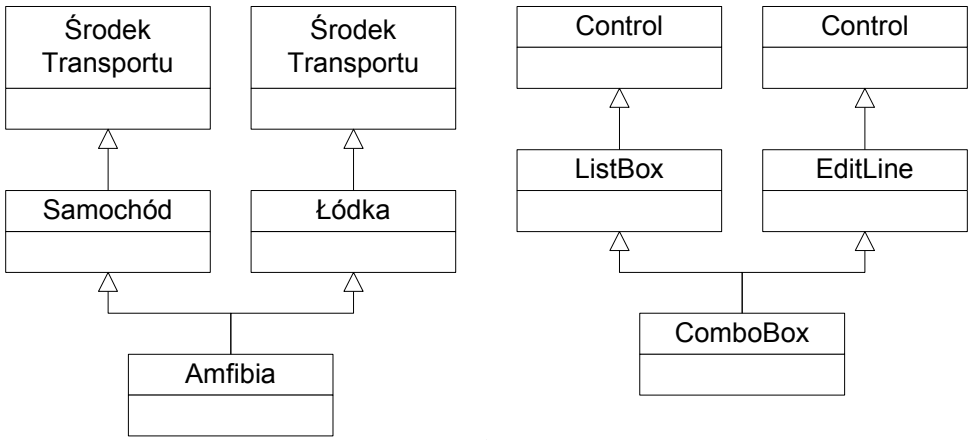
```
class Amphibian : public Vehicle, public Boat
{
  public:
    Amphibian( int maxLandSpeed = 0, int maxWaterSpeed = 0,
               int numOfWheels = 0, int displacement = 0 )
    : Vehicle( maxLandSpeed, numOfWheels ),
      Boat( maxWaterSpeed, displacement )
    {
    }

    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*.

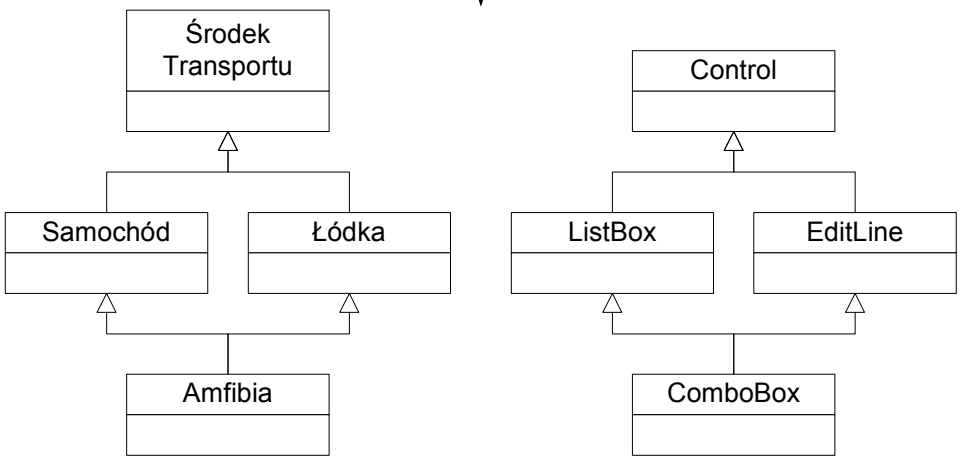
Omówienie tych zagadnień *wykracza poza ramy tego wykładu*.

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*



Środa
14 grudnia 2013

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?

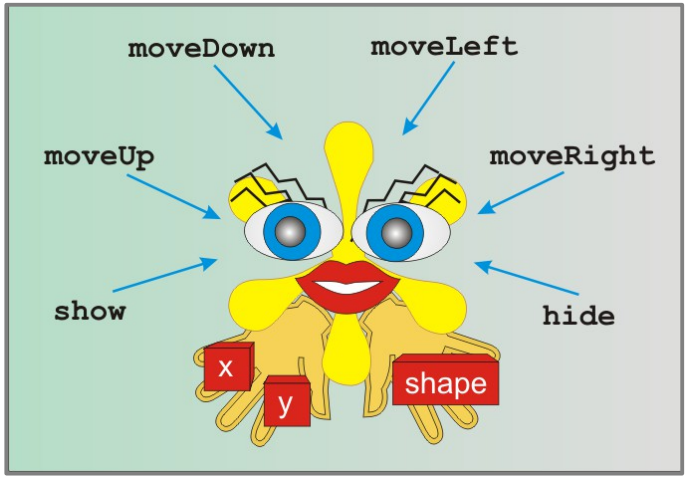
— To zależy jak się patrzy. Pamiętasz coś z naszego ostatniego wywiadu redaktorku?

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 nie 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*...

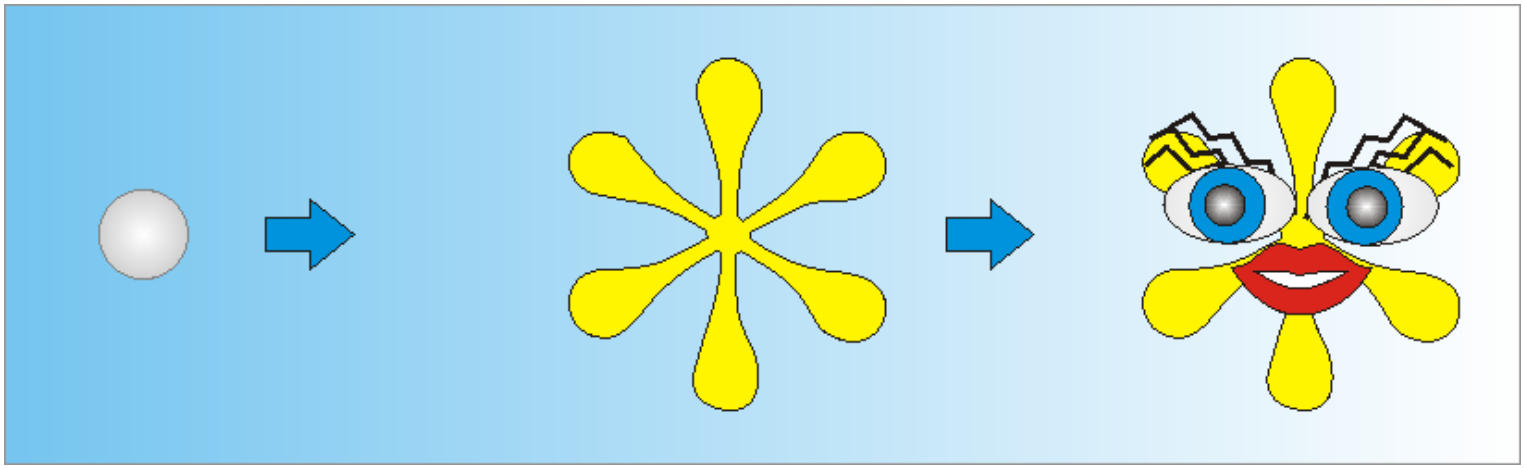
— Yes! Yes! Yes! Nareszcie zrozumiałeś. Ojcem klasy *Runner* jest właśnie *ScreenPoint*. Umiem wszystko to, co każdy obiekt tej klasy. Ale umiem więcej, bo w mojej klasę zdefiniowano nowe funkcje: *moveUp*(), *moveDown*(), *moveLeft*(), *moveRigt*().

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, nawet tego, że mój dziadek był głupiutki... .

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();
  }
}
```