Dziedziczenie implementacyjne, delegacja

Implementation inheritance, delegation

Beata Skiba[1], Amadeusz Stasiak[1], Alicja Żyła[1]

STRESZCZENIE: Celem tego artykułu jest przedstawienie dziedziczenia implementacyjnego oraz delegacji. Zostanie przedstawione jak działają obie techniki, a także jakie mają zastosowania, ograniczenia oraz problemy. Praca ma na celu przedstawienie również klasy dziedziczenia implementacyjnego oraz jego rodzajów elementów takich jak elementy prywatne, chronione i publiczne. Ograniczeniami dziedziczenia implementacyjnego są pojedynczość i niezmienność. Dziedziczenie implementacyjne posiada również swoje moduły programowe oraz topologię sieci. Delegacja ma własne projektowanie struktur danych oraz jest bardzo dobrą alternatywą dla dziedziczenia implementacyjnego. W artykule przedstawione jest jak jest tworzona delegacja oraz jakie posiada cechy. Podczas wykorzystywania delegacji pomocne jest programowanie zdarzeniowe. Zaprezentowane jest także działanie programów na starszych systemach przy użyciu deklaracji. Zarówno dziedziczenie implementacyjne jak i deklaracja są ważną częścią programowania obiektowego. Przedstawione są bardzo ważne różnice między dziedziczeniem implementacyjnym a delegacją. Na końcu referatu pokazane są spostrzeżenia odnośnie do dziedziczenia implementacyjnego oraz delegacji.

Słowa kluczowe: Dziedziczenie implementacyjne, delegacja, programowanie sterowane zdarzeniami.

ABSTRACT: The aim of this paper is to present the implementation inheritance and delegation. Will be presented how both techniques work, and what are their applications, limitations and problems. The article aims to present the implementation inheritance class and its types of elements such as private, protected and public elements. The constraints of implementation inheritance are singularity and immutability. Implementation inheritance has many limitations, such as composition, admixture, interface and features. Implementation inheritance also has its program modules and a network topology. The delegation has its own design of data structures and is a very good alternative to implementation inheritance. This paper outlines how a delegation is created and what features it has. Event programming is helpful when using delegations. The operation of programs on older systems using the declaration is also presented. Both implementation, inheritance and declaration are an important part of object-oriented programming. Very important differences between implementation inheritance and delegation are also presented. At the end of the paper, the observations about implementation inheritance and delegation are shown.

Keywords: Implementation inheritance, Delegation, Event-driven programming.

1. WSTĘP

Dziedziczenie implementacyjne i delegacja są jednymi z wielu elementów programowania obiektowego. Jak każde, mają swoje określone funkcje, które są niezbędne do napisania niektórych programów czy aplikacji. Ułatwiają nam swoją istotą pisanie kodu, który staje się krótszy o wiele linijek i zajmuje mniej pamięci operacyjnej na naszym sprzęcie. Niemalże każdy programista używa ich do swojego projektu na co dzień.

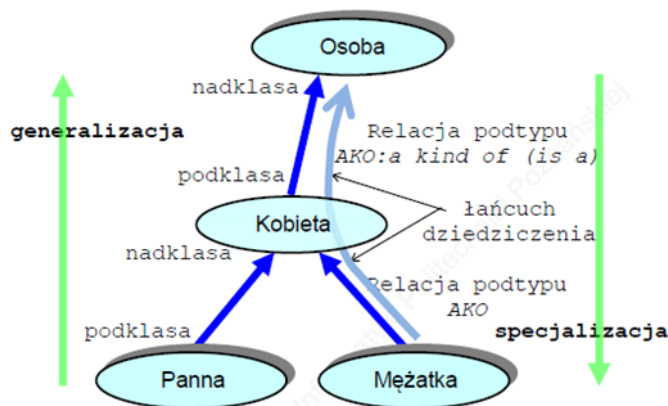
2. DZIEDZICZENIE IMPLEMENTACYJNE - definicja, zastosowanie, ograniczenia, rozwiązania ograniczeń

2.1 Dziedziczenie implementacyjne

Dziedziczenie implementacyjne jest to mechanizm współdzielenia funkcjonalności między klasami. Klasa może dziedziczyć po innej klasie, co oznacza, że oprócz swoich własnych atrybutów oraz zachowań uzyskuje także te pochodzące z klasy, z której dziedziczy. Klasa dziedzicząca jest nazywana klasą pochodną lub potomną, zaś klasa, z której następuje dziedziczenie — klasą bazową.

1.Wrocławska Wyższa Szkoła Informatyki Stosowanej “Horyzont” Wejherowska 28, 54-239 Wrocław beata.skiba1994@gmail.com, stasiak.amadeusz@gmail.com, alicja.zyla96@gmail.com.

Z jednej klasy bazowej można uzyskać dowolną liczbę klas pochodnych. Klasy pochodne posiadają obok swoich własnych metod i pól również kompletny interfejs klasy bazowej. W językach programowania z prototypowaniem (np. JavaScript) nie występuje pojęcie klasy, dlatego dziedziczenie implementacyjne zachodzi tam pomiędzy poszczególnymi obiektami. Pojęcie dziedziczenia implementacyjnego zostało wprowadzone po raz pierwszy przez twórców języka Simula.[14]



Ryc. 1 Przykład dziedziczenia.
Fig. 1 Example of inheritance

2.2 Klasy oraz rodzaje dziedziczenia implementacyjnego

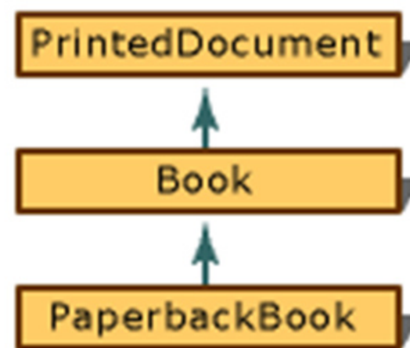
Zależności między klasami bazowymi i pochodnymi tworzą tak zwane hierarchie klas. Klasy pochodne otrzymują wszystkie metody i atrybuty swoich klas bazowych oraz mogą dodawać nowe. Dopuszczalne jest także nadpisywanie istniejących metod, przy czym poszczególne języki programowania mogą żądać spełnienia dodatkowych warunków, np. pozostawienia niezmienionej listy argumentów wejściowych i typu wyniku. Wiele języków programowania umożliwia deklarowanie klas jako abstrakcyjnych. Nie można tworzyć obiektu klasy abstrakcyjnej, lecz można po takiej klasie dziedziczyć. Klasa abstrakcyjna może zawierać metody czysto wirtualne, które muszą zostać zaimplementowane przez klasy pochodne. Mechanizmu tego używa się, jeśli twórca klasy chce dostarczyć jedynie części funkcjonalności, tworząc szkielet dla innych, bardziej wyspecjalizowanych klas. W części języków programowania istnieje możliwość ograniczania widoczności dziedziczonych pól i metod:

Elementy publiczne — nieograniczony dostęp, można je wywoływać zarówno z wnętrza klas, jak i spoza nich. Wadą ich jest to, że może dojść do przypadkowej zmiany ich wartości, co może zakłócić działanie reszty programu.

Elementy chronione — można je wywoływać jedynie z wnętrza klasy oraz z wnętrza wszystkich klas pochodnych. Najczęściej stosowane przy dziedziczeniu implementacyjnym w klasie bazowej.

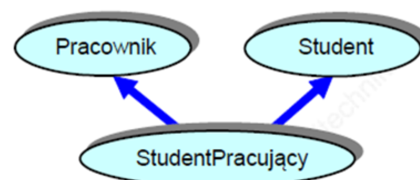
Elementy prywatne — można je wywoływać jedynie z wnętrza bieżącej klasy, natomiast nie ma do nich dostępu w klasach pochodnych. Najbezpieczniejsze zabezpieczenie danych ze względu na to, że można je zmienić tylko gotowymi funkcjami. Używane przy klasach pochodnych ostatniego rzędu.

W programowaniu obiektowym wyróżniane są dwa dziedziczenia implementacyjne. Jest to dziedziczenie pojedyncze oraz dziedziczenie wielokrotne. Z dziedziczeniem pojedynczym mamy do czynienia, gdy klasa pochodna dziedziczy po dokładnie jednej klasie bazowej (oczywiście klasa bazowa wciąż może dziedziczyć z jakiejś innej klasy), tak jak jest to pokazane na rysunku poniżej:



Ryc. 2 Dziedziczenie pojedyncze. [5]
Fig. 2 Single inheritance

Natomiast w dziedziczeniu wielokrotnym klas bazowych może być więcej. Wielokrotne dziedziczenie jest obsługiwane w takich językach, jak C++, Common Lisp czy Perl. Rysunek poniżej pokazuje nam dziedziczenie wielokrotne:

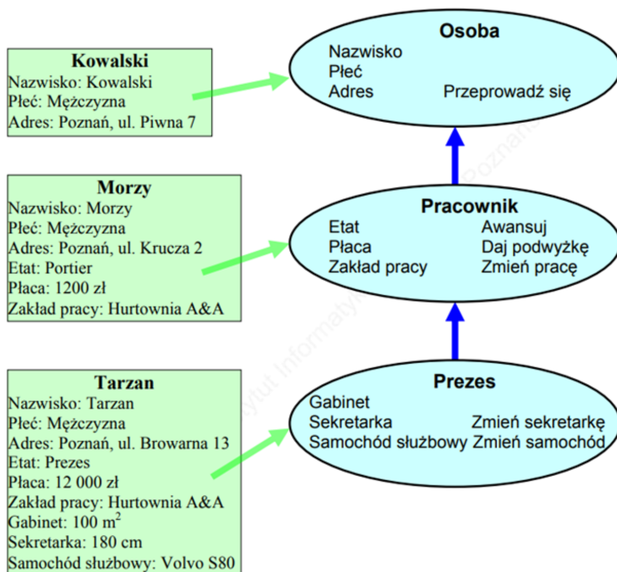



```
class Pracownik {
    ...
    float płaca;
    char *etat;
    ... };
class Student {
    ...
    char *uczelnia;
    int rokStudiów;
    float średniaOcen;
    ... };

class StudentPracujący : public Pracownik,
    public Student { ... };
```

Ryc. 3 Dziedziczenie wielokrotne w C++. [1]
Fig. 3 Multiple inheritance in C++

Zwiększa ono możliwości ponownego wykorzystania kodu, lecz jednocześnie jest krytykowane za:

A. Niejednoznaczność semantyczną (tzw. Diamond problem),
B. Problemy z łańcuchowym wywoływaniem konstruktorów,
C. Trudności implementacyjne.



Ryc. 4 Implementacja dziedziczenia implementacyjnego. [1]
Fig. 4 Impementation of implementation inheritance

Klasy abstrakcyjne są to klasy, które nie mają kompletnej implementacji. W związku z tym klasy te nie mogą mieć wystąpień. Mogą one służyć jedynie jako pośredni etap dla dziedziczących po nich klas zawierających implementację cech abstrakcyjnych. Zdefiniowanie danej klasy jako abstrakcyjnej uniemożliwia tworzenie wystąpień tej klasy. Własna implementacja klasy abstrakcyjnej wiąże się z występowaniem błędów w czasie wykonywania programów [1].
Powyższe problemy dotyczą przede wszystkim konfliktów implementacji. Dlatego nawet jeśli
w danym języku programowania wielokrotne dziedziczenie klas jest niedozwolone, można je stosować w przypadku interfejsów, które mogą być traktowane jak klasy abstrakcyjne zawierające wyłącznie metody czysto wirtualne [10][11].

2.3 Zastosowanie dziedziczenia implementacyjnego.

Podstawowym zastosowaniem dziedziczenia implementacyjnego jest ponowne wykorzystanie kodu. Jeśli dwie klasy wykonują podobne zadania, możemy utworzyć dla nich wspólną klasę bazową, do której przeniesiemy identyczne metody oraz atrybuty. Ułatwi to testowanie oraz potencjalnie zwiększy niezawodność aplikacji w przypadku zmian. W razie ewentualnych problemów łatwiej będzie również odnaleźć przyczynę błędu. Poniżej znajduje się przykład dziedziczenia implementacyjnego w języku C++:

```
class Osoba {
protected:
    char nazwisko [Max];
    char płeć;
    unsigned wiek;
public:
    Osoba (char*, char, unsigned);
    Void ZmieńNazwisko(char*);
    void DaneOsobowe( );
};                                          klasa bazowa

class Pracownik : public Osoba {
protected:
    char Etat [Max];        //nowa cecha
    unsigned płaca;         //nowa cecha
public:
    Pracownik (char*, char, unsigned, char*);
    void Awansuj(char*);   //nowa cecha
    void DajPodwyżkę(unsigned); //nowa cecha
    void DaneOsobowe( );   //redefinicja cechy
};
...
Osoba Morzy("Morzy",'M',45);
Morzy.DaneOsobowe();
Pracownik Buła("Buła",'M',38,"portier");
Buła.Awansuj("prezes");
Buła.ZmieńNazwisko("Tarzan");
Buła.DaneOsobowe();
```

Ryc. 5 Przykład dziedziczenia implementacyjnego w języku C++. [1]
Fig. 5 Example of implementation inheritance in C++

W języku C++ nie są dziedziczone:

A. konstruktory i destruktory klasy
B. przeciążony operator=()
C. „przyjaciele" klasy [1]

2.4 Hierarchia klas

Hierarchia klas może przekładać się na hierarchię typów. Możliwe jest wtedy podstawienie pod zmienną (lub atrybut funkcji) typu T obiektu typu S będącego podtypem T i dalsze używanie go jakby był typu T. Jest to możliwe dzięki temu, że podklasa posiada kompletny interfejs swojej nadklasy.
W podklasie może być zdefiniowana metoda już istniejąca w nadklasie. Konstrukcja taka umożliwia wykonywanie operacji na obiektach bez informacji, z jakim właściwie obiektem mamy do czynienia. Rozpatrzmy typową aplikację GUI wyświetlającą na ekranie różne komponenty (np. przycisk, pole tekstowe czy listę rozwijaną). Reagują one na te same zdarzenia: kliknięcie myszką, naciśnięcie klawisza, lecz każdy z nich reaguje inaczej, stosownie do tego, czym jest. System obsługi zdarzeń najpierw określa, który z komponentów powinien obsłużyć zdarzenie, a następnie przekazuje mu je. Pobierając aktywny obiekt, możemy wywołać tę metodę bez zastanawiania się czy dany obiekt jest przyciskiem, czy polem tekstowym. Decyzja
o tym, która wersja zachowania zostanie wywołana
w konkretnym miejscu, zależy od języka programowania i sposobu zdefiniowania metod. Rozpatrzmy następującą sytuację:

```
class A {
   method foo();
}

class B extends A {
   method foo();
}

A obiektBazowy = new A();
B obiektPochodny = new B();
obiektBazowy.foo(); // 1

obiektBazowy = obiektPochodny;
obiektBazowy.foo(); // 2
```

Ryc. 6 Przykład hierarchii klas. [1]
Fig. 6 Example of class hierarchy

Mamy klasę bazową A oraz dziedziczącą z niej klasę B. Klasa bazowa definiuje metodę foo(), która jest nadpisywana przez klasę pochodną. Przypadek pierwszy nie budzi żadnych wątpliwości: mamy utworzony obiekt klasy A, dlatego wywołujemy wersję metody foo() zdefiniowaną w tej klasie. W przypadku drugim pod zmienną obiekt Bazowy podstawiony jest obiekt klasy pochodnej. Jednak wtedy w linijce oznaczonej przez 2 możemy wywołać wersję metody foo() z klasy A, mimo iż zmienna wskazuje na obiekt klasy pochodnej posiadającej zmodyfikowaną wersję tej metody lub wywołać wersję metody foo() z klasy pochodnej B, mimo iż zmienna obiektBazowy jest typu A.
Jeśli zachodzi sytuacja druga, powiemy, że metoda foo() jest metodą wirtualną. W niektórych językach (np. C++) metody muszą być jawnie deklarowane jako wirtualne przez programistę.
W innych (np. Java) wszystkie metody są z definicji wirtualne.W ogólnym ujęciu podtypowanie i dziedziczenie to dwa różne pojęcia. Dziedziczenie dotyczy powtórnego wykorzystania klasy bazowej, natomiast podtypowanie (polimorfizm) możliwości wykorzystania podtypu w miejscu nadtypu.

2.5 Ograniczenia dziedziczenia implementacyjnego

Dziedziczenie implementacyjne posiada kilka ograniczeń wynikających z faktu, że hierarchia klas jest ustalana w momencie kompilacji programu i nie może podlegać późniejszym zmianom. Wyobraźmy sobie klasę Osoba, z której dziedziczą klasy Pracownik oraz Student. Napotykamy tutaj na istotne problemy:

Pojedynczość — w językach z pojedynczym dziedziczeniem osoba może być albo pracownikiem, albo studentem.
Niezmienność — nawet jeśli skorzystamy z wielokrotnego dziedziczenia i utworzymy klasę StudentPracownik, klasę wybieramy w momencie tworzenia obiektu i nie możemy jej później zmienić. Oznacza to, że system nie może poprawnie reagować na sytuacje, gdy dotychczasowy student zostaje dodatkowo pracownikiem, gdyż konieczne jest wtedy utworzenie całkowicie nowego obiektu.
Innym istotnym ograniczeniem jest uzależnienie kodu od konkretnej implementacji klasy, które może doprowadzić do błędów przy jej zmianie. Tego typu problem pojawia się zwłaszcza gdy dziedziczymy między klasami znajdującymi się w różnych komponentach (tzw. Problem kruchości klasy podstawowej).

2.6 Rozwiązania ograniczeń dziedziczenia implementacyjnego

Istnieje kilka rozwiązań alternatywnych eliminujących poszczególne ograniczenia dziedziczenia implementacyjnego, są to kompozycja, domieszki oraz interfejsy i cechy. Przykładami rozwiązań ograniczeń dziedziczenia implementacyjnego są:
Kompozycja — polega na zastąpieniu dziedziczenia implementacyjnego składaniem mniejszych obiektów. Posługując się dalej powyższym przykładem, możemy zostawić klasę Osoba, która posiada listę ról takich, jak Pracownik czy Student. Obiektowi można przydzielać nowe oraz usuwać stare rolę w dowolnym momencie wykonania programu, eliminując tym samym problemy pojedynczości oraz niezmienności. Wadą kompozycji jest większe zużycie pamięci (dużo małych obiektów) oraz niewielki spadek wydajności podczas dostępu do metod. Kompozycję można stosować w każdym języku obsługującym programowanie obiektowe.

Ryc. 7 Przykład kompozycji w agregacji. [1]
Fig. 7 Example of composition in aggregation

Domieszka — pozwala uzyskać funkcjonalność podobną do wielokrotnego dziedziczenia, unikając jednocześnie trapiących je paradoksów. Jest to rodzaj klasy abstrakcyjnej, którą można „dodać” do właściwych klas. Klasa uzyskuje wszystkie atrybuty oraz metody zdefiniowane w dodanych do niej domieszkach. Oddzielenie klas od domieszek pozwala wprowadzić jasne reguły rozwiązywania konfliktów. Przykładem języka wykorzystującego domieszki jest Ruby.
Interfejs — to rodzaj klasy abstrakcyjnej, która może zawierać wyłącznie metody czysto wirtualne oraz stałe. Ponieważ paradoksy dotyczą wyłącznie implementacji, której tu nie ma, w interfejsach można bezpiecznie korzystać z wielokrotnego dziedziczenia. Również klasy mogą implementować więcej niż jeden interfejs jednocześnie. Przykładami języków wykorzystujących interfejsy są Java oraz C#. Język Java oprócz klas abstrakcyjnych obejmuje dodatkowo pojęcie interfejsu, który może być traktowany jako szczególna klasa abstrakcyjna, która w ogóle nie po-

siada implementacji. Definicja interfejsu nie może zawierać elementów prywatnych, definicji zmiennych obiektów oraz metod typu final. Interfejsy powinny być implementowane przez klasy. Pojedyncza klasa może implementować wiele interfejsów. Dany interfejs może reprezentować pojedynczy aspekt klasy. Mogą być one powiązane siecią dziedziczenia. [1]

```
interface Interfejs{

  public void proc();

}

class Klasa implements Interfejs{

  @Override
  public void proc(){
    // ...
  }
}
```

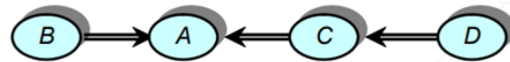
Ryc. 8 Przykład interfejsu w Java. [1]
Fig. 8 Java interface example

Cechy — umożliwiają wielokrotne wykorzystanie tego samego kawałka kodu w różnych klasach.
W przeciwieństwie do domieszek kod ten zachowuje się tak, jakby był zapisany w tych klasach bezpośrednio, a w momencie wykonania programu nie ma możliwości stwierdzenia, czy dana metoda została zaimplementowana bezpośrednio w klasie, czy dodana przez cechę.

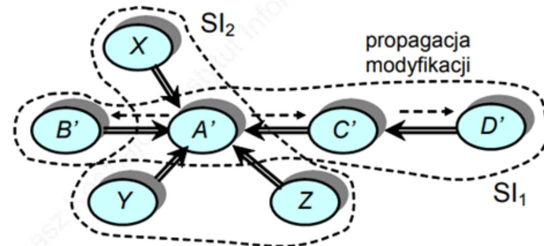
2.7 Moduły programowe

Moduły programowe powinny być jednocześnie otwarte i stabilne. Moduły programowe powinny być otwarte na dalszy rozwój: dodanie nowej funkcjonalności lub zmianę implementacji. Eksploatowane moduły powinny być stabilne. Modyfikowanie poprawnie działającego modułu może spowodować jego błędne działanie. Wymaganie otwartości modułów jest konsekwencją potrzeby rozwoju i utrzymania systemów informatycznych. Wymaganie stabilności modułów jest wynikiem potrzeby osiągnięcia stabilizacji eksploatowanych systemów informatycznych. Równoczesne spełnienie tych dwóch wymagań jest bardzo kłopotliwe.
Konsekwencją braku stabilności modułu jest współdzielenie kodu przez modyfikację.
Przykład możemy zaobserwować poniżej. [1]

Dany jest system informatyczny SI_1 obejmujący moduły programowe A, B, C i D.



Chcemy wykorzystać moduł A do budowy nowego systemu SI_2 składającego się z modułów X, Y, Z i zaadoptowanego modułu A. Chcąc utrzymywać tylko jedną wersję modułu A, rozszerzamy funkcjonalność modułu A, tak by spełniał on jednocześnie wymagania systemów SI_1 i SI_2.



Moduł A stracił stabilność. Jego modyfikacja do postaci A' może spowodować niepoprawne działanie systemu SI_1 i wymusić modyfikacje modułów B, C i D.

Ryc. 9 Przykład braku stabilności. [1]
Fig. 9 Example of no stability

2.8 Topologia sieci dziedziczenia implementacyjnego

Dziedziczenie jednokrotne (SmallTalk 80, C#, klasy w języku Java) - każda klasa ma co najwyżej jedną nadklasę. Sieć klas ma kształt hierarchii. Dziedziczenie wielokrotne (C++, Eiffel, interfejsy
w języku Java) - klasy mogą dziedziczyć po wielu nadklasach. Sieć klas ma kształt grafu acyklicznego skierowanego. Sieć klas może być rozłączna (C++) lub być niepodzielna i mieć wyróżniony wierzchołek, który jest korzeniem sieci (SmallTalk 80, Java, C#). [1]

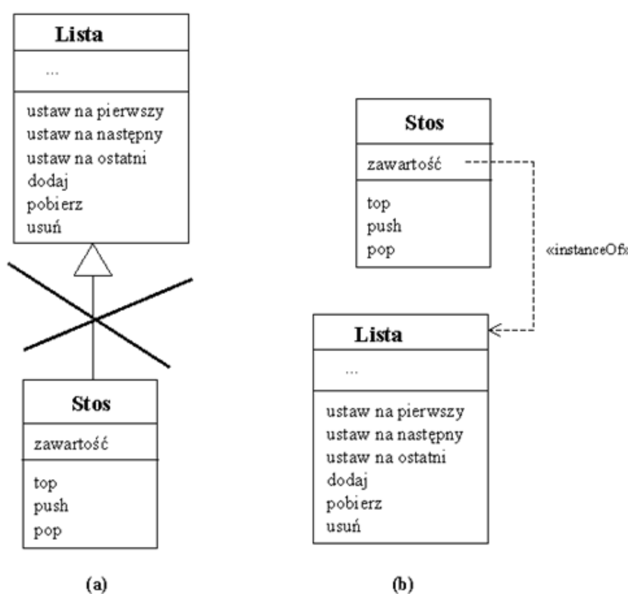
```
class Osoba {
protected:
    char *nazwisko;
    ...
public:
    void Nazwisko(char*);
    void wiek(int);
    ...
};
class Pracownik : public Osoba{ ... };
class Student : public Osoba { ... };
class StudentPracujący : public Pracownik,
    public Student { ... };

StudentPracujący sp(...);
sp.Nazwisko("Tarzan"); // niejednoznaczne wywołanie
sp.Student::Nazwisko("Tarzan"); // OK
sp.Pracownik::Nazwisko("Kowalski"); // OK
-- student sp ma dwa niezależne nazwiska
```

Ryc. 10 Przykład dziedziczenia wielokrotnego z tego samego źródła. [1]
Fig. 10 Example of multiple inheritance from the same source

3.1 Projektowanie struktur danych w delegacji

Delegacja jest bardzo dobrą alternatywą dla dziedziczenia implementacyjnego oraz koncepcją projektowania struktur danych, zgodnie z którą operacje, które można wykonać na danym obiekcie, są własnością innego obiektu; innymi słowy, są "oddelegowywane" do innego obiektu. Delegację można uważać za alternatywę dla klasycznego mechanizmu dziedziczenia implementacyjnego, gdyż w przypadku delegacji mamy do czynienia z dziedziczeniem dynamicznym.

Na rysunku poniżej do implementacji stosu wykorzystana jest implementacja listy. Na diagramie (a) klasa Stos jest podklasą klasy Lista, w związku z czym dziedziczy jej wszystkie własności. Nie jest to jednak sytuacja pożądana, ponieważ stos powinien udostępniać użytkownikowi tylko właściwe dla siebie operacje takie jak push, pop, top itd., podczas gdy udostępnia także operacje charakterystyczne dla listy. Alternatywne rozwiązanie posiadające tę cechę przedstawia diagram (b), gdzie klasa Stos zawiera atrybut zawartość typu Lista - w ten sposób obiekt klasy Stos składa się z podobiektu będącego wystąpieniem klasy Lista, w wyniku czego obsługa obiektu modelującego stos jest (częściowo) oddelegowana do obiektu modelującego listę. Na poziomie implementacyjnym takie rozwiązanie oznaczałoby wywołanie w ciele metod push, pop i top odpowiednich metod zdefiniowanych w klasie Lista. Bezpośredni dostęp do metod klasy Lista dla wystąpień klasy Stos, jak na schemacie (a), nie jest tu możliwy. [2]



Ryc. 11 Przykład delegacji.[2]
Fig. 11 Delegation example

3.2 Jak jest stworzona delegacja oraz jakimi regułami się cechuje

Delegacja jest także specjalnym typem danych, który przechowuje referencję (adres) do procedury lub funkcji.

W środowisku .NET delegacja jest odpowiednikiem wskaźnika (pointer) do funkcji znanego z języka C/C++. W małych projektach można się obyć bez używania delegacji, w większych jest to praktycznie niemożliwe. Środowisko .NET korzysta z delegacji powszechnie, a ich dobrym przykładem jest obsługa zdarzeń z późnym wiązaniem (z wykorzystaniem instrukcji AddHandler).

Nazwa delegacji powinna być zgodna z regułami tworzenia nazw w .NET. Delegacja może zawierać (ale nie musi) parametry, w przypadku zbudowania delegacji do funkcji obowiązkowo musi być określony zwracany typ danych. Delegacja może wskazywać adres tylko tej procedury czy funkcji, której sygnatura (zestaw parametrów, zwracany typ danych) jest zgodna z sygnaturą delegacji.

Przed wykorzystaniem delegacji musimy utworzyć procedury czy funkcje, które będziemy chcieli kojarzyć z delegacją. Obowiązuje zasada, że z delegacją można skojarzyć dowolną procedurę czy funkcję pod jednym warunkiem: zgodności sygnatur. Jeżeli instancja delegacji już istnieje, to można zmienić adres procedury czy funkcji. Składnia języka C++ umożliwia definiowanie klas abstrakcyjnych. Kompilator języka uniemożliwia tworzenie zmiennych (typów funkcji i metod oraz parametrów wejściowych funkcji i metod), których typem jest klasa abstrakcyjna. [3]

```
' utworzenie delegacji do procedury z jednym parametrem typu String
Delegate Sub MessageDelegate(ByVal jg As String)
' utworzenie dwóch procedur, które będą wywoływane via instancja
 delegacji
Private Sub SendToMessageBox(ByVal txt As String)
   MessageBox.Show(txt, Me.Text)
End Sub


Private Sub SendToEventLog(ByVal txt As String)
   ' utworzenie logu aplikacji
   Dim AppLog As New EventLog
   AppLog.Source = Me.Text
   AppLog.WriteEntry(txt)
End Sub
' utworzenie instancji delegacji z jednym parametrem typu String
Dim SendMessage As MessageDelegate
' utworzenie instancji delegacji typu MessageDelegate wskazującej
  adres procedury SendToMessageBox
SendMessage = New MessageDelegate(AddressOf SendToMessageBox)
' utworzenie instancji delegacji typu MessageDelegate wskazującej
  adres procedury SendToEventLog
SendMessage = New MessageDelegate(AddressOf SendToEventLog)
```

Ryc. 12 Przykład skojarzenia delegacji z procedurą. [3]
Ryc. 12 Example of associating a delegation with procedure

3.3 Programowanie zdarzeniowe

Programowanie sterowane zdarzeniami jest metodologią tworzenia programów komputerowych, która określa sposób ich pisania z punktu widzenia procesu przekazywania sterowania między poszczególnymi modułami tej samej aplikacji. Programowanie sterowane zdarzeniami jest mocno powiązane ze środowiskami wieloprocesowymi, z graficznymi środowiskami systemów operacyjnych oraz z programowaniem obiektowym.

Jest to paradygmat programowania, według którego program jest cały czas „bombardowany" zdarzeniami, na które musi odpowiedzieć, i zakładający, że przepływ sterowania w programie jest całkowicie niemożliwy do przewidzenia z góry.

Programowanie zdarzeniowe jest dominującym typem programowania GUI – zdarzenia to naciśnięcia myszy,

klawiszy, żądania odświeżenia przez system okienkowy, różne zdarzenia sieciowe i inne.
Jest też używane przez wysoce wydajne serwery sieciowe – zdarzeniami są tu żądania połączenia, nadejście danych do odbioru, zwolnienie się miejsca w buforach wysyłania odbioru itd.
W systemach uniksowych zwykle wszystkie połączenia (np. z plikami, sieciowe, z relacyjną bazą danych) mają charakter deskryptorów plików i na ich zbiorze jest wywoływana funkcja systemu operacyjnego select lub poll, która informuje, na jakim deskryptorze wydarzyło się jakieś zdarzenie.
W programowaniu zdarzeniowym ważne jest żeby nie obsługiwać zbyt długo danego zdarzenia, bo blokuje się w ten sposób obsługę innych. W przypadku serwerów obniżyłoby to znacznie wydajność, w przypadku GUI program zbyt wolno odpowiadałby na akcje użytkownika. Można to osiągnąć za pomocą asynchronicznego I/O, wielowątkowości, rozbijania zdarzenia na podzdarzenia i wielu innych mechanizmów. [12][13]

4.1 Działanie programów na starszych systemach

W starych systemach operacyjnych takich jak na przykład DOS w jednej chwili w komputerze mogła pracować tylko jedna aplikacja. Sterowanie jej działaniem odbywało się bądź poprzez jawne zadeklarowanie kolejności wykonania kodu (w programie jednoprzebiegowym), bądź poprzez pętlę obsługi wyboru opcji (w programie interaktywnym). W miarę rozwoju środowisk wieloprocesowych ten stary sposób obsługi interaktywności przestał się sprawdzać. W jednej chwili w systemie operacyjnym mogło pracować wiele różnych programów. Projektanci systemów nie mogli pozwolić poszczególnym programom na całkowite przejęcie kontroli nad systemem w taki sposób, by inne programy nic nie mogły robić. W związku z tym zaczęły pojawiać się pomysły na zdarzeniowe sterowanie programami. Idea tego rozwiązania była zupełnie inna od dotychczas stosowanej. Pętla odbierająca zdarzenia od urządzeń zewnętrznych, które pozwalały użytkownikowi komunikować się z programem, została utworzona tylko w jednym egzemplarzu dla całego systemu operacyjnego. Obsługą tej pętli (w tym przekazywaniem sterowania do poszczególnych programów) zajmował się sam system. Wszystkie procedury reakcji na zdarzenia nie są więc wywoływane jawnie w każdym programie. Zamiast tego we wszystkich programach tworzone są delegacje do obsługi odpowiednich zdarzeń.
Kilka znaczących różnic między „starą" pętlą obsługi zdarzeń a programowaniem sterowanym zdarzeniowo, oto one:

A. Nie występuje (w sposób jawny) pętla obsługi zdarzeń.
B. Rozdział zdarzeń scedowano na system operacyjny.
C. Konwersja zdarzeń z poziomu sprzętu (pochodzących od urządzeń zewnętrznych takich jak klawiatura, mysz czy ekran dotykowy) na zdarzenia typu znaczeniowego (na przykład „wybrano opcję menu", „naciśnięto przycisk ekranowy" itd.).
D. Obsługa menu (zmiana wyglądu i przejście między jego opcjami) jest wykonywana przez system operacyjny (również niejawnie).

Przykładem programowania zdarzeniowego może być porównanie metod sterowania programem.

```
PoczątekProgramu
WyświetlMenu(
  "1.Faktura",
  "2.Dodaj klienta",
  "3.Indeksacja bazy",
  "4.Koniec pracy")
AktywnaOpcja=1
PowtarzajPętlę:
  PodświetlMenu(AktywnaOpcja)
  PobierzDaneZKlawiatury(Znak)
  Jeżeli (Znak=Enter)
  wtedy WybierzAkcję(AktywnaOpcja):
    1: WystawFakturę
    2: DodajNowegoKlienta
    3: IndeksujBazęDanych
    4: PrzerwijPętlę
  KoniecWyboruAkcji
  Jeżeli (Znak=(Strzałka, Home, End, PgUp, PgDn))
  wtedy ZmieńAktywnąOpcję(AktywnaOpcja, Znak)
KoniecPętli
KoniecProgramu
```

Ryc. 13 Przykład fragmentu decyzyjnego zapisanego w pseudokodzie starsza wersja. [3]
Fig. 13 Example of decision fragment written in pseudocode - older version

```
PoczątekProgramu
ZaładujProceduryObsługiZdarzeńDoPamięci(
  WystawFakturę,
  DodajNowegoKlienta,
  IndeksujBazęDanych)
DodajDoSystemuMenu(
  "1.Faktura",
  "2.Dodaj klienta",
  "3.Indeksacja bazy",
  "4.Koniec pracy")
ZdefiniujPrzypisanieZdarzeńDoProcedur(
  NaZdarzenie WybranoOpcjęFaktura reaguj:
  WystawFakturę
  NaZdarzenie WybranoOpcjęDodajKlienta reaguj:
  DodajNowegoKlienta
  NaZdarzenie WybranoOpcjęReindeksacja reaguj:
  IndeksujBazęDanych
  NaZdarzenie WybranoOpcjęKoniecPracy reaguj:
  (UsuńZPamięciProceduryObsługiZdarzeń,
  UsuńDefinicjePrzypisaniaObsługiZdarzeńTegoProgramu,
  PrzekażSterowanieSystemowiOperacyjnemu)
)
PrzekażSterowanieSystemowiOperacyjnemu
KoniecProgramu
```

Ryc. 14 Przykład fragmentu decyzyjnego zapisanego w pseudokodzie- nowa wersja. [3]
Fig. 14 Example of decision fragment written in pseudocode - new version

4.2 Programowanie obiektowe

Programowanie obiektowe zaczęło być popularne wraz z rozpowszechnieniem się środowisk graficznych, które miały charakter obiektowy, mimo iż często powstawały jeszcze w językach strukturalnych. Jednocześnie ze względu

na swoje możliwości środowiska próbowały implementować wieloprocesowość. Nic więc dziwnego, że obie te metody tworzenia programów scaliły się ze sobą. Było to o tyle naturalne, że połączenie programowania obiektowego i sterowanego zdarzeniami znacznie podniosło walory obu tych technik. Zamiast uruchamiać jakieś procedury na podstawie zdarzenia, lepiej przesyłać je do konkretnego obiektu – niech on obsłuży je sobie we własnej piaskownicy. System zyskiwał na tym większą swobodę działania, nie musiał decydować o tym, jaką procedurę wykonać po naciśnięciu przycisku ekranowego 1, a jaką po 2. Odsyłał do obiektu-okna zdarzenie „naciśnięto przycisk ekranowy (numer przycisku)" i był wolny. Programowanie obiektowe zyskało hermetyzację i bezpieczeństwo. System nie wtrącał się do tego, co obiekt robi z tym zdarzeniem. Obecnie nikt już nie wyobraża sobie innego sposobu sterowania programem obiektowym niż za pomocą zdarzeń. [15][16]

4.3 Różnice między dziedziczeniem implementacyjnym a delegacją

Obie funkcje są różnymi narzędziami i pełnią inne funkcje w kodzie programisty. Dziedziczenie implementacyjne służy do ułatwienia procesu tworzenia klas i skrócenia kodu. Funkcje podrzędne w dziedziczeniu implementacyjnym zachowują pełne zachowanie klasy nadrzędnej,
a unikamy zbędnego kopiuj/wklej. Jednak możemy dziedziczyć tylko po jednej klasie, dlatego musimy uważać żeby jak najlepiej wykorzystać tę funkcję. Delegacja służy do wykonania pewnego działania. Nie tworzy się cała podklasa, tylko wykorzystana zostaje konkretna funkcja, bądź zmienione zostają konkretne dane z klasy, dlatego często wykonuje się ona szybciej i bez zbędnego zabierania pamięci systemowej. Dodatkowo możemy dzięki niej posiadać referencję do wielu klas oraz udostępniać ich metody. [4]

4.4 Porównania

Dziedziczenie można porównać do tworzenia stylów akapitowych w programie InDesign. W programie tym tworzymy styl dla tekstu, dzięki czemu mamy cały czas dostęp do zmian. Wystarczy, że raz zaplanujemy jak ma ten tekst wyglądać, a gdy będziemy chcieli zmienić napisany przez nas tekst, nie będziemy musieli martwić się o upewnienie, czy na pewno nowy i stary tekst jest taki sam. Gotowe style możemy powielać i edytować podobnie jak z klasami pochodnymi w dziedziczeniu. Obie te rzeczy ułatwiają pracę, jedne dziennikarzom i autorom książek, a te drugie programistom. [9]

Delegacje można zobrazować na przykładzie aplikacji Trello w której możemy powierzać ludziom zadania na dany dzień lub też przydzielać je całym grupom osób. Dzięki temu prace mogą iść w lepszy, bardziej zorganizowany sposób, łatwiej jest też zaplanować przebieg prac oraz na jakim etapie się aktualnie znajdujemy. W delegacji jest podobnie, pozwala ona zapanować nad chaosem i umożliwia lepszą organizację tego kodu. [6][7][8]

5. WNIOSKI

Wykorzystywanie dziedziczenia implementacyjnego oraz delegacji jest bardzo potrzebne, skuteczne
a także przyjemne. Są przydatnymi narzędziami programistycznymi i powinny być wykorzystywane w każdym większym projekcie czy kodzie, gdyż dzięki nim tekst programisty staje się bardziej przejrzysty i łatwiej z niego korzystać oraz pisać czy też poprawiać. Funkcje te pozwalają łatwiej rozszerzyć kod i rozwijać go w przyszłości przy dalszych zmianach w projekcie.

LITERATURA

[1] http://fc.put.poznan.pl/materials/138-4--dziedziczenie.pdf
[2] http://edu.pjwstk.edu.pl/wyklady/pri/scb/index70.html
[3]https://forum.pasja-informatyki.pl/19901/dziedziczenie-a-delegacja
[4]https://www.altkomakademia.pl/baza-wiedzy/qna/discussion/1788/jaka-jest-roznica-miedzy-delegacja-a-dziedziczeniem/p1
[5] https://msdn.microsoft.com/pl-pl/library/84eaw35x.aspx
[6]http://wniedoczasie.pl/narzedzia/trello-zarzadzanie-zadaniami/
[7]https://www.spidersweb.pl/2016/04/trello-po-polsku.html
[8]https://www.kreatywa.net/2016/04/trello-opinia.html
[9]https://www.youtube.com/watch?v=Ba5ZujkR21o
[10]http://cpp0x.pl/kursy/Programowanie-obiektowe-C++/Polimorfizm/Dziedziczenie/494
[11]https://pl.wikibooks.org/wiki/C%2B%2B Dziedziczenie
[12]https://pl.wikipedia.org/wiki Programowanie_sterowane_zdarzeniami
[13]https://4programmers.net/Forum/C_i_C++/248556-programowanie_zdarzeniowe
[14]http://icis.pcz.pl/~agrosser/test/mijp-I.pdf
[15]https://pl.wikipedia.org/wiki/Programowanie_obiektowe
[16]https://webmastah.pl/jak-programowac-obiektowo-cz-1-wstep/