Pracownia programowania obiektowego

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf".
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Pracownia programowania obiektowego

Andrzej Daniluk

Lublin 2011

Instytut Informatyki UMCS
Lublin 2011

Andrzej Daniluk

PRACOWNIA PROGRAMOWANIA OBIEKTOWEGO

Recenzent: Paweł Wnuk
Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
ul. Ratajczaka 26/8
61-815 Poznań
www: www.esus.pl

ISBN: 978-83-62773-01-5

SPIS TREŚCI

PRZEDMOWA ... VII
JĘZYK MODELOWANIA ... 1
1.1. Budowanie modeli ... 2
1.2. UML jako język modelowania ... 3
1.3. Zawartość UML ... 5
Podsumowanie ... 53
PROJEKTOWANIE OPROGRAMOWANIA ... 55
2.1. Proces kaskadowy ... 56
2.2. Proces iteracyjny ... 57
2.3. Architektura sterowana modelem ... 69
Podsumowanie ... 73
DZIEDZICZENIE I DELEGOWANIE ... 75
3.1. Mechanizm dziedziczenia ... 76
3.2. Odwołania i wskaźniki do klas pochodnych ... 77
3.3. Wskaźniki do funkcji składowych ... 80
3.4. Funkcje wirtualne ... 83
3.5. Obiekty funkcyjne w hierarchii dziedziczenia ... 90
3.6. Statyczny polimorfizm ... 92
3.7. Dziedziczenie wielokrotne ... 98
3.8. Informacja czasu wykonywania ... 104
3.9. Delegowanie ... 110
Podsumowanie ... 112
INTERFEJSY ... 113
4.1. Interfejsy a klasy abstrakcyjne ... 114
4.2. Zliczanie odwołań do interfejsu ... 117
4.3. Delegowanie obiektów poprzez wspólny interfejs ... 124
4.4. Programowanie obiektowe z użyciem interfejsów ... 127
4.5. Składniki ... 131
Podsumowanie ... 133
WZORCE PROJEKTOWE PROGRAMOWANIA OBIEKTOWEGO .. 135
5.1. Podział wzorców ... 136
5.2. Singleton ... 137
5.3. Fabryka abstrakcyjna ... 140
5.4. Prototyp ... 143
5.5. Metoda wytwórcza ... 146
5.6. Budowniczy ... 149
5.7. Adapter ... 151
5.8. Dekorator ... 155
5.9. Fasada ... 158
5.10. Pełnomocnik ... 160
5.11. Kompozyt ... 165
5.12. Most ... 166
5.13. Pyłek ... 167

5.14. Metoda szablonu 168
5.15. Strategia 171
5.16. Obserwator 174
5.17. Stan 179
5.18. Wizytator 182
5.19. Polecenie 186
5.20. Łańcuch odpowiedzialności 188
5.21. Interpreter 189
5.22. Iterator 193
5.23. Memento 195
5.24. Mediator 196
Podsumowanie 197
BIBLIOTEKI ŁĄCZONE DYNAMICZNIE 199
6.1. Biblioteki współdzielone 200
6.2. API Windows 201
6.3. Klasy jako elementy bibliotek dołączanych dynamicznie 208
6.4. GNU/Linux 212
Podsumowanie 216
OBSŁUGA WYJĄTKÓW 217
7.1. Standardowa obsługa wyjątków 218
7.2. Klasy wyjątków 225
Podsumowanie 227
MODELOWANIE INTERFEJSU GUI 229
8.1. Elementy GUI 230
8.2. Konstruowanie GUI. Podejście deklaratywno-proceduralne 231
8.3. Konstruowanie GUI. Podejście RAD 236
8.4. Przydatne techniki modelowania GUI 238
Podsumowanie 240
PRZYKŁADOWE ĆWICZENIA DO SAMODZIELNEGO WYKONANIA 241
Zadania do rozdziału 3 242
Zadania do rozdziału 4 242
Zadania do rozdziału 5 242
Zadania do rozdziału 6 243
Zadania do rozdziału 7 243
Zadania do rozdziału 8 244
BIBLIOGRAFIA 245
SKOROWIDZ 249

PRZEDMOWA

Programowanie obiektowe, lub, programowanie zorientowane obiektowo OOP (ang. *object-oriented programming*) od wielu lat pozostaje podstawową techniką tworzenia oprogramowania. Od czasu swoich początków, tj. od lat sześćdziesiątych XX wieku, kiedy opracowano język Simula 67, obiektowy styl programowania znalazł szerokie zastosowanie w niemal każdym z obszarów informatyki. Kolejnym ważnym krokiem było wprowadzenie w 1980 roku języka Smalltalk-80. Jednak prawdziwy skok jakościowy w rozwoju metodyk obiektowych nastąpił wraz z opracowaniem C++. Najpopularniejsze obecnie języki obiektowe to C++, Java oraz C#. Najpopularniejsze, niezależne od przyjętego języka programowania techniki wspomagające projektowanie oprogramowania to zunifikowany język modelowania UML i wzorce projektowe programowania obiektowego. Praktyczna użyteczność stosowania UML i wzorców projektowych opiera się na założeniu, iż do implementacji modelu będzie stosowany język obiektowy.

Celem tego podręcznika jest przedstawienie obiektowych metod tworzenia oprogramowania opierających się na wykorzystaniu wzorców projektowych oraz zunifikowanego języka modelowania wspierającego zastosowanie tych metod. Zastosowany formalizm unika sposobu prezentacji technik obiektowych stosowanych w XX wieku, kiedy ograniczano się jedynie do zdefiniowania rzeczowników opisujących dany problem, a następnie na tej podstawie stworzono odpowiednie obiekty. Podejście takie sprowadzało się przede wszystkim do zaimplementowania mechanizmów hermetyzujących (ukrywających) dane, oraz zaimplementowania obiektów jako nieskomplikowanego pojęciowo zestawu danych i operujących metod. Tego typu rozumienie technik obiektowych koncentruje się jedynie na implementacji obiektów – jako połączenia kodu i danych zaniedbując perspektywy konceptualizacji i szczegółowej specyfikacji projektu.

Niniejszy podręcznik zawiera ogólne wprowadzenie do współczesnych metod wytwarzania oprogramowania w oparciu o paradygmat OOP, w którym zwrócono szczególną uwagę na zagadnienia stosunkowo rzadko omawiane w popularnej literaturze – relacje pomiędzy modelem i językiem modelowania, precyzyjną semantyką języka programowania oraz wybranymi decyzjami projektowymi.

W praktycznym użyciu istnieje obecnie wiele doskonałych obiektowych języków programowania takich jak: Java, C#, PHP, Object Pascal, jednak omawiane w tej książce przykłady będą prezentowane w terminologii C++. Chociaż w rankingach TIOBE Software od wielu lat Java wydaje się być liderem wśród współcześnie używanych języków programowania, należy jednak mieć świadomość faktu, iż interfejsy programistyczne najpopularniejszych obecnie systemów operacyjnych: Windows, Unix, GNU/Linux, BSD, QNX

konstruowane są w C/C++. W konsekwencji, z punku widzenia praktyki akademickiej, C++ powinien być traktowany jako język podstawowy.

Książki tej nie należy traktować jako podręcznika do nauki programowania. Zakres przedstawionego materiału obejmuje zagadnienia bardziej zaawansowane i wymaga od Czytelnika znajomości podstaw języka C++.

W trakcie przygotowywania niniejszego opracowania autor korzystał z dwóch kompilatorów języka C++: C++ Compiler 5.5 dla systemów operacyjnych Windows oraz g++ dla systemów operacyjnych GNU/Linux. Kompilator g++ jest obecnie standardowym elementem systemów operacyjnych GNU/Linux. C++ Compiler 5.5 jest kompilatorem języków C i C++ zgodnym ze standardem ANSI/ISO i generującym kod wykonywalny dla systemów Microsoft Windows. Pakiet zawiera bibliotekę STL. C++ Compiler 5.5 dostępny jest nieodpłatnie na stronie firmy Embarcadero (http://edn.embarcadero.com/article/20633).

ROZDZIAŁ 1

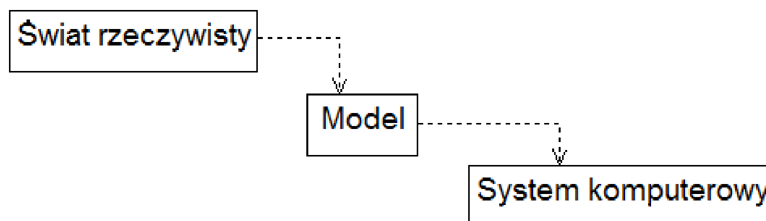
JĘZYK MODELOWANIA

1.1. Budowanie modeli 2
1.2. UML jako język modelowania 3
1.3. Zawartość UML 5
1.3.1. Diagramy klas 6
1.3.1.1. Deklaracja atrybutu w klasie 9
1.3.1.2. Deklaracja operacji w klasie 10
1.3.1.3. Typy operacji 10
1.3.1.4. Warunki wstępne i końcowe operacji 12
1.3.1.5. Klasy konkretne 12
1.3.1.6. Klasy polimorficzne 12
1.3.1.7. Klasy abstrakcyjne 12
1.3.1.8. Metaklasy 13
1.3.1.9. Klasy aktywne 13
1.3.2. Związki klas 14
1.3.2.1. Generalizacja 14
1.3.2.2. Zależność 15
1.3.2.3. Zależność klas zaprzyjaźnionych 17
1.3.2.4. Powiązanie 18
1.3.2.5. Klasa powiązania 20
1.3.2.6. Powiązania kwalifikowane 21
1.3.2.7. Agregacja prosta 23
1.3.2.8. Agregacja całkowita 25
1.3.2.9. Liczebności powiązań 27
1.3.2.10. Klasy zagnieżdżone 31
1.3.2.11. Klasy lokalne 33
1.3.3. Interfejsy 33
1.3.4. Struktury i typy wyliczeniowe 34
1.3.5. Wzorce klas 34
1.3.6. Diagramy obiektów 37
1.3.7. Diagramy stanów 42
1.3.8. Diagramy sekwencji 43
1.3.9. Diagramy komunikacji 45
1.3.10. Komponenty 46
1.3.11. Pakiety 47
1.3.12. Przypadki użycia 48
1.3.13. Diagramy czynności 50
1.3.14. Mechanizmy rozszerzania 52
Podsumowanie 53

1.1. Budowanie modeli

Zaprojektowanie i wdrożenie projektu rozwiązującego wybrany problem sprowadza się do poprawnego rozwiązania danego problemu z punktu widzenia teorii, którą się posługujemy, zaprojektowania architektury i abstrakcji programu w języku modelowania oraz stworzenia prawidłowej implementacji w języku programowania. Obecny rozdział stanowi krótkie wprowadzenie do zunifikowanego języka modelowania wizualnego, jakim jest UML (ang. *Unified Modeling Language*).

Model jest jednym z najważniejszych pojęć stosowanych w naukach przyrodniczych i technicznych. W naukach ścisłych poprzez model rozumiemy pewien zbiór ogólnych założeń, pojęć i zależności pozwalający w uproszczony sposób opisać wybrany aspekt rzeczywistości. Modelem jest również reprezentacja otaczającego świata w umyśle człowieka, której jednak nie należy mylić z rzeczywistością. Rysunek 1.1 w sposób schematyczny obrazuje zależności pomiędzy światem rzeczywistym, modelem i systemem komputerowym.



Rysunek 1.1. Zależności pomiędzy systemem komputerowym, modelem i światem rzeczywistym

Modele tworzone są głównie z dwóch powodów: dla lepszego zrozumienia dziedziny problemu oraz umożliwienia wymiany informacji w trakcie jego rozwiązywania przez zainteresowane osoby. Niewątpliwie znaczący nakład pracy oraz zasób czasu, jakie należy poświęcić w stworzenie odpowiedniego i poprawnego modelu przyszłościowo okazuje się być doskonałą inwestycją, jednakże pod warunkiem, iż model zostanie prawidłowo wykonany i w sposób rozsądny wykorzystany. Rosnący zakres oraz stopień komplikacji problemów, przed którymi staje współczesna nauka i inżynieria coraz częściej powodują konieczność stosowania przemyślanego podejścia do ich rozwiązywania. Okazuje się bowiem, iż w bardzo wielu przypadkach daleko niewystarczającym jest nieskomplikowane podejście polegające na próbach ciągłego zwiększenia mocy obliczeniowych, liczby zaangażowanych osób oraz nakładów finansowych poświęconych rozwiązywaniu danego problemu. Współczesna nauka dostrzega potrzebę zastosowanie bardziej efektywnych i subtelnych metod. Modelowanie systemów wydaje się być jedną z nich. Ponieważ poprawny opis dziedziny problemu jest zawsze podstawą znalezienia rozwiązania, kluczowym zagadnieniem jest ścisłe i odpowiadające rzeczywistości jego sformułowanie. Modelowanie w znakomity sposób pomaga zidentyfikować problem, określić skalę jego

złożoności, zaproponować rozwiązanie oraz, co jest szczególnie istotne, umożliwić przepływ informacji pomiędzy zainteresowanymi podmiotami. Współczesna nauka traktuje model jako podstawę komunikacji pomiędzy podmiotami zainteresowanymi danym problemem oraz zaangażowanymi w jego rozwiązanie.

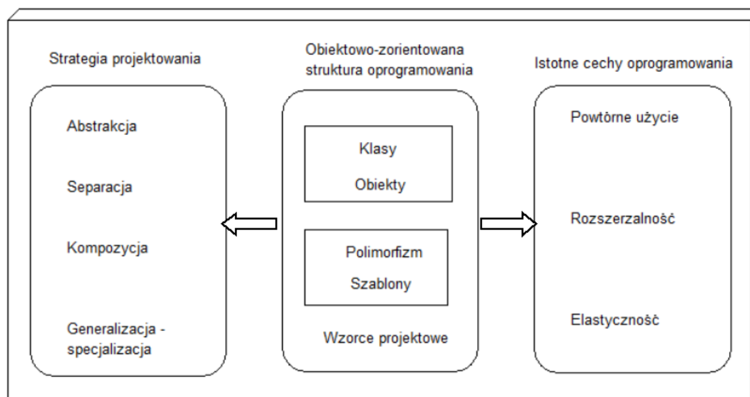
Jeśli chcemy zrozumieć dziedzinę problemu (opisać fragment istniejącej rzeczywistości) powinniśmy zbudować model dziedziny problemu. Celem tego modelu jest stworzenie poprawnej abstrakcji rzeczywistego świata. Taki abstrakcyjny model powinien być możliwie mało skomplikowany, ale mimo to powinien poprawnie odzwierciedlać świat rzeczywisty, tak aby na podstawie modelu można było przewidzieć zachowanie bytów w realnym świecie. System komputerowy to zbiór podsystemów sformowany w celu wykonania określonego zadania i opisany za pomocą pewnego zestawu modeli, z których każdy opisuje inny aspekt rzeczywistości.

1.2. UML jako język modelowania

Zunifikowany język modelowania UML został opracowany na bazie dobrych i złych doświadczeń środowisk zajmujących się analizą i projektowaniem zorientowanym obiektowo. Metody i notacje, wypracowane przez różne grupy do 1994 roku, współzawodniczyły ze sobą, kładąc przy tym duży nacisk na własną odrębność. James Rumbaugh opracował metodykę nazwaną Object Modeling Technique (OMT), która okazała się wystarczająca do modelowania dziedziny problemu [1]. Nie odzwierciedla jednak dostatecznie dokładnie zarówno wymagań użytkowników systemów, jak i wymagań implementacji. Ivar Jacobson rozwiną metodykę Object-Oriented System Engineering (OOSE) w sposób zadawalający uwzględniającą aspekty modelowania użytkowników i cyklu życia systemu jako całości [2,3]. Nie odzwierciedla ona jednak w sposób wystarczający sposobu modelowania dziedziny problemu oraz aspektów implementacji. Grady Booch jest autorem Object-Oriented Analysis and Design Methods (OOAD) spełniającej wszelkie wymogi w kwestii projektowania, konstrukcji i związków ze środowiskiem implementacji [4]. Metodyka ta nie uwzględniała jednak w sposób dostateczny fazy rozpoznania i analizy wymagań użytkowników. W ogólnym zamyśle opracowane metodyki miały wiele podobieństw zaś różniły się jedynie szczegółami. Powodowało to dowolność ujęcia podstawowych pojęć oraz uciążliwą różnorodność notacji stosowanych do opisu modeli. Z tego powodu wypracowanie i przyjęcie jednolitego standardu stało się koniecznością warunkującą dalszy rozwój metod projektowania systemów w ujęciu obiektowym. W połowie lat 90. ubiegłego wieku wymienieni wcześniej autorzy połączyli siły, publikując wstępną dokumentację UML [5]. W 2004 roku została oficjalnie zatwierdzona wersja UML 2.0 [6]. Najnowsza wersja UML 2.2 pojawiła się w lutym 2009 roku. We wciąż doskonalonej specyfikacji UML można znaleźć semantykę i syntaktykę dozwolonych konstrukcji językowych, które stanowią

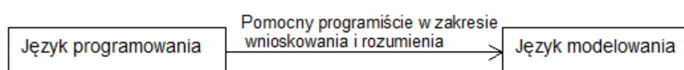
spójny, wewnętrznie niesprzeczny i kompletny opis, będący faktycznie modelem języka. Bardzo ważną cechą UML jest mechanizm rozszerzeń umożliwiający niemal swobodne redefiniowanie elementów języka. Dzięki temu UML jako zunifikowany język modelowania systemów informatycznych jest niezwykle elastyczny w dopasowywaniu go do pojawiających się nowych zastosowań [7].

UML jest językiem, za pomocą którego modelujemy rzeczywistość związaną z pewnymi obszarami informatyki dotyczącymi projektowania i opisywania systemów informatycznych. Głównym powodem, dla którego UML (z założenia niezależny od przyjętego języka programowania) zdobył uznanie i *de facto* stał się standardem w przemyśle informatycznym, jest to, że przy jego pomocy istnieje możliwość poprawnego i w miarę sprawnego wymodelowania zależności pomiędzy strategią tworzenia, obiektowo-zorientowaną strukturą oraz cechami, którymi powinno charakteryzować się współczesne oprogramowanie. Na rysunku 1.2 schematycznie pokazano te zależności.



Rysunek 1.2. Związki pomiędzy strategią tworzenia, obiektowo-zorientowaną strukturą oraz cechami współczesnego oprogramowania

Jak sama nazwa wskazuje, UML służy do budowania dobrych i ogólnych modeli i nie należy traktować go w sposób przesadnie ścisły. Pamiętajmy, że model służy jedynie do przybliżenia realnej rzeczywistości i nigdy całkowicie nie opisuje zachowania się bytów rzeczywiście istniejących. Rysunek 1.3 obrazuje związek pomiędzy językiem programowania a językiem modelowania. UML jest językiem modelowania obiektowego, jednak bardzo wiele jego elementów może być pomocnych podczas pracy nad prostszymi projektami, konstruowanymi w formie proceduralnej lub strukturalnej.



Rysunek 1.3. Związek pomiędzy językiem programowania a językiem modelowania

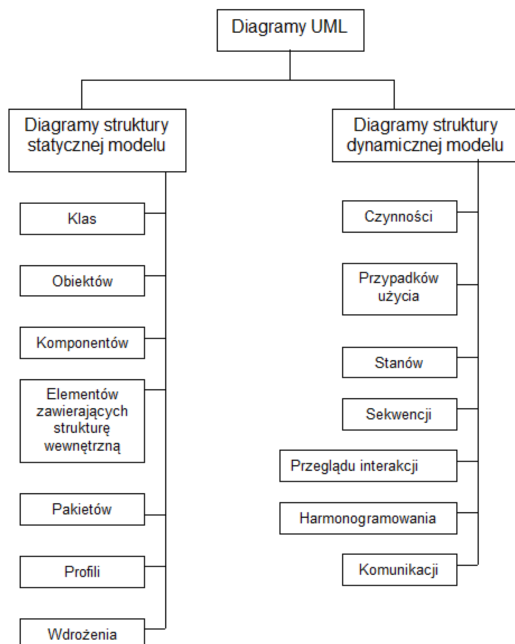
1.3. Zawartość UML

Zunifikowany język modelowania powstał na bazie wieloletnich doświadczeń analityków i projektantów. Jego głównym zadaniem jest definiowanie, konstruowanie, obrazowanie i dokumentowanie części składowych systemu komputerowego. UML zawiera zbiór symboli w postaci elementów graficznych, przy użyciu których można tworzyć bardziej złożone wyrażenia w postaci diagramów według bardziej lub mniej ściśle określonych reguł syntaktycznych. Celem diagramów jest zobrazowanie wielu perspektyw projektowanego systemu informatycznego. Zestaw takich perspektyw określa się mianem modelu, którego każdy element potrafimy opisać za pomocą skończenie wielu symboli.

Model systemu wyrażony w UML może zawierać pięć perspektyw obrazujących różne aspekty systemu. Są to:

- *perspektywa przypadków użycia* – opisuje zachowanie systemu z punku widzenia użytkowników, analityków oraz osób wykonujących testy. Zawiera zarówno elementy statyczne, opisane za pomocą diagramów przypadków użycia, jak i dynamiczne, opisane za pomocą diagramów przebiegu, kooperacji, stanów i czynności,
- *perspektywa projektowa* – uwzględnia opis klas, interfejsów, sekwencji i kooperacji, które razem składają się na opis danego problemu oraz jego rozwiązanie. Elementy statyczne wyrażane są za pomocą diagramów klas i obiektów, dynamiczne – za pomocą diagramów interakcji, stanów i czynności,
- *perspektywa procesowa* – odzwierciedla mechanizm kreowania wątków i procesów w systemie. Elementy statyczne i dynamiczne wyrażanie są za pomocą diagramów klas, obiektów, interakcji, stanów i czynności, przy czym główny nacisk kładzie się na klasy aktywne,
- *perspektywa implementacyjna* – opisuje komponenty i artefakty, użyte do scalenia i fizycznego udostępnienia systemu. Wiąże się ona z zarządzaniem konfiguracją poszczególnych wersji systemu. Elementy statyczne wyrażanie są za pomocą diagramów komponentów i artefaktów,
- *perspektywa wdrożeniowa* – opisuje węzły modelujące sprzęt, na którym system będzie uruchomiony. Wiąże się ona głównie z rozmieszczeniem, dostarczeniem i instalacją części systemu fizycznego. Aspekty statyczne wyrażane są za pomocą diagramów wdrożenia.

Każda z perspektyw przedstawia inny aspekt systemu, co w efekcie pozwala na pełny jego opis. Tworzeniu tych modeli, będących swego rodzaju przekrojami o specjalizowanym profilu informacyjnym, służy wiele rodzajów diagramów. UML 2.x (w zależności od stopnia szczegółowości stosowanego podziału) definiuje 14 podstawowych typów diagramów opisujących dwa aspekty modelowanego systemu, tak jak pokazano to na rysunku 1.4.



Rysunek 1.4. Podstawowe typy diagramów UML

Konstrukcja UML nie jest szczególnie skomplikowana, zaś diagramy nie są trudne w praktycznym użyciu i rozumieniu. Celem tego rozdziału nie jest szczegółowe przedstawienie całości języka modelowania. Warto pamiętać, iż na temat UML powstała ogromna liczba specjalistycznych publikacji, jednak w olbrzymiej ilości przypadków korzystamy z ograniczonego podzbioru tego języka, a właśnie ten podzbiór jest bardzo łatwy do praktycznego opanowania.

1.3.1. Diagramy klas

Klasa jest najbardziej podstawowym pojęciem obiektowych języków programowania. Definiuje nowy typ danych będący zbiorem stanów reprezentowanych przez wartości, jakie mogą przyjmować dane (pola) zdefiniowane w klasie oraz zbiorem funkcji dokonujących przejść między tymi stanami. Klasa jest abstrakcją obiektów z modelowanej dziedziny problemu. Stanowi wzorzec, opis, na bazie którego będą tworzone obiekty.

W konwencji UML dane zdefiniowane w klasie nazywamy atrybutami, zaś operujące na tych danych funkcje składowe określa się mianem operacji. UML rozróżnia pojęcia operacji i metody. Metoda jest zapisaną w języku programowania realną implementacją operacji (funkcji składowej) zdefiniowanej w modelu klasy. Metody pobierają dane stanu jako parametry, w których przekazywane są atrybuty i modyfikują je. Metody mogą wywoływać się nawzajem wyłącznie pośrednio, poprzez literały ich nazw. Metoda, reprezentuje czynność, jaka może być wykonana przez dany obiekt w świecie rzeczywistym.

W większości popularnych obiektowych językach programowania ogólny

zapis deklaracji klasy wygląda bardzo podobnie, np:

C++

```
class Nazwa Klasy {
   private:
    int atrybut; //lista atrybutów
   public:
    void operacja(); //lista operacji
};
```

C#

```
public class Nazwa Klasy {
  private object atrybut ; //lista atrybutów
  public void operacja(){ //lista operacji
  }
}
```

Java

```
public class Nazwa Klasy {
  private Object atrybut; //lista atrybutów
  public void operacja() { //lista operacji
  }
}
```

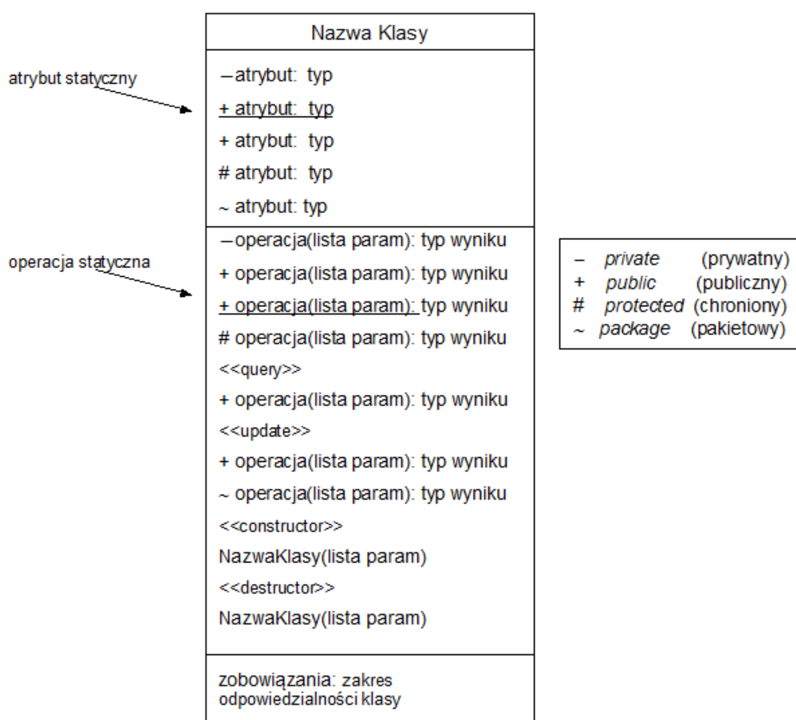
Object Pascal

```
type Nazwa Klasy = class
  private
     //lista atrybutów
  public
    //lista operacji
end;
```

W UML ikona klasy dzieli się na 3 główne obszary: obszar zawierający nazwę klasy, obszar zawierający listę atrybutów oraz obszar z listą operacji. Każda klasa ma przypisaną unikalną nazwę, wyróżniającą ją spośród innych klas. Nazwa klasy może być zapisana w formie prostej lub poprzedzona nazwą pakietu (tzw. forma ścieżkowa). Opcjonalnie ikonę klasy można uzupełnić o czwarty obszar, w którym opisywane są zobowiązania (ang. *class responsibilities*), czyli to, za co w konstruowanym modelu klasa jest odpowiedzialna.

Na ikonie klasy opcjonalnie można też opisać typy operacji. Głównymi typami operacji są: konstruktor klasy, który w zależności od implementacji klasy oznaczany jest stereotypem <<constructor>>, <<ctor>> lub <<create>>, destruktor oznaczany stereotypem <<destructor>>,

<<dtor>> lub <<destroy>>, operacja <<update>> zdolna zmienić stan obiektu, który w przyszłości zostanie stworzony na bazie klasy, oraz operacja <<query>> nie zmieniająca stanu obiektu. Na rysunku 1.5 pokazano ogólny diagram klasy, w której zostały zdefiniowane omówione elementy wraz z określeniem specyfikatorów dostępu (poziomów widoczności elementów składowych klasy).



Rysunek 1.5. Ogólna zawartość ikony klasy UML. W ramce podano oznaczenia poziomów widoczności elementów składowych klasy

Wszystkie składowe klasy są widoczne wewnątrz klasy, ale mogą mieć różny poziom dostępności z zewnątrz, który określany jest za pomocą jednego ze znaków: +, -, # lub ~ odpowiadających słowom kluczowym: public, private, protected, package. Pod względem dostępności składowe dzielą się na:

- publiczne +, których nazwy mogą być używane we wszystkich miejscach programu, gdzie widoczna jest definicja klasy,
- prywatne – , których nazwy mogą być używane tylko przez operacje, które same są składowymi tej samej klasy, lub funkcje z daną klasą zaprzyjaźnione (C++),
- chronione # , których nazwy mogą być używane tylko przez operacje, które same są składowymi tej samej klasy, funkcje z daną klasą zaprzyjaźnione

(C++), funkcje składowe i zaprzyjaźnione klas dziedziczących po danej klasie bazowej (C++),

- pakietowe ~, których nazwy mogą być używane w obrębie pakietu, w którym znajduje się definicja klasy (Java).

Na diagramach klas poziom dostępności określa się dla każdej ze składowych osobno (podobnie jak w Javie i C#).

W trakcie konstruowania logicznego modelu systemu warto pamiętać, iż mimo że istnieje możliwość deklarowania atrybutów (zmiennych) publicznych w klasie, to jednak zalecane jest, aby dążyć do tego, by ich deklaracje były umieszczane w sekcji prywatnej, dostęp zaś do nich był możliwy poprzez operacje (funkcje) z sekcji publicznej. Tego typu technika programowania nosi nazwę enkapsulacji danych, co oznacza, że dostęp do prywatnych danych jest zawsze ściśle kontrolowany. Enkapsulacja (ang. *encapsulation*) jest mechanizmem wiążącym instrukcje z danymi i zabezpieczającym je przed ingerencją z zewnątrz i błędnym użyciem.

Zdefiniowana w standardzie UML ikona klasy nie zawiera rozszerzeń właściwych różnym implementacjom obiektowych języków programowania. Modelując klasę pod kątem konkretnej implementacji zawsze można użyć predefiniowanych rozszerzeń w postaci stereotypów, czyli nazw zapisanych w nawiasach francuskich << >>.

Rysunek 1.5 zawiera dokładny opis elementów składowych klasy, jednak w praktyce bardzo często (zwłaszcza w trakcie budowania wstępnego modelu dziedziny problemu) stosuje się notację uproszczoną, którą w wielu miejscach tej książki będziemy się posługiwać. Notacja uproszczona może np. zawierać jedynie nazwę klasy lub nazwę oraz wyliczenie atrybutów i operacji bez szczegółowego podawania odpowiednio ich typów i argumentów.

1.3.1.1. Deklaracja atrybutu w klasie

Najczęściej atrybut jest opisywany tylko przez dwa elementy: nazwę i typ (patrz rysunek 1.5). Pełna definicja może uwzględniać: widoczność atrybutu, definiującą, z jakich miejsc systemu atrybut jest dostępny, liczebność, która określa ile obiektów mieści się w atrybucie, ograniczenia nałożone na wartość atrybutu oraz wartość domyślną:

```
[widoczność] nazwaAtrybutu : typ[liczebność]
             {ograniczenia} = wartość_domyślna
```

Najczęściej stosowane ograniczenia określające właściwości atrybutu, to:
`{ordered}` – obiekty reprezentowane przez atrybut są uporządkowane,
`{unordered}` – obiekty są nieuporządkowane,
`{unique}` – obiekty nie powtarzają się,
`{nonunique}` – obiekty mogą się powtarzać,

`{readOnly}` – wartość atrybutu przeznaczona jest tylko do odczytu,
`{frozen}` – wartość atrybutu nie może być zmodyfikowana po jej przypisaniu.

1.3.1.2. Deklaracja operacji w klasie

W modelu klasy ogólna postać deklaracji funkcji składowych (operacji) ma następującą postać:

```
[widoczność] nazwaOperacji [(lista_parametrów)]
                          [:typ_wyniku] [właściwość]
```

gdzie lista parametrów operacji:

```
[tryb] nazwa : typ [ = wartość_domyślna]
```

tryb:
`in` – parametr wejściowy, nie może być modyfikowany,
`out` – parametr wyjściowy, może być modyfikowany,
`inout` – parametr wejściowy, wyjściowy, może być modyfikowany

właściwość:
`leaf` – funkcja niepolimorficzna (nie może być redefiniowana),
`isQuerry` – funkcja nie zmieniająca stanu obiektu,
`sequental`, `guarded`, `concurent` (mają zastosowanie przy operacjach współbieżnych).

1.3.1.3. Typy operacji

Operacje zdefiniowane w klasie mogą być klasyfikowane na różne sposoby. Najczęściej stosowana klasyfikacja rozróżnia następujące typy operacji:

- *Konstruktor* – jest szczególnym rodzajem operacji, która wykorzystywana jest w momencie tworzenia obiektu. Zadaniem konstruktora jest wykonywanie odpowiednich czynności podczas tworzenia obiektu, zatem realizuje on postulat odpowiedzialności obiektu za samego siebie. W ciele konstruktora zwykle inicjuje się atrybuty obiektu, nadaje im domyślne wartości, określa relacje w stosunku do innych obiektów. We wszystkich językach programowania obiektowego w momencie tworzenia obiektu wyszukiwany jest jego konstruktor, a następnie automatycznie wykonywany. W różnych obiektowych językach programowania w różny sposób oznacza się konstruktor, np.: w C++, Javie i C# – jest to metoda o nazwie zgodnej z nazwą klasy, w Object Pascalu – metoda której nazwę poprzedzono słowem kluczowym `constructor`. Języki programowania mogą udostępniać specjalny rodzaj konstruktorów zwany konstruktorami domyślnymi. Konstruktor domyślny nie wykonuje żadnej czynności. Istnieje wyłącznie po

to, aby użytkownik mógł bezpiecznie wywołać operator `new` z dowolnego konstruktora i w ten sposób utworzyć anonimowy obiekt dowolnej klasy. Implementując konstruktor w klasie potomnej, należy zawsze wywoływać konstruktor klasy nadrzędnej.

- *Destruktor* – operacja wywoływana przez program przed usunięciem obiektu z pamięci. W językach których składnia wzorowana jest na C++, destruktor ma taką samą nazwę jak klasa, poprzedzoną znakiem tyldy ~ (dla odróżnienia od konstruktora). Destruktora nie należy wywoływać w sposób jawny (chociaż są od tej reguły pewne wyjątki). Istnieją kompilatory, np. Java, posiadające wbudowany mechanizm automatycznego zwalniania nieużywanych obszarów pamięci operacyjnej (ang. *garbage collection*). W kompilatorach nie zaopatrzonych w ten mechanizmu, np. C++ i Object Pascal, w celu wywołania destruktora klasy przed zakończeniem programu należy użyć odpowiednio operatora `delete` (z właściwym wskaźnikiem) lub funkcji `Free()` z właściwym argumentem.
- *Mutator* – operacja zmieniająca stan jednego lub więcej atrybutu klasy (tzw. operacja *set* lub *setter*).
- *Akcesor* – operacja odczytująca aktualną wartość (stan) atrybutu klasy bez jego modyfikacji (tzw. operacja *get* lub *getter*). W C++ metody klasy mogą być zdeklarowane jako metody stałe. Oznacza to, iż dana metoda nie będzie modyfikować stanu obiektu, na rzecz którego została wywołana, czyli nie zmienia wartości żadnej jego składowej. Metodę jako stałą deklaruje się umieszczając słowo kluczowe `const` tuż za nawiasem zamykającym listę parametrów funkcji.
- *Iterator* – operacja pozwalająca na kolejne przetwarzanie wszystkich elementów struktury danych.
- *Finalizator* – w niektórych językach z wbudowanym mechanizm automatycznego zwalniania nieużywanych obszarów pamięci (np. Java i C#) dostępna jest składnia finalizatora – specjalnej metody wywoływanej, gdy obiekt jest usuwany przy porządkowaniu zasobów pamięci. Jednak w przeciwieństwie do destruktora, programista nie kontroluje, w którym dokładnie momencie działania programu to nastąpi.
- *Operacja statyczna* – niekiedy zachodzi potrzeba dodania do klasy elementu, który będzie związany z klasą, ale nie z konkretnym jej obiektem. Wówczas deklaruje się elementy statyczne klasy. Element statyczny jest elementem, który jest powiązany z klasą, a nie z obiektem tej klasy, czyli np. statyczna operacja nie może się odwołać do niestatycznego atrybutu lub funkcji składowej. Do statycznego elementu klasy (np. operacji) można się odwołać nawet jeżeli nie został stworzony żaden obiekt klasy.
- *Własność* – własność podobna jest do atrybutu w klasie, ale może zachowywać się jak operacja. Własności pełnią rolę akscesorów i mutatorów. W niektórych implementacjach C++ właściwość posiada mechanizmy odczytu (read) i zapisu (write), służące do pobierania i ustawiania wartości właściwości. Mechanizmem odczytu może być nazwa atrybutu lub operacja zwra-

cająca wartość właściwości. Mechanizmem zapisu może być nazwa atrybutu lub operacja ustawiająca wartość atrybutu. Zaniedbując mechanizm zapisywania, tworzy się właściwość tylko do odczytu. Dopuszczalne jest również stworzenie właściwości tylko do zapisu, jednak celowość takiego tworu jest bardzo wątpliwa. Większość mechanizmów zapisujących i odczytujących stanowi nazwy atrybutów lub operacji (tak jak w C#), chociaż mogą to być również części atrybutów agregujących, takich jak struktury lub tablice.

1.3.1.4. Warunki wstępne i końcowe operacji

Każdą operację można dodatkowo opisywać podając dwa rodzaje warunków: wstępne (ang. *preconditions*) oraz końcowe (ang. *postconditions*). Modelują one wymagany i oczekiwany stan systemu (lub jego fragmentu) odpowiednio przed i po wykonaniu określonej operacji. Tego typu notacja pozwala na opisanie zadania realizowanego przez operację, jej wymagań oraz rezultatów jej wykonania. Osoba modelująca system ma możliwość wyrażenia poprzez nie warunków, które powinny być spełnione w celu poprawnego wykonania zadania przez operację.

1.3.1.5. Klasy konkretne

Klasa konkretne to klasa, w której wszystkie wywoływane operacje zostały zaimplementowane.

1.3.1.6. Klasy polimorficzne

Klasy, w których zdefiniowano jedną lub więcej operacji wirtualnych, nazywamy klasami polimorficznymi. Operacje wirtualne mają bardzo ciekawą właściwość. Charakteryzują się tym, iż podczas wywoływania dowolnej z nich za pomocą odwołania (referencji) lub wskaźnika do klasy bazowej wskazującego na egzemplarz klasy pochodnej, aktualna wersja wywoływanej operacji każdorazowo ustalana jest w trakcie wykonywania programu z rozróżnieniem typu wskazywanej klasy. W UML nazwę operacji wirtualnej oznaczamy kursywą (w ikonie klasy).

Konstruując model, który będzie implementowany w Javie warto pamiętać, iż w języku tym wszystkie operacje traktowane są jako wirtualne, za wyjątkiem:

• operacji statycznych (nie dotyczących obiektów),

• operacji ze specyfikatorem `final` (postać operacji jest ostateczna i nie może być redefiniowywana w klasach pochodnych) ,

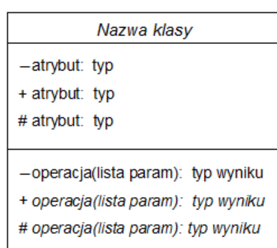
• operacji prywatnych (dla których odwołania w innych operacjach danej klasy nie są polimorficzne).

1.3.1.7. Klasy abstrakcyjne

W odróżnieniu od klas konkretnych, klasy abstrakcyjne są tworami, w których występuje jedna lub większa liczba funkcji czysto wirtualnych (w terminologii stosowanej w Javie, C# i Object Pascalu odpowiednikiem funkcji czysto

wirtualnych C++ są metody abstrakcyjne). Funkcja czysto wirtualna jest zapisana w klasie abstrakcyjnej, jednak nie posiada tam swojej implementacji (mówimy, że funkcje takie pozostają niezdefiniowane w swojej klasie). Klasy abstrakcyjne występują w roli klas bazowych. Implementacja funkcji czysto wirtualnej zawsze zapisana jest w którejś z klas dziedziczących.

W UML nazwę klasy abstrakcyjnej piszemy kursywą. Drukiem pochyłym oznacza się nazwy operacji czysto wirtualnych, tak jak schematycznie pokazano to na rysunku 1.6. W modelu, klasy abstrakcyjne służą jedynie do reprezentowania pewnych pojęć będących generalizacją (uogólnieniem) innych pojęć występujących w dziedzinie problemu, ale same w sobie nie mogą reprezentować jakichkolwiek obiektów.



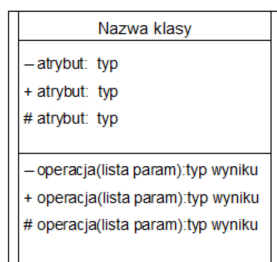
Rysunek 1.6. Oznaczenie klasy abstrakcyjnej, w której zadeklarowano dwie funkcje czysto wirtualne

1.3.1.8. Metaklasy

Metaklasa to klasa dysponująca specjalnym zestawem metod odpowiadającym podstawowym operacjom klasy, takim jak: sposób tworzenia obiektów, destrukcja obiektów, wywoływanie metod, stosowanie mechanizmów dziedziczenia, przypisywanie wartości atrybutom, uzyskiwanie dostępu do atrybutów, mechanizmy posługiwania się wskaźnikami `this` (`self`).

1.3.1.9. Klasy aktywne

Klasy (a dokładniej stworzone na ich bazie obiekty) mogą być w projektowanym systemie źródłem nowego procesu lub wątków. Klasy takie określa się mianem aktywnych i oznacza tak, jak pokazano na rysunku 1.7.



Rysunek 1.7. Oznaczenie klasy aktywnej

1.3.2. Związki klas

Definicja klasy składa się z trzech głównych obszarów (nazwy klasy, listy atrybutów oraz listy operacji), zatem łatwo wywnioskować jest iż klasy mogą występować w trzech głównych związkach strukturalnych.

1.3.2.1. Generalizacja

Generalizacja jest często utożsamiana z „dziedziczeniem," lecz istnieje pomiędzy tymi pojęciami znacząca różnica. Generalizacja opisuje relację w modelu dziedziny, zaś dziedziczenie jest programową implementacją generalizacji – jest tym jak przedstawiamy generalizację w kodzie. Przeciwną stroną generalizacji jest specjalizacja. Klasa pochodna z rysunku 1.8 jest wyspecjalizowaną formą klasy bazowej, zaś klasa bazowa jest generalną formą dla jednej lub większej liczby swoich klas potomnych.

Generalizacja jest związkiem między klasą bazową (ang. *base class*), zwaną przodkiem, a specyficznym jej rodzajem, zwanym potomkiem, klasą potomną lub pochodną (ang. *derived class*). Klasa potomna dziedziczy strukturę i zachowanie po klasie bazowej.

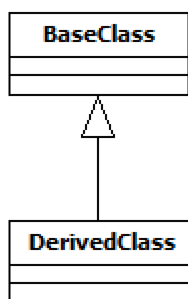
Generalizacja jest procesem przejmowania przez jeden obiekt właściwości innego umożliwiając tym samym klasyfikowania obiektów. W UML opis klasy najbardziej ogólnej (będącej na szczycie hierarchii generalizacji i niemającej przodków) często uzupełnia się, dodając pod jej nazwą słowo `{root}`. Klasy takie określa się mianem klas-korzeni. Opis klasy, która nie może mieć podklas, często uzupełnia się słowem `{leaf}`. Klasy takie w zależności od implementacji określa się mianem klas-liści lub klas finalnych. Klasy finalne definiuje się w celu zapewnienia bezpieczeństwa. Definicja klasy finalnej nie może ulegać dalszym zmianom poprzez specjalizację gwarantując takie jej zachowanie jak w pierwotnej postaci. W Javie cecha finalności może przysługiwać nie całej klasie, ale niektórym jej składowym. W przypadku atrybutów oznacza to, iż atrybut jest stałą. Operacja finalna to taka, która nie może być redefiniowana w klasach potomnych.

W UML ogólną postać klas pozostających w związku generalizacji-specjalizacji przedstawiamy w sposób pokazany na rysunku 1.8. Generalizację oznacza się linią ciągłą zakończoną niewypełnionym grotem skierowanym w kierunku klasy bazowej (klasy pojęciowo niezależnej).

Dziedziczenie (ang. *inheritance*) jest jednym z najważniejszych mechanizmów programowania zorientowanego obiektowo. Pozwala na przekazywanie elementów klasy bazowej klasom pochodnym. Oznacza to, że w prosty sposób można zbudować pewną hierarchię klas uporządkowaną od najbardziej ogólnej do najbardziej szczegółowej. Implementacja modelu z rysunku 1.8 realizowana jest poprzez nagłówki klas i np. w języku C++ ma następującą postać:

```
class DerivedClass: public BaseClass {
 // atrybuty (dane)  i operacje (funkcje składowe)
}
```

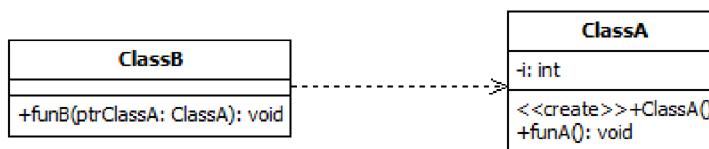
Domyślnie, dziedziczenie implementowane jest poprzez publiczny poziom dostępu.



Rysunek 1.8. Oznaczenie związku generalizacji-specjalizacji klas

1.3.2.2. Zależność

Modelowanie związku klas poprzez odpowiednie konstrukcje ich funkcji składowych nazywamy zależnością. Na diagramie UML zależność oznacza się linią przerywaną zakończoną strzałką zwróconą w kierunku klasy niezależnej, tak jak pokazano to na rysunku 1.9. W przykładzie z listingu 1.1 zaprezentowano implementację diagramu 1.9 wykonaną w języku C++.



Rysunek 1.9. Statyczny diagram klas zależnych. Klasa A jest klasą funkcjonalnie niezależną

Listing 1.1. Przykładowa implementacja diagramu klas zależnych

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

class ClassA { //klasa niezależna
   private:
     int i;
     //lista atrybutów;
   public:
     ClassA() { i = 10;} //konstruktor klasy A
     void funA() {cout <<"operacja klasy A\n";
                  i *= 10;
                  cout << i << endl;}
} ;
//-----------------------------------------------------------
```

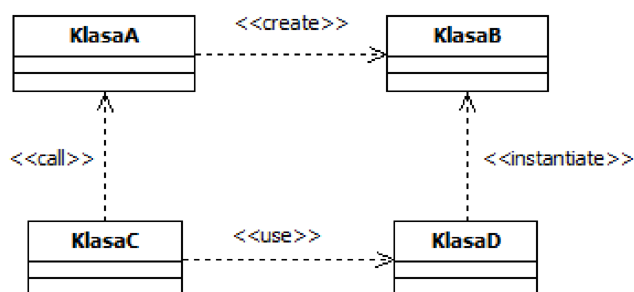
```
class ClassB { //klasa zależna
   private:
     //lista atrybutów;
   public:
      void funB(ClassA *ptrClassA){
        ptrClassA->funA();
      }
} ;
//------------------------------------------------------------
int main()
{
  ClassA classA;
  ClassB *ptrClassB;
  ptrClassB->funB(&classA);
  cin.get();
  return 0;
}
```

Zależność jest związkiem strukturalnym wskazującym, że obiekty jednego typu są funkcyjnie zależne od obiektów innego typu. Ogólnie powiemy, iż z zależnością klas mamy do czynienia wówczas, gdy:

- na liście argumentów funkcji składowej klasy występuje nazwa innej klasy,
- na liście argumentów funkcji składowej klasy występuje wskaźnik do innej klasy,
- na liście argumentów funkcji składowej klasy występuje odwołanie (referencja) do innej klasy,
- w ciele funkcji składowej klasy występuje zmienna, typu innej klasy,
- w ciele funkcji składowej klasy występuje wskaźnik do innej klasy,
- w ciele funkcji składowej klasy występuje odwołanie (referencja) do innej klasy,
- typem powrotnym funkcji składowej klasy jest wskaźnik do innej klasy.

Zależności są najprostszym i najsłabszym rodzajem relacji łączących klasy i jak wszystkie związki klas są bytami pojęciowymi. Jeżeli w trakcie konstruowania modelu zachodzi potrzeba dokładniejszego określenia roli stosowanej zależności zawsze można użyć właściwego stereotypu, tak jak pokazano to na rysunku 1.10. Do najczęściej wykorzystywanych należą:

`<<cal >>` – operacje w klasie C wywołują operacje z klasy A,
`<<create>>` – klasa A tworzy egzemplarz klasy B,
`<<instantiate>>` – obiekt klasy D jest egzemplarzem (instancją) klasy B,
`<<use>>` – do zaimplementowania klasy C wymagana jest klasa D.



Rysunek 1.10. Najczęściej wykorzystywane typy zależności

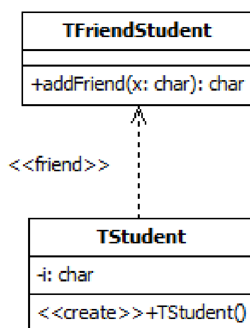
1.3.2.3. Zależność klas zaprzyjaźnionych

Istnieją sytuacje, w których funkcje nienależące do danej klasy mogą uzyskiwać dostęp do jej elementów prywatnych. Sytuację taką można programowo zaimplementować w C++, pod warunkiem, że klasa zostanie zadeklarowana jako zaprzyjaźniona. Listing 1.2 ilustruje tę sytuację. Na rysunku 1.11 przedstawiono statyczny diagram klas ze stereotypem wskazującym, że zależność reprezentuje zaprzyjaźnienie. Oczywiście w tym modelu `TFriendStudent` jest klasą niezależną.

Listing 1.2. Przykładowa implementacja diagramu z rysunku 1.11

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

class TStudent{
   private:
     char* i;
     friend class TFriendStudent;
   public:
     TStudent() {strcpy(i, "Janek Kowalski");}
};
//-----------------------------------------------------------
class TFriendStudent {
  public:
    char* addFriend(char* x) {
       TStudent student;
       return strncat(student.i,  x, 18);
    }
};
//-----------------------------------------------------------
int main()
{
  TFriendStudent friendStudent;
```

```
  cout << friendStudent.addFriend(" i Bartek Jankowski");
  cin.get();
  return 0;
}
```



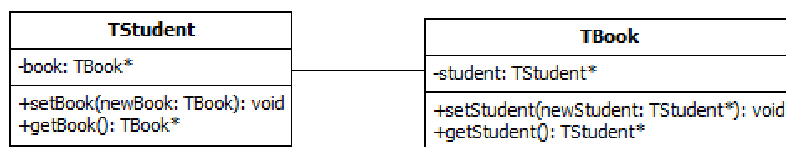
Rysunek 1.11. Zależność klas zaprzyjaźnionych

1.3.2.4. Powiązanie

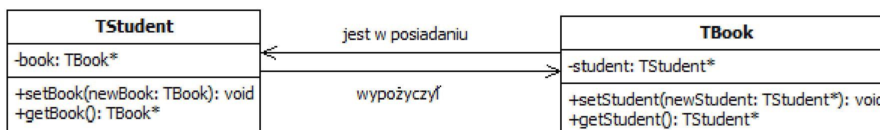
Powiązanie (ang. *association*) jest kolejnym ważnym mechanizmem stosowanym podczas modelowania i konstruowania programów zorientowanych obiektowo. W odróżnieniu od dziedziczenia, powiązania zapewniają niekiedy większą elastyczność budowanego modelu, a co za tym idzie – również tworzonego kodu, jawnie wskazując na dwukierunkowe zależności powiązanych elementów. Jeżeli w przypadku każdej pary klas elementy jednej z nich mogą modyfikować zachowanie elementów drugiej i na odwrót, mamy do czynienia z powiązaniem.

Powiązanie jest związkiem strukturalnym wskazującym, że obiekty jednego typu są połączone z obiektami innego typu poprzez atrybuty danych klas. Zwykłe powiązanie między dwoma klasami wskazuje, że można przejść z obiektu jednej z tych klas do obiektu drugiej i na odwrót. Można jednak jawnie wskazać kierunek nawigacji. W UML powiązanie z kierunkiem oznacza się linią ciągłą zakończoną otwartym grotem skierowanym w stronę elementu logicznie niezależnego. W implementacji C++ powiązanie realizuje się poprzez deklarację na liście atrybutów jednej klasy zmiennej typu lub wskaźnika do innej klasy.

Zazwyczaj powiązanie występujące między dwiema klasami umożliwia przechodzenie (w obie strony) od obiektów jednej klasy do obiektów drugiej klasy. Jeżeli wyraźnie nie zaznaczono tego na diagramie, nawigacja wzdłuż powiązania jest zawsze dwukierunkowa.



Rysunek 1.12. Przykład dwukierunkowego powiązania klas



Rysunek 1.13. Jawne oznaczenie dwukierunkowych powiązań wraz z ich nazwami

Fragment kodu z listingu 1.3 jest odzwierciedleniem sytuacji z rysunków 1.12 oraz 1.13, w której zastosowano dwukierunkową nawigację powiązań. Klasa `TStudent` poprzez prywatny wskaźnik `book` jest powiązana z klasą `TBook` (książka), zaś klasa `TBook` poprzez prywatny wskaźnik `student` jest powiązana z klasą `TStudent` wskazujące, że każdy (np. wypożyczony z biblioteki) egzemplarz książki może znajdować się w posiadaniu konkretnej osoby. Mając dostęp do książki, można odnaleźć studenta, który ją wypożyczył, a także znając nazwisko studenta, można określić, z jakich książek aktualnie korzysta.

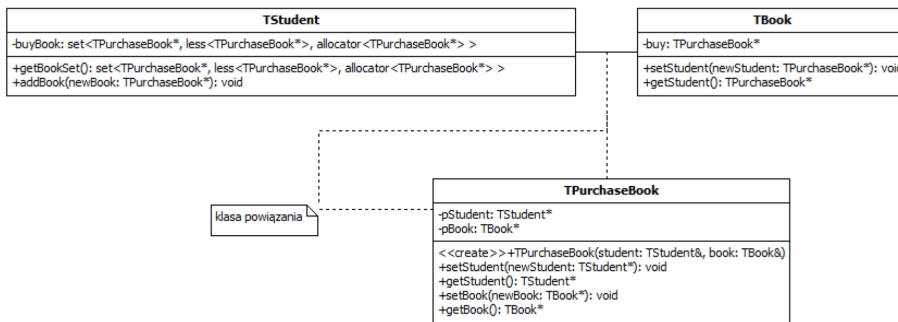
Listing 1.3. Implementacja dwukierunkowego powiązania klas

```
class TBook;
class TStudent {
  public:
    void setBook(TBook* newBook);
    const TBook* getBook() const;
    //...
  private:
    //...
    TBook* book; //dowiązanie anonimowego
                 //obiektu klasy TBook
};
//-----------------------------------------------------
class TBook {
  public:
    void setStudent(TStudent* newStudent);
    const TStudent* getStudent() const;
    //...
  private:
    //...
```

```
    TStudent* student; //dowiązanie anonimowego
                       //obiektu klasy TStudent
};
```

1.3.2.5. Klasa powiązania

Klasa powiązania będąc formą przejściową pomiędzy powiązaniem a klasą pozwala na dodawanie właściwości do powiązania. Bez trudu można wyobrazić sobie typy powiązań, które same w sobie posiadają właściwości. Rozważmy związek studenta z zakupioną przez niego książką. Pewną nieścisłością jest uwzględnienie istnienia związku opartego na powiązaniu studenta z zakupem książki oraz zakupu książki ze studentem. Zakup książki (ang. *purchase of book*) reprezentuje właściwości tego związku i odnosi się dokładnie do jednej pary student-książka, tak jak pokazano na rysunku 1.14 oraz przykładzie z listingu 1.4.



Rysunek 1.14. Model klasy powiązania

Klasy powiązania graficznie reprezentowane są jako klasy połączone linią przerywaną z relacją powiązania, której dotyczą.

Listing 1.4. Implementacja klasy powiązania

```
class TStudent;
class TBook;
class TPurchaseBook {
  public:
    TPurchaseBook(TStudent &student, TBook &book);
    //...
    void setStudent(TStudent* newStudent);
    const TStudent* getStudent() const;
    void setBook(TBook* newBook);
    const TBook* getBook() const;
  private:
    //...
    TStudent* pStudent;
```

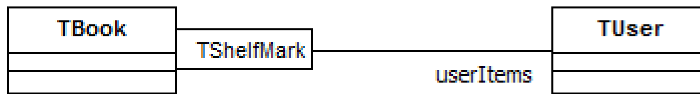
```
    TBook* pBook;
};
//--------------------------------------------------------
class TBook {
  public:
    //...
    void setStudent(TPurchaseBook* newStudent);
    const TPurchaseBook* getStudent() const;
  private:
    //...
    TPurchaseBook* buy;
};
//---------------------------------------------------------
class TStudent {
  public:
    //...
    const set<TPurchaseBook*, less<TPurchaseBook*>,
              allocator<TPurchaseBook*>> &getBookSet()
const;
    void addBook(TPurchaseBook* newBook);
  private:
    //...
    set<TPurchaseBook*, less<TPurchaseBook*>,
        allocator<TPurchaseBook*>> buyBook;
};
```

W UML dostępnych jest sześć podstawowych dodatków do powiązań: nazwa, rola, liczebności przy każdym końcu związku, nawigacja, kwalifikacja oraz różne rodzaje agregacji, w tym również agregacje z nawigacją.

1.3.2.6. Powiązania kwalifikowane

Powiązanie kwalifikowane (ang. *qualified association*) może być traktowane jako dodatek do zwykłego powiązania uzupełniając je o możliwość określenia, który z atrybutów powiązania decyduje o związku między klasami (jest jego kwalifikatorem), tak jak pokazano to na rysunku 1.15. Rozpatrzmy przykład, w którym osoba będąca aktualnie użytkownikiem systemu rezerwacji chcąc z biblioteki wypożyczyć książkę podaje m.in. jej sygnaturę (ang. *shelf mark*) otrzymując listę aktualnie dostępnych egzemplarzy. Oznacza to, iż między użytkownikiem systemu rezerwacji a książką występuje relacja typu jeden do wielu. Jednak w danym momencie użytkownik może zarezerwować tylko jeden egzemplarz konkretnej książki, co w konsekwencji powoduje, iż `TShelfMark` staje kwalifikatorem (kluczem) tej relacji. W efekcie pomiędzy użytkownikiem a egzemplarzem klasy `TBook` występuje relacja typu jeden do jednego, ponieważ

konkretny użytkownik rezerwuje konkretny egzemplarz książki w danym momencie tylko raz.



Rysunek 1.15. Przykład powiązania kwalifikowanego

Powiązania kwalifikowane mogą też być odpowiednikami struktur słownikowych (takich jak: `map`, `hash_map`, `multimap`, `hash_multimap`) z biblioteki standardowej C++ wykorzystujących definicję pary (klucza i wartości):

```
template <class T1, class T2>
  struct pair
  {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair (const T1& a, const T2& b) : first(a), second(b) {}
    pair ()
#ifndef _RWSTD_NO_BUILT_IN_CTOR
      : first(T1()), second(T2())
#endif
    { ; }
    pair(const pair& p)
      : first(p.first), second(p.second)
    { ; }
#ifndef _RWSTD_NO_MEMBER_TEMPLATES
    template <class U, class V> pair(const pair<U,V>& p)
      : first(p.first), second(p.second)
    { ; }
#endif
```

W tego typu strukturach określonemu kluczowi przypisujemy pewną wartość lub wartości, tak jak pokazano to w przykładzie z listingu 1.5.

Listing 1.5. Jeden z możliwych sposobów implementowania powiązania kwalifikowanego

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

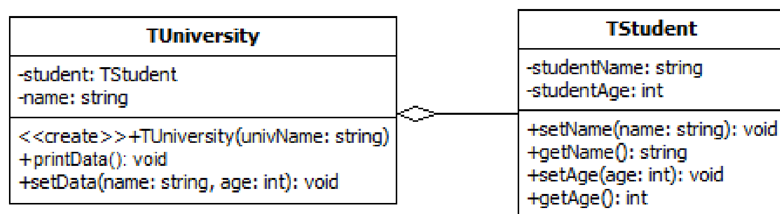
```
class TUser
{
 //...
};
//--------------------------------------------------
class TShelfMark
{
 //...
};
//--------------------------------------------------
class TBook
{
  private:
     map <TUser, TShelfMark> userItems;
  public:
     const TUser getUserItem(TShelfMark shelfMark) const
                 {/*...*/};
     void setUserItem(unsigned int amount,
                      TShelfMark forShelfMark) {/*...*/};
};
```

1.3.2.7. Agregacja prosta

Powiązanie dwóch klas jest związkiem strukturalnym elementów równorzędnych, podkreślającym sytuację, w której powiązane klasy znajdują się na tym samym poziomie pojęciowym. Czasami jednak zachodzi potrzeba zapisania związku rodzaju „całość-część”. W takiej relacji jedna klasa reprezentuje większy element stanowiący całość, zaś druga reprezentuje elementy mniejsze, czyli części, z których składa się całość. Agregacja jest szczególnym rodzajem powiązania. Agregacja będąc bytem pojęciowym umożliwia oddzielenie całości od części.

Rozważmy związek studenta i uczelni, na której studiuje. Student może studiować przynajmniej na jednej uczelni nie wpływając na sposób jej funkcjonowania. Związek studenta z uczelnią jest typowym przykładem agregacji. Agregacja jest pojęciem o szerokim znaczeniu. Umożliwia odróżnienie całości od części, nie wpływa na kierunek nawigacji pomiędzy częściami systemu, ani nie wiąże czasu życia części z czasem życia całości systemu. Agregację prostą oznaczamy za pomocą rombu po stronie całości, tak jak pokazano to na rysunku 1.16, gdzie klasa `TUniversity` jest klasą agregującą, zaś `TStudent` – klasą agregowaną.

Relacja uczelnia-student jest przykładem agregacji prostej. Student może jednocześnie studiować na kilku uczelniach, może je również zmieniać. Czas życia studenta i uczelni w żaden sposób nie powinien być ze sobą powiązany. Student może opuścić uczelnię i fakt ten nie powinien w żaden sposób wpływać na jej funkcjonowanie.



Rysunek 1.16. Agregacja prosta

Listing 1.6. Kod odpowiadający modelowi z rysunku 1.16

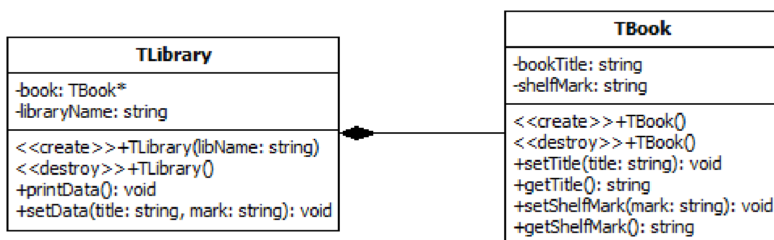
```
#include <iostream>
#include <cstring>
using namespace std;

class TStudent
 {
  private:
    string studentName;
    int studentAge;
  public:
    void setName(const string& name) {studentName = name;}
    string getName() const  {return studentName;}
    void setAge(const int age) {studentAge = age;}
    int getAge() const {return studentAge;}
 };
//-----------------------------------------------------------
class TUniversity {
  private:
    TStudent student; //proste zagregowanie klasy TStudent
                      //z klasą TUniversity
    string name;
  public:
    TUniversity(const string& univName){
      name = univName;
      cout << name <<endl;
    }
    void printData() {
       cout << student.getName() << endl;
       cout <<student.getAge() <<endl;
    }
    void setData(string name, int age){
     student.setName(name);
     student.setAge(age);
   }
```

```
};
//--------------------------------------------------------
main() {
  TUniversity *ptrUniversity = \
  new TUniversity("Maria Curie Sklodowska University");
  ptrUniversity->setData("Jan Kowalski", 21);
  ptrUniversity->printData();
  delete ptrUniversity;
  cin.get();
  return 0;
}
```

1.3.2.8. Agregacja całkowita

Agregacja całkowita (kompozycja) to związek charakteryzujący się relacją wyłącznej własności oraz jednością czasu życia całości i części. Proces konstrukcji całości musi być poprzedzony skonstruowaniem wszystkich jej elementów składowych (części nie mogą powstawać po utworzeniu całości, ale w momencie gdy destrukcji podlega całość muszą zginąć wszystkie jej części). W przypadku agregacji prostej część może zależeć od kilku całości, zaś w przypadku kompozycji tylko od jednej. Agregacja całkowita może istnieć np. pomiędzy biblioteką i książką, którą zaopatrzono w unikalną sygnaturę.

Agregację całkowitą oznaczamy symbolem diamentu (wypełnionym rombem) umieszczonym po stronie całości. Na rysunku 1.17 pokazano statyczny model fragmentu systemu, gdzie zobrazowano, iż książka o podanej sygnaturze może należeć tylko do konkretnej biblioteki. Przykład z listingu 1.7 pokazuje uproszczoną implementację diagramu 1.17. Klasa `TLibrary` (biblioteka) całkowicie agreguje klasę `TBook` dzięki odpowiedniemu zaimplementowaniu konstruktora `TLibrary(...)` i destruktora `~TLibrary()`. Obiekt klasy `TBook` tworzony jest w konstruktorze klasy `TLibrary` oraz niszczony jest w momencie wywołania jej destruktora.



Rysunek 1.17. Kompozycja

Listing 1.7. Implementacja statycznego modelu klas z rysunku 1.17

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

```
class TBook
 {
  private:
    string bookTitle;
    string shelfMark;
  public:
    TBook()  {cout <<"Konstruktor klasy TBook\n";}
    ~TBook() { cout <<"Destruktor klasy TBook\n";}
    void setTitle(const string& title) {bookTitle = title;}
    string getTitle() const  {return bookTitle;}
    void setShelfMark(const string& mark) {
                        shelfMark = mark;}
    string getShelfMark() const {return shelfMark;}
 };
//-----------------------------------------------------------
class TLibrary {
  private:
    TBook* book; //całkowite zagregowanie klasy TBook
                 //z klasą TLibrary
    string libraryName;
  public:
    TLibrary(const string& libName){
       libraryName = libName;
       cout << libName <<endl;
       book = new TBook; }
    ~TLibrary() { delete book;}
    void printData() {
       cout << book->getTitle() << "\t";
       cout << book->getShelfMark() <<endl;
    }
    void setData(string title, string mark){
       book->setTitle(title);
       book->setShelfMark(mark);
   }
};
//-----------------------------------------------------------
main() {
  cout << "Wywołuje konstruktor klasy TLibrary\n";
  TLibrary *library = new TLibrary("UMCS Library");
  library->setData("Programowanie Obiektowe",
                   "sygnatura U 203I");
  library->printData();
  cout << "Wywołuje destruktor klasy TLibrary\n";
  delete library;
```

```
  cin.get();
  return 0;
}
```

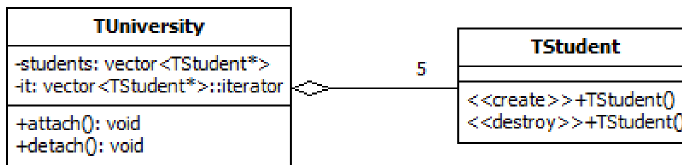
1.3.2.9. Liczebności powiązań

Analizując przykłady z rysunków 1.12-17 można zauważyć, iż modelowały one sytuacje, w których np. student mógł być w posiadaniu tylko jednego podręcznika, na uczelni mogła studiować tylko jedna osoba, zaś w bibliotece znajdował się tylko jeden egzemplarz książki. Ze względu na prostotę obrazowania, tego typu przykłady doskonale nadają się do wstępnej analizy projektowanego systemu nie odzwierciedlając jednak w pełni jego złożoności.

W realnym świecie student może wypożyczyć kilka książek, na uczelni studiuje wiele osób, zaś w zasobach biblioteki uniwersyteckiej z reguły znajduje się bardzo wiele egzemplarzy rozmaitych podręczników. W celu urealnienia omawianych poprzednio przykładów skorzystamy z pojęcia liczebności będącego bardzo ważnym dodatkiem do powiązań.

W praktyce, w trakcie modelowania różnych odmian powiązań zachodzi potrzeba podania liczebności, czyli liczby (krotności) obiektów jaka może być przyłączona przez jeden egzemplarz powiązania. W UML liczebność zapisywana jest w postaci wyrażenia, którego wartością jest dobrze określony przedział liczbowy lub pojedyncza liczba. Podając liczebność przy jednym końcu powiązania wskazujemy ile obiektów jednej klasy powinno być połączonych z każdym obiektem klasy znajdującej się na drugim końcu powiązania. Liczebność można ustalić poprzez wyszczególnienie żądanej liczby, np.: 5, przedziału liczbowego 1..5 (jeden lub pięć), dowolnie wiele (0..*) albo co najmniej dwa (2..*).

Na diagramie z rysunku 1.18 wymodelowano sytuację, w której na uniwersytecie studiuje określona liczba studentów. W celu łatwości testowania ograniczono się tutaj do liczebności 5 osób (chociaż w rzeczywistości liczebność osób pobierających naukę może być znacznie większa). Związek agregacji klasy `TStudent` z klasą `TUniversity` zaimplementowany został poprzez zadeklarowanie w jednym z prywatnych atrybutów `TUniversity` klasy kontenerowej `vector` przechowującej wskaźniki do klasy `TStudent`. Klasy, których zadaniem jest realizowanie agregacji poprzez grupowanie innych obiektów w postaci list, kolekcji itd. nazywane są klasami kontenerowymi, a obiekty tych klas – kontenerami. Klasy kontenerowe dostarczają metod służących do operowania na kolekcjach obiektów składowych (dodawanie, usuwanie, zmiana kolejności itp.) oraz iteratorów wskazujących na poszczególne elementy składowane w kontenerze. Na rysunku 1.18 widzimy, iż w klasie `TUniversity` zadeklarowano prywatny iterator `it`, który wskazywać będzie na poszczególne elementy składowane w kontenerze `students`.



Rysunek 1.18. Agregacja prosta z dokładnie wyszczególnioną liczebnością związku

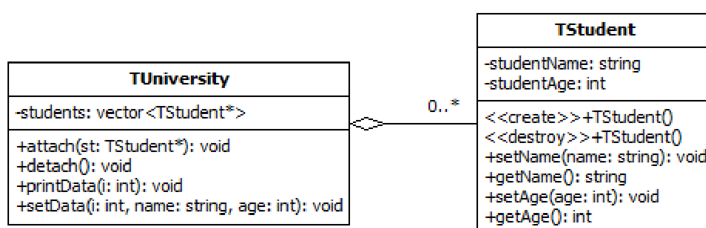
W publicznej operacji `attach()` klasy `TUniversity` tworzona jest tablica określonej liczby obiektów klasy `TStudent`. Każdy z tych obiektów umieszczany jest w kontenerze `students` dzięki wykorzystaniu funkcji składowej `push_back()` klasy kontenerowej `vector`. Programista może w dowolnym czasie zagregować z obiektem klasy `TUniversity` żądaną liczbę obiektów klasy `TStudent`. Również w dowolnym momencie za pomocą operacji `detach()` można usunąć uprzednio zagregowane z klasą `TUniversity` obiekty klasy `TStudent`. Cykl życia obiektu stworzonego na bazie klasy `TUniversity` nie jest w żaden sposób związany z cyklem życia tablicy obiektów reprezentujących klasę `TStudent`, tak jak zaimplementowano to w przykładzie z listingu 1.8.

Listing 1.8. Jedna z możliwych implementacji w C++ diagramu 1.18

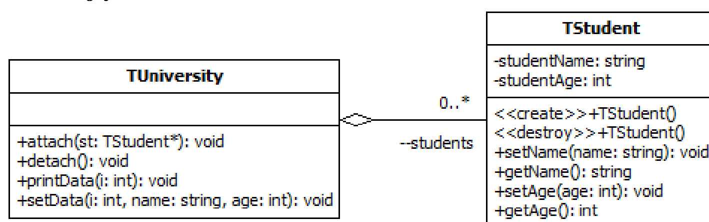
```
#include <vector>
#include <iostream>
using namespace std;

class TStudent
{
  public:
    TStudent(){cout << "konstruktor TStudent\n";}
    ~TStudent(){cout << "destruktor TStudent" << endl; }
};
class TUniversity {
  private:
    vector<TStudent*> students;
    vector<TStudent*>::iterator it;
  public:
    void attach() {
      TStudent* stud = new TStudent[5];
      students.push_back(stud);
    }
    void detach() {
      for (it=students.begin(); it!=students.end(); ++it)
      delete [] *it;
    }
```

```
};
//------------------------------------------------------------------
int main()
{
   TUniversity *ptrUniversity = new TUniversity;
   ptrUniversity->attach();
   ptrUniversity->detach();
   delete ptrUniversity;

   cin.get();
   return 0;
}
```

Fragment modelu systemu zaprezentowany na diagramie 1.18 można jeszcze bardziej przybliżyć do rzeczywistości umożliwiając osobie zarządzającej systemem dołączanie obiektów klasy `TStudent` do obiektu klasy `TUniversity` w czasie działania programu nie nakładając przy tym sztywnych ograniczeń na liczebność związku. Na rysunkach 1.19 oraz 1.20 zaprezentowano sytuację, w której nie ustalono górnego zakresu liczby studentów mogących studiować na uniwersytecie. Oba diagramy są w zasadzie tożsame, różnica polega na tym, iż na diagramie z rysunku 1.19 w sposób szczegółowy zasugerowano implementację powiązania 0..* poprzez elementy prywatnego kontenera `students` (patrz przykład 1.9). Oczywiście nie jest to jedyne możliwe rozwiązanie. W przypadku, gdy nie chcemy jawnie sugerować implementacji modelu można ograniczyć się do oznaczeń pokazanych na rysunku 1.20.



Rysunek 1.19. Agregacja prosta z liczebnością 0..* (dowolnie wiele) z sugerowaną implementacją



Rysunek 1.20. Agregacja prosta z liczebnością 0..* bez sugerowanej implementacji

Listing 1.9. Jedna z możliwych implementacji modelu z diagramu 1.19

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

class TStudent
{
  public:
    TStudent(){cout << "konstruktor TStudent\n\n";}
    ~TStudent(){cout << "destruktor TStudent\n";}
    void setName(const string& name) {studentName = name;}
    string getName() const  {return studentName;}
    void setAge(const int age) {studentAge = age;}
    int getAge() const {return studentAge;}
   private:
    string studentName;
    int studentAge;
};
//-----------------------------------------------------------
class TUniversity {
  private:
    vector<TStudent*> students;
  public:
    void attach(TStudent *st){students.push_back(st);}
    void detach(){ students.pop_back();
         cout << "\nPo usunięciu elementów,
                   vector TStudent "
         <<(students.empty()?"jest":"nie jest")<<" pusty\n";
    }
    void printData(int i) {cout << students[i]->getName()
                          << endl;
         cout <<students[i]->getAge() <<endl;
    }
    void setData(int i, string name, int age){
         students[i]->setName(name);
         students[i]->setAge(age);
    }
};
//-----------------------------------------------------------
int main()
{
   TStudent *ptrStudent = new TStudent();
   TUniversity *ptrUniversity = new TUniversity;
```

```
   ptrUniversity->attach(ptrStudent); //dołączanie obiektu
                                      //klasy TStudent
   ptrUniversity ->setData(0, "Wojtek",24);
   ptrUniversity ->printData(0);

   ptrUniversity ->attach(ptrStudent); //dołączanie obiektu
                                       //klasy TStudent
   ptrUniversity ->setData(1, "Jola",19);
   ptrUniversity ->printData(1);

   ptrUniversity->detach();   //odłączanie zagregowanych
                              // obiektów
   ptrUniversity->detach();   //klasy TStudent

   delete ptrStudent;
   delete ptrUniversity;
   cin.get();
   return 0;
}
```

1.3.2.10. Klasy zagnieżdżone

Oprócz dziedziczenia, zależności i różnych odmian powiązań relacje między klasami i ich wystąpieniami mogą być tworzone poprzez mechanizm zagnieżdżania polegający na definiowaniu jednej klasy, nazywanej *klasą zagnieżdżoną*, w obrębie innej nazywanej *klasą zawierającą* (otaczającą, zagnieżdżającą). Klasa zagnieżdżona nazywana jest klasą wewnętrzną. Klasy otaczająca i zagnieżdżona nie są specjalnie ze sobą związane, a co za tym idzie, w UML nie posiadają specjalnej reprezentacji. W C++, w odróżnieniu od Javy, obowiązują dla nich normalne zasady dostępności. Na listingu z przykładu 1.10 pokazano sytuację, w której wewnątrz klasy `TStudent` zdefiniowana jest klasa `TLibraryAccount` (konto biblioteczne). Jej operacja `printInfo()` jest w klasie zadeklarowana, ale zdefiniowana poza nią. Do zdefiniowania tego rodzaju operacji należy użyć podwójnej kwalifikacji: funkcja `printInfo()` należy do zakresu klasy `TLibraryAccount`, który z kolei zawarty jest w zakresie klasy `TStudent`.

Listing 1.10. Implementacja klas zagnieżdżonych

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

class TStudent {
  private:
    char*  studentName;
```

```
    int    has;
  public:
    class TLibraryAccount { //klasa zagnieżdżona
      private:
        int numberOfBooks;
      public:
        TLibraryAccount(int numberOfBooks)
        :numberOfBooks(numberOfBooks){}
        void printInfo();
    };

    TStudent(const char* name)
    :studentName(strcpy(new char[strlen(studentName)+1],
                 name)),
     has(0){
      cout << studentName;
    }
    TStudent& bookLent(int b)  {has += b; return *this; }
    TLibraryAccount* printAccount();
    ~TStudent() { delete [] studentName; }
};
//-----------------------------------------------------------
TStudent::TLibraryAccount* TStudent::printAccount() {
   return new TLibraryAccount(has);
}
//-----------------------------------------------------------
void TStudent::TLibraryAccount::printInfo() {
   cout <<": liczba wypozyczonych ksiazek: "
   << numberOfBooks << endl;
}
//-----------------------------------------------------------
int main() {
   TStudent* student = new TStudent("Jan Kowalski");
   student->bookLent(3).bookLent(1);
   TStudent::TLibraryAccount* libAccount =
                              student->printAccount();
   libAccount->printInfo();
   delete libAccount;
   delete student;
   cin.get();
   return 0;
}
```

Każde wystąpienie klasy zewnętrznej tworzy wystąpienia klas wewnętrznych. Do klasy zagnieżdżonej można odwoływać się w obrębie klasy zagnież-

dżającej oraz poza nią za pomocą odwołania kwalifikowanego.

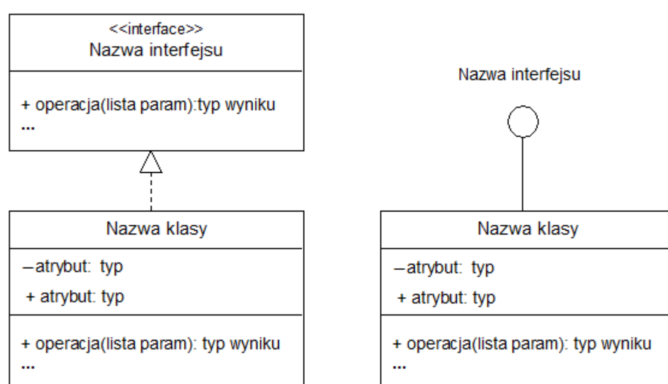
1.3.2.11. Klasy lokalne

Klasa zdefiniowana w ciele operacji innej klasy nazywana jest klasą lokalną. Klasy lokalne nie są dostępne poza operacjami w których zostały zdefiniowane. W UML nie posiadają swojej reprezentacji.

1.3.3. Interfejsy

Niekiedy zachodzi potrzeba określenia zestawu operacji oferowanych przez klasę lub komponent bez określania ich struktury oraz implementacji. Temu celowi służą interfejsy. Istnieją dwa główne rodzaje interfejsów: komponentowy i nakładkowy. Interfejs komponentowy jest podobny do klasy abstrakcyjnej całkowicie pozbawionej atrybutów i zawierającej wyłącznie publiczne funkcje czysto wirtualne (abstrakcyjne). Interfejs nakładkowy jest podobny do interfejsu komponentowego. Różnica polega na tym, iż w tym przypadku nie wszystkie zadeklarowane operacje muszą być czysto wirtualne.

Interfejs na diagramie wyróżnia się stereotypem `<<interface>>`. Operacje zdefiniowane w interfejsie powinny być zaimplementowane w klasach realizujących interfejs, co w praktyce oznacza, iż realizacja interfejsu w klasie polega na zdefiniowaniu w tej klasie wszystkich operacji zadeklarowanych w implementowanym interfejsie. Podstawowy związek interfejsu z klasą implementującą jego funkcje nazywamy realizacją i możemy zamiennie oznaczać dwoma równoważnymi symbolami, z których pierwszy jest bardzo podobny do symbolu dziedziczenia, a drugi stanowi jego uproszczoną postać, tak jak pokazano na rysunku 1.21. Należy nadmienić, iż w celu podkreślenia tego związku używa się niekiedy stereotypu `<<realize>>`, jednak nie jest to wymaganą konwencją.



Rysunek 1.21. Równoważne przedstawienia realizacji interfejsu przez klasę. Interfejs zawiera zestaw operacji, które wyznaczają zakres usług oferowanych przez klasę

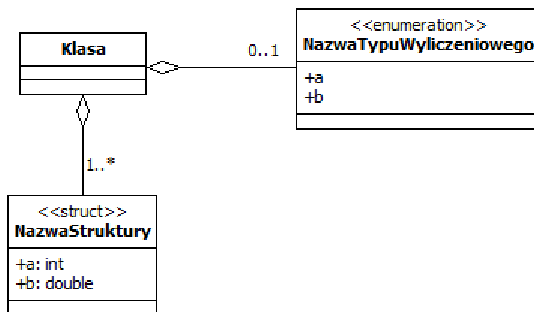
W większości obiektowych języków programowania (za wyjątkiem C++) nie jest dopuszczalne stosowanie mechanizmu wielodziedziczenia (dziedziczenia wielobazowego). Oznacza to, iż klasa może dziedziczyć jedynie po pojedynczej klasie bazowej. W przypadku realizacji interfejsów nie ma tego ograniczenia. Klasa może realizować wiele interfejsów. Programowo, realizację interfejsu implementuje się podobnie jak dziedziczenie klas. W odróżnieniu od takich języków obiektowych jak Java, C# czy Object Pascal, interfejsy C++ *emulowane* są za pomocą klas abstrakcyjnych całkowicie pozbawionych atrybutów. W niektórych implementacjach C++ w trakcie pisania interfejsu można użyć słowa `interface`, należy jednak mieć świadomość faktu, iż będzie to jedynie makrodefinicja (wprowadzona w celu zachowania zgodności z językiem IDL), której użycie wymaga włączenia odpowiedniego pliku nagłówkowego.

Warto też pamiętać, iż struktura interfejsu może być różnie implementowana przez różne języki programowania, np. w C# i Object Pascalu elementami interfejsu (oprócz publicznych metod abstrakcyjnych) mogą być własności i zdarzenia. Z kolei Java dopuszcza występowanie w interfejsie publicznych statycznych zmiennych finalnych (stałych) o ustalonych typach i wartościach.

Według konwencji stosowanej w modelu COM, nazwy interfejsów rozpoczynają się od litery `I`. W dalszej części podręcznika autor będzie konsekwentnie stosował tego rodzaju konwencję zapisu.

1.3.4. Struktury i typy wyliczeniowe

Częściami składowymi modelu mogą być takie elementy, jak struktury i typy wyliczeniowe. Elementy te z reguły będą agregować z klasami, do których przekazują swoją zawartość w postaci parametrów. Na rysunku 1.22 pokazano możliwe oznaczenia agregującej struktury oraz typu wyliczeniowego.



Rysunek 1.22. Stereotypowe oznaczenie struktury i typu wyliczeniowego

1.3.5. Wzorce klas

Wzorce klas (określane też mianem klas parametryzowanych lub szablonów klas) uogólniają algorytmy wykonywane na różnych typach danych. W C++ szkielet definicji wzorca klasy można przedstawić następująco:

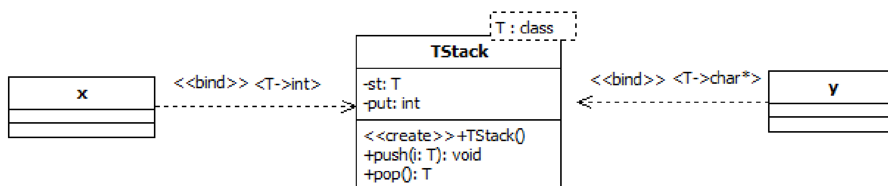
```
template < class TypDanych, ... >
class NazwaKlasy
{
  //atrybuty;
  //operacje();
};
```

Wzorzec klasy posiada szczeliny dla typów danych (prostych typów danych, klas, wskaźników) będących jego parametrami uogólnionymi. Wzorca nie można użyć w sposób bezpośredni, najpierw należy utworzyć jego egzemplarz. Czynność tę określa się jako dowiązanie parametru aktualnego do każdego parametru formalnego wzorca:

```
NazwaKlasy<TypDanych> EgzemplarzKlasy;
```

Od tego momentu egzemplarz wzorca traktowany jest jak konkretny obiekt, który można dowolnie wykorzystywać. Najczęściej wzorców klas używa się do definiowania tablic dynamicznych (kontenerów, pojemników), zdolnych do przechowywania i wykonywania logicznie uogólnionego zestawu operacji na danych różnych typów. Przykład z listingu 1.11 przedstawia implementację prostego kontenera zdolnego składować elementy różnych typów. Na rysunku 1.23 przedstawiono diagram omawianego wzorca klasy.

W UML wzorce klas modelujemy tak samo, jak zwykłe klasy, ale z dodatkowym prostokątem o brzegach narysowanych linią przerywaną, w którym podaje się parametry wzorca. Wykorzystany na diagramie stereotyp `<<bind>>` wskazuje, że element źródłowy tworzy egzemplarz wzorca docelowego z użyciem danych parametrów aktualnych.



Rysunek 1.23. UML-owy diagram wzorca `TStack`

Listing 1.11. Implementacja diagramu 1.23

```
#include <iostream>
using namespace std;
const size = 5;

template <class T>
class TStack {
  private:
```

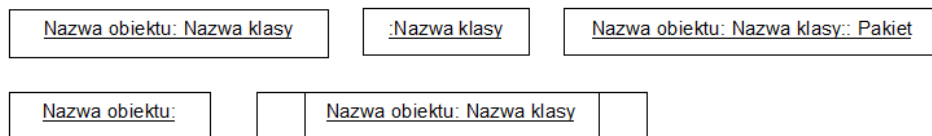
```
    T st[size];
    int put;
  public:
    TStack();
    void push(T i);
    T pop();
};
//-----------------------------------------------------------
template <class T> TStack<T>::TStack() //konstruktor
{
  put = 0;
  cout << "Utworzono stos danych\n\n";
};
//-----------------------------------------------------------
template <class T> void TStack<T>::push(T i)
{
  if (put >= size) {
    cout << "Zapełniono stos\n";
    return;
  }
  else {
    st[put] = i;
    put++;
  }
};
//-----------------------------------------------------------
template <class T> T TStack<T>::pop()
{
  if (put == 0) {
    cout << "\n\nBrak elementów na stosie\n";
    return 0;
  }
  else {
    put--;
    return st[put];
  }
};
//-----------------------------------------------------------
int main()
{
  TStack<int> x;
  TStack<char *> y;
  x.push(2010);
  cout << x.pop();
  y.push(" lipiec, ");
```

```
  cout << y.pop();
  y.push("czwartek ");
  cout << y.pop();
  x.push(15);
  cout << x.pop();
  cout << x.pop();
  cin.get();
  return 0;
}
```

1.3.6. Diagramy obiektów

Realny świat ma strukturę obiektową. Prawie wszystkie otaczające nas rzeczy można opisać jako konkretne wystąpienia bytów, dla których wcześniej w taki czy inny sposób został stworzony określony przepis. Klasa jest właśnie takim przepisem, który istnieje w umyśle programisty. Klasa jest przepisem obrazującym sposób łączenia danych i instrukcji (instrukcje wykonują swoje operacje na danych). Obiekt jest konkretnym wystąpieniem (egzemplarzem) klasy. Jednocześnie może istnieć wiele obiektów tej samej klasy. Obiekt hermetyzuje utworzoną specjalnie dla niego komórkę pamięci, co w praktyce oznacza, iż obiekt istnieje do czasu, gdy wykonywany program utraci wszystkie odwołania do niego.

W UML ikonę obiektu można przedstawić na wiele sposobów. Rysunek 1.24 pokazuje pięć najczęściej spotykanych oznaczeń.



Rysunek 1.24. Przedstawienie (w kolejności od lewej strony) obiektu nazwanego, anonimowego, zawierającego ścieżkę dostępu do pakietu z definicją klasy, obiektu osieroconego oraz obiektu aktywnego

W Javie, C# i Object Pascalu obiekty klas tworzone są zawsze na stercie. Obiekty są zawsze anonimowe, zaś dostęp do nich uzyskiwany jest wyłącznie poprzez referencje (odwołania).

Obiekty C++ mogą być tworzone na stosie (wtedy obowiązuje dla nich semantyka wartości) lub na stercie za pomocą operatora `new`. W tym drugim przypadku obiekty są anonimowe, a dostęp do nich uzyskiwany jest poprzez wskaźniki.

Wszystkie obiekty powinny być jednoznacznie nazwane lub pozostać anonimowe. Nazwa obiektu jest zakończona dwukropkiem, po którym jest umieszczona nazwa typu. Cała nazwa, razem z typem, jest podkreślona. Obiekty anonimowe nie posiadają nazwy, jednak mają takie same zasady nadawania nazw

jak klasy. Nazwa obiektu może być nazwą prostą, lub kwalifikowaną z nazwą ścieżki do pakietu, w którym zapisana jest definicja klasy.

Obiekty aktywne inicjują działania innych obiektów. Każdy aktywny obiekt jest implementacyjnie odwzorowywany jako osobny wątek lub proces w systemie. Uruchomiony obiekt powstaje, gdy wątek wywołuje pewien obiekt. Uruchomiony obiekt istnieje do czasu zakończenia jego wykonywania przez wątek. Jednocześnie może istnieć wiele wywołań tego samego obiektu.

Obiekt można scharakteryzować poprzez opisanie jego stanu, czyli podanie sekwencji wartości przypisanych konkretnym atrybutom klasy, na bazie której obiekt został stworzony. Rysunki 1.25 i 1.26 pokazują odpowiednio ikony nazwanego obiektu `student` oraz anonimowego obiektu klasy `TStudent` z konkretnymi wartościami przypisanymi do odpowiednich atrybutów klasy. Jeden ze sposobów tworzenia nazwanego obiektu `student` oraz obiektu anonimowego klasy `TStudent` zaprezentowany został na listingu 1.12.

Listing 1.12. Statyczne oraz dynamiczne tworzenie obiektu

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

class TStudent
 {
   public:
      ~TStudent() {cout <<"destruktor klasy TStudent\n";};
      TStudent(const string& name, const int age);
      TStudent(){cout <<"konstruktor domyślny klasy
                         TStudent\n";};
      void setName(const string& name);
      string getName() const;
      void setAge(const int age);
      int getAge() const;
   private:
      string studentName;
      int studentAge;
 };
//-----------------------------------------------------------
TStudent::TStudent(const string& name, const int age)
  :studentName(name), studentAge(age) {
  cout << "konstruktor klasy TStudent\n";
}
//-----------------------------------------------------------
void TStudent::setName(const string& name) {
  studentName = name;
}
//-----------------------------------------------------------
```

```
string TStudent::getName() const {
  return studentName;
}
//--------------------------------------------------------
void TStudent::setAge(const int age)  {
  studentAge = age;
}
//--------------------------------------------------------
int TStudent::getAge() const {
  return studentAge;
}
//--------------------------------------------------------
main() {
  //tworzenie obiektu nazwanego student klasy TStudent
  TStudent student = TStudent("Jan Kowalski",21);
  cout << student.getName() << endl;
  cout << student.getAge() << endl;

  //tworzenie obiektu anonimowego klasy TStudent
  TStudent *ptrStudent = new TStudent("Jola Nowak",20);
  cout << ptrStudent->getName() << endl;
  cout << ptrStudent->getAge() << endl;
  delete ptrStudent;
  cin.get();
  return 0;
}
```

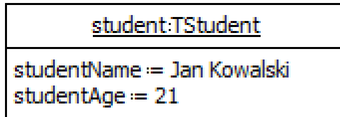
W powyższym programie definiowano klasę `TStudent` z dwoma prywatnymi atrybutami przechowującymi nazwisko oraz wiek studenta. Klasa ta ma dwuparametrowy konstruktor. Należy zauważyć, iż w takim wypadu żaden konstruktor bezparametrowy (domyślny) nie jest generowany automatycznie. Jeżeli zatem chcemy, aby taki konstruktor istniał, to należy zdefiniować go samodzielnie.

Przykład zamieszczony na listingu 1.12 obrazuje sytuację, w której w programie głównym (w funkcji `main()`) w pierwszej kolejności zdefiniowano nazwany obiekt `student` typu `TStudent` z wykorzystaniem dwuargumentowego konstruktora klasy:

```
TStudent student = TStudent("Jan Kowalski",21);
```

Warto zwrócić uwagę, iż `student` jest nazwą obiektu, a nie wskaźnikiem lub referencją. Ze względu na fakt, iż obiekt jest konkretnym wystąpieniem klasy, powinien posiadać wartości przypisane jej atrybutom, gdyż w przeciwieństwie do klasy, obiekt charakteryzuje się opisując jego stan. Stan obiektu jest

zawsze reprezentowany poprzez pokazanie wartości atrybutów w danym momencie działania programu, ponieważ wartości atrybutów są znane tylko i wyłącznie w czasie jego wykonywania. Typy atrybutów są pomijane, z tego powodu, iż klasa na bazie której obiekt został wykreowany zawiera wszystkie niezbędne informacje o typach (patrz przykład 1.12).

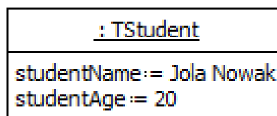


Rysunek 1.25. Opisane atrybuty nazwanego obiektu klasy z przykładu 1.12

Na listingu z przykładu 1.12 pokazano również jeden ze sposobów tworzenia obiektu anonimowego, za pomocą wskaźnika `ptrStudent` do klasy `TStudent` i operatora `new`:

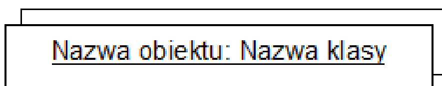
```
TStudent *ptrStudent = new TStudent("Jola Nowak", 20);
```

Ponieważ operator ten zwraca adres utworzonego obiektu, zatem `ptrStudent` jest jedynie nazwą wskaźnika, a nie nazwą utworzonego obiektu, tak jak pokazano to na rysunku 1.26.



Rysunek 1.26 Opisane atrybuty anonimowego obiektu klasy z przykładu 1.12

Na bazie jednej klasy można stworzyć wiele obiektów. Na rysunku 1.27 pokazano oznaczenie nazwanych (konkretnych) obiektów wielokrotnych.



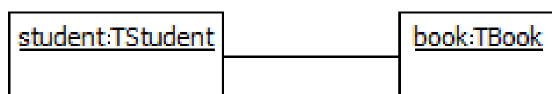
Rysunek 1.27. Obiekty wielokrotne stworzone na bazie jednej klasy

Listing 1.13 obrazuje jeden z możliwych sposobów tworzenia odpowiednio nazwanych oraz anonimowych obiektów wielokrotnych klasy `TStudent`. Jeśli obiekty wielokrotne (np. w postaci tablicy obiektów) tworzymy na stercie (z wykorzystaniem operatora `new`), należy pamiętać, iż nie ma możliwości indywidualnego inicjalizowania elementów tablicy za pomocą konstruktora z parametrami (patrz przykład z listingu 1.12); wszystkie zostaną utworzone za pomocą konstruktora domyślnego, który zawsze powinien istnieć.

Listing 1.13. Przykładowa implementacja nazwanych oraz anonimowych obiektów wielokrotnych

```
main() {
  cout<< "Tworzenie nazwanych obiektów wielokrotnych\n";
  TStudent students[3]={ TStudent("Jan Kowalski", 21),
                         TStudent("Bartek Jankowski", 22),
                         TStudent("Jola Jankowska", 20),
                       };
  for (int i = 0; i < 3; i++)
    cout << students[i].getName() << " lat "
    << students[i].getAge()  << endl;

  cout<< "Tworzenie anonimowych obiektów wielokrotnych\n";
  TStudent *ptrStudents[3]={ new TStudent(),
                             new TStudent(),
                             new TStudent(),
                           };
  ptrStudents[0]->setName("Jan Kowalski");
  ptrStudents[0]->setAge(21);
  ptrStudents[1]->setName("Bartek Jankowski");
  ptrStudents[1]->setAge(22);
  ptrStudents[2]->setName("Jola Jankowska");
  ptrStudents[2]->setAge(20);

  for (int i = 0; i < 3; i++)
    cout << ptrStudents[i]->getName() << " lat "
    << ptrStudents[i]->getAge()  << endl;

  for (int i = 0; i < 3; i++)
    delete ptrStudents[i];
  cin.get();
  return 0;
}
```

Obiekty na diagramach mogą tworzyć związki strukturowe, uczestnicząc w wiązaniach (połączeniach) lub agregacjach. Wiązania są egzemplarzami powiązań i dlatego mogą być charakteryzowane poprzez podanie nazwy, kierunku działania, roli i właściwego stereotypu. Wiązanie to odpowiednik powiązania klas, występujący pomiędzy obiektami. Oznaczamy je za pomocą linii ciągłej, tak jak pokazano to na diagramie z rysunku 1.28.

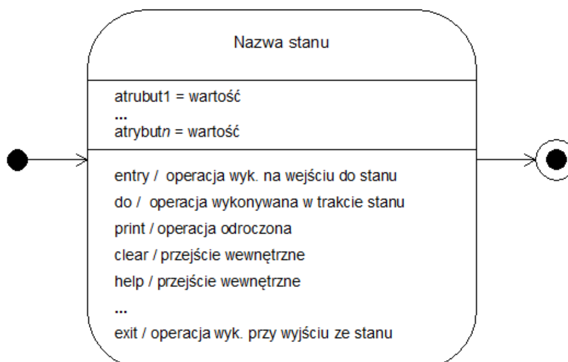


Rysunek 1.28. Wiązanie obiektów

Jeżeli np. klasa `TStudent` jest powiązana z klasą `TBook` (patrz listing 1.4), to między ich obiektami może wystąpić wiązanie. Na ogół wiązania (między obiektami) są egzemplarzami powiązań (między klasami). Dwa obiekty między którymi istnieje wiązanie mogą wysyłać do siebie komunikaty. Standardowe stereotypy wiązań to:
`<<association>>` – umieszczony na jednym końcu wiązania wskazuje, że odpowiadający obiekt jest widoczny przez to wiązanie,
`<<global>>` – umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ jest w otaczającym zasięgu,
`<<local>>` – umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ jest w lokalnym zasięgu,
`<<parameter>>` – umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ jest parametrem,
`<<self>>` – umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ to właśnie on odebrał zlecenie wykonania danej operacji.

1.3.7. Diagramy stanów

Diagramy stanów obrazują sposób, w jaki elementy modelu zmieniają w czasie swoje własności w odpowiedzi na różnego rodzaju zdarzenia i interakcje. Podstawowym elementem tego typu diagramów jest ikona stanu, która może być przedstawiona w sposób uproszczony, poprzez podanie nazwy stanu, lub w sposób szczegółowy, z dodatkowym wyszczególnieniem wartości atrybutów oraz podstawowych, aktualnie wykonywanych operacji i zdarzeń, tak jak pokazano na rysunku 1.29. Symbol stanu oznaczany jest prostokątem z zaokrąglonymi narożnikami. Czarne kółko oznacza punkt wejścia do stanu, strzałka jest symbolem przejścia, zaś kółko z czarną kropką (tzw. bycze oko) oznacza punkt wyjścia ze stanu.



Rysunek 1.29. Przykład modelowania obiektu, którego stan opisują wartości przypisane atrybutom, przejścia wewnętrzne, operacje odroczone oraz operacje wykonywane w trakcie trwania stanu

1.3.8. Diagramy sekwencji

Diagramy klas są doskonałym narzędziem do prezentowania struktury logicznej aplikacji. Są głównym elementem modeli opartych na UML, co sprawiło, że są bardzo popularne i powszechnie stosowane. Jednak ze stosowaniem tego typu diagramów wiążą się poważne problemy:

- diagramy klas nie są wstanie wymodelować dynamicznego zachowania programu – jego zmian w czasie,
- diagramy klas ilustrują jeden poziom hierarchii elementów składowych oprogramowania. Dobrze zaprojektowany program powinien poddawać się dekompozycji hierarchicznej. Klasy i obiekty są podstawą tej dekompozycji.

UML dostrzegając te ograniczenia zapewnia uzupełniające narzędzia takie jak diagramy sekwencji i komunikacji oraz diagramy komponentów i pakietów. Diagramy sekwencji i komunikacji modelują część zachowania dynamicznego, zaś diagramy komponentów i pakietów modelują składniki na poziomie wyższym niż klasy.

Diagramy sekwencji (nazywane też diagramami przebiegu) pozwalają modelować wzajemną interakcję obiektów w funkcji czasu jej trwania. Interakcja traktowana jest jako ciąg zdarzeń występujących w czasie w określonym porządku. Diagram przebiegu składa się z przedstawianych w standardowej postaci obiektów, komunikatów oraz osi czasu (tzw. linii życia).

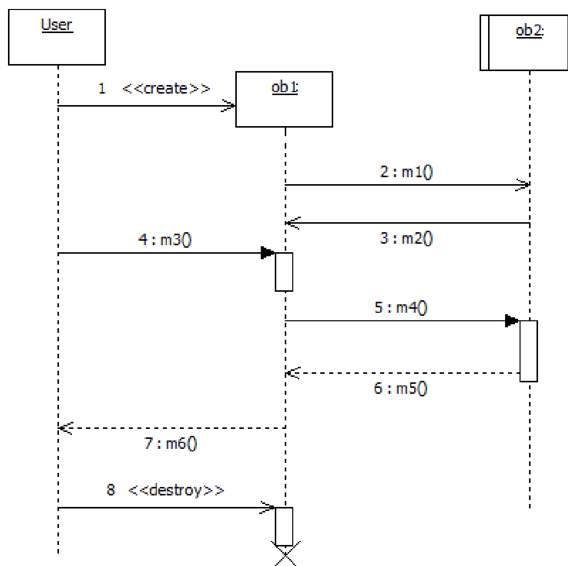
Obiekty mogą na siebie wzajemnie oddziaływać modyfikując stan (wartości) swoich atrybutów lub żądając realizacji usług w postaci udostępnianych operacji. Żądanie wykonania operacji nazywane jest komunikatem. Zwykle komunikat składa się z trzech elementów:

- Wskazania docelowego obiektu, do którego kierowany jest komunikat. Wskazanie takie najczęściej realizowane jest poprzez odwołanie się do nazwy obiektu, tj. do nazwy atrybutu przechowującego obiekt lub referencję do niego.
- Wskazania udostępnianej przez obiekt docelowy operacji, której wykonania żąda obiekt źródłowy. Element ten realizowany jest poprzez wywołanie operacji obiektu docelowego o określonej nazwie i sygnaturze.
- Dodatkowych (opcjonalnych) informacji określających sposób realizacji żądanej usługi. Dodatkowe dane przekazywane są przez obiekt źródłowy do operacji obiektu docelowego realizującej żądaną usługę jako parametry wywołania.

Przetworzenie otrzymanego komunikatu powoduje zmianę stan obiektu docelowego, zaś do obiektu źródłowego może być przesłana wartość powrotna (odpowiedź) jako wynik wykonania operacji obiektu docelowego. Operację, która powoduje oczekiwanie obiektu wywołującego na jej zakończenie (aby móc realizować kolejne instrukcje), nazywamy *blokującą* lub *synchroniczną*. Operację, która zwraca sterowanie natychmiast po jej wywołaniu nazywamy *nieblokującą* lub *asynchroniczną*. Operacje asynchroniczne realizowane są często jako metody zwrotne typu `callback()`. Operacje zwrotne jako jeden z

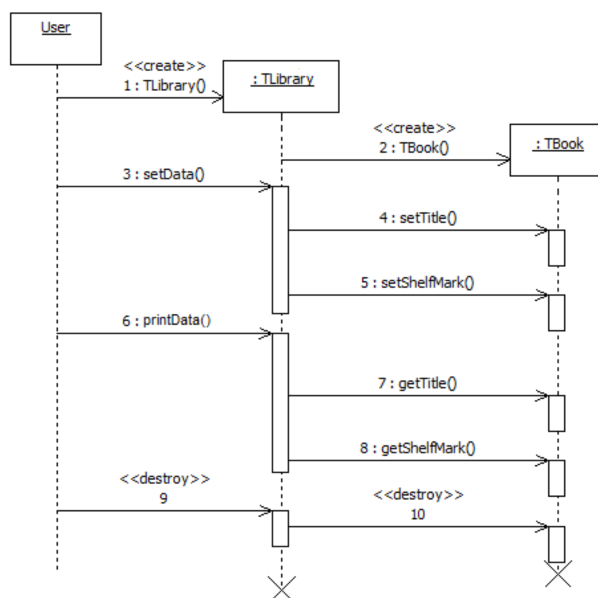
parametrów otrzymują adres obiektu wywołującego, aby po wykonaniu żądania lub w trakcie jego realizacji mieć możliwość dalszego komunikowania się z obiektem wywołującym.

Na diagramach sekwencji obiekty umieszczane są w kolejności od góry z lewej strony diagramu. Kolejność wysyłania lub odbierania komunikatów jest funkcją czasu ich wystąpienia, tzn. biegnie od linii życia obiektu wysyłającego do linii życia obiektu docelowego. UML 2.x definiuje trzy rodzaje komunikatów: synchroniczny, który jest oznaczony strzałką z wypełnionym grotem, komunikat zwrotny, który jest oznaczany przerywaną linią zakończoną otwartym grotem, oraz komunikat asynchroniczny, oznaczany strzałką z otwartym grotem. Przykładem komunikatów asynchronicznych są wywołania konstruktora dla tworzonego obiektu i jego destruktora w trakcie niszczenia obiektu. Komunikaty takie oznacza się odpowiednio stereotypami `<<create>>` i `<<destroy>>`. Na rysunku 1.30 pokazano przykłady obrazowania tego typu komunikatów. Oznaczono ponadto osierocony obiekt aktywny `ob2`, komunikujący się z tworzonym obiektem osieroconym `ob1` poprzez asynchroniczne komunikaty `m1()` i `m2()`.



Rysunek 1.29. Typowy diagram przebiegu

Dla obiektu aktywnego zaznaczono linię jego życia. Obiekt symbolizujący użytkownika jest ze zrozumiałych względów cały czas aktywny, aktywacja `ob1` kończy się w momencie otrzymania komunikatu stereotypowego `<<destroy>>`, co w praktyce oznacza wywołanie przez użytkownika operatora `delete`.

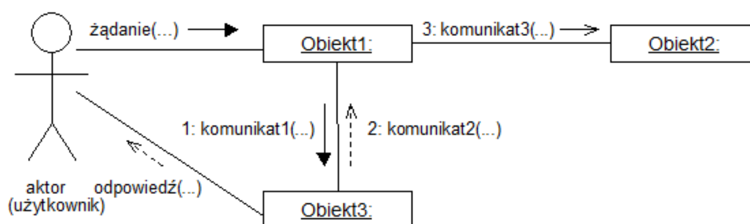


Rysunek 1.30. Szczegółowy diagram sekwencji dla przykładu z listingu 1.7

Na rysunku 1.30 zaprezentowano szczegółowy diagram sekwencji obrazujący kolejność kreacji, destrukcji oraz wymiany komunikatów pomiędzy obiektami z przykładu zamieszczonego na listingu 1.7.

1.3.9. Diagramy komunikacji

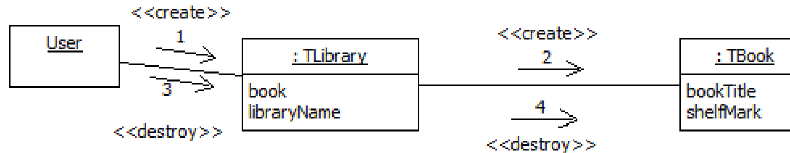
Diagramy komunikacji (dawniej współpracy) umożliwiają pokazanie kolejności wysyłania komunikatów. Kolejność ta obrazowana jest poprzez numery umieszczane na początku etykiet zawierających nazwy i ew. listę argumentów (sygnaturę) odpowiedniego komunikatu. Oznaczenia komunikatów są podobne do stosowanych na diagramach sekwencji. Na rysunku 1.31 pokazano zestaw typowych symboli używanych podczas konstruowania diagramów komunikacji.



Rysunek 1.31. Notacja stosowana na typowych diagramach komunikacji

Diagramy komunikacji pozwalają wymodelować interakcję zachodzącą między obiektami lub/i pomiędzy użytkownikiem systemu i obiektami. W

odróżnieniu od semantycznie bardzo podobnych diagramów przebiegu obrazujących interakcję obiektów w funkcji czasu, diagramy współpracy pokazują otoczenie i ogólną organizację obiektów uczestniczących w konkretnym typie interakcji (np. w wiązaniu). Na rysunku 1.32 zaprezentowano szczegółowy diagram komunikacji obrazujący kolejność kreacji i destrukcji dla obiektów z rysunku 1.30.



Rysunek 1.32. Diagram komunikacji odpowiadający diagramowi sekwencji z rysunku 1.30

W odróżnieniu od diagramów sekwencji, które obrazują obiekty uczestniczące w określonej interakcji i wymieniane przez nie komunikaty jako sekwencję czasową, diagramy komunikacji służą do przedstawiania relacji między rolami obiektu i nie umożliwiają wyrażania czasu jako odrębnego wymiaru. Dlatego kolejność komunikatów na diagramie komunikacji jest wyrażana za pomocą ich numerów.

1.3.10. Komponenty

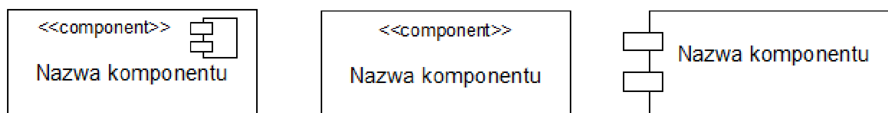
W UML 1.x pliki danych, pliki z kodami programu, programy wykonywalne, biblioteki dołączane dynamicznie, pliki zawierające dane powstałe w wyniku wykonania programu itp. definiowane były odpowiednio jako komponenty procesu wytwórczego, wdrożenia oraz komponenty będące rezultatem wykonania programu. Podział ten często prowadził do nieporozumień. Aby uniknąć wielu niejasności, UML 2.x definiuje komponent jako wymienną część systemu implementującą jedną lub większą liczbę klas.

Wszelkiego rodzaju fizycznie istniejące na dysku pliki określane są ogólnie mianem artefaktów i mogą być oznaczane podstawowym stereotypem `<<artefact>>` (notacja brytyjska) lub `<<artifact>>` (notacja amerykańska).

W odróżnieniu od klasy, która jest pojęciem logicznym istniejącym w umyśle programisty, komponent realnie rezyduje w komputerze. Komponenty z reguły implementowane są poprzez odpowiednie pakiety czasu projektowania.

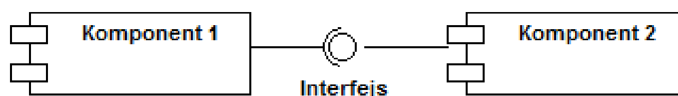
W odróżnieniu od artefaktów, komponenty definiują funkcjonalność systemu. Podobnie jak komponent jest implementacją jednej lub większej liczby klas, tak artefakt (jeżeli jest wykonywalny) może być implementacją komponentu. Stereotyp `<<implement>>` używany jest w przypadku, gdy plik wykonywalny implementuje komponent. Stereotypu `<<manifest>>` używamy, aby pokazać, że komponent jest wyrażany poprzez pliki zawierające jego kod źródłowy. Na diagramach UML komponent może być reprezentowany na trzy

sposoby pokazane na rysunku 1.33.



Rysunek 1.33. Możliwe reprezentacje komponentu

Interfejsy są elementami, za pomocą których współpracują komponenty. Każdy komponent charakteryzuje się dobrze określonym interfejsem. Na rysunku 1.34 zaprezentowano związek dwóch komponentów poprzez interfejs.

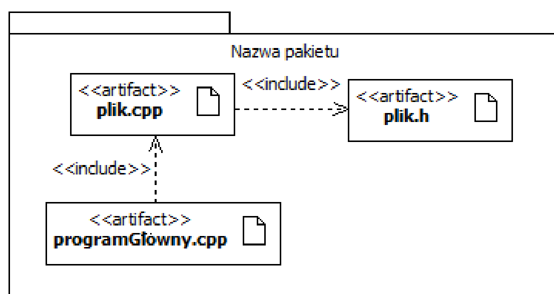


Rysunek 1.34. Komponenty i interfejsy

Komponent to grupa klas pozostających ze sobą w dobrze zdefiniowanych relacjach (dziedziczenie, powiązania, zależności), i służących jednemu, konkretnemu celowi. Klasy w komponencie charakteryzują się tym, iż mają silne związki z innymi klasami wewnątrz tego samego komponentu i słabe poza nim. Dzięki temu komponenty mogą być rozwijane niezależnie i zamiennie wykorzystywane w zależności od kontekstu użycia, bez konieczności modyfikacji innych komponentów. Komponenty komunikują się z innymi komponentami systemu jedynie poprzez jednoznacznie zdefiniowane interfejsy.

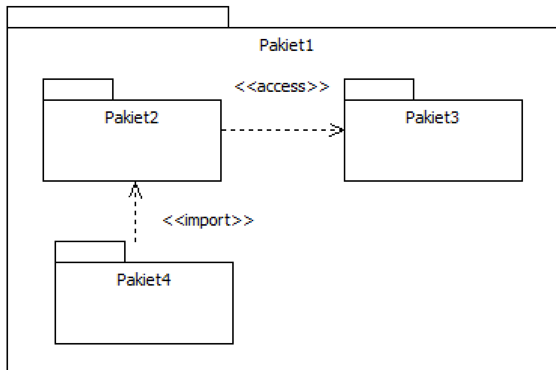
1.3.11. Pakiety

Pakiety służą do grupowania i systematyzowania składników modelu o podobnym przeznaczeniu we wspólnej przestrzeni nazw. Wszystkie elementy pakietu muszą posiadać unikatowy identyfikator. Na rysunku 1.35 pokazano ikonę pakietu grupującego artefakty (pliki) będące składnikami typowego programu C++.



Rysunek 1.35. Podstawowa reprezentacja pakietu

Oprócz artefaktów w pakiety można grupować klasy, interfejsy, komponenty, operacje, przypadki użycia, diagramy lub inne pakiety. Pakiety mogą udostępniać swoje elementy innym pakietom lub importować elementy z innych pakietów.



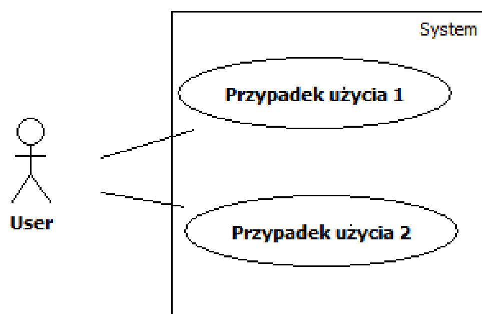
Rysunek 1.36. Operacje na pakietach

Jeżeli jakiś element widoczny jest w obrębie tylko macierzystego pakietu, nazywamy go elementem prywatnym. Jeżeli do wybranego elementu pakietu istnieje możliwość uzyskania dostępu z zewnątrz, wówczas mówimy, że jest on publiczny. Na rysunku 1.36 pokazano zasady importowania `<<import>>` oraz uzyskiwania dostępu `<<access>>` do pakietów lub ich elementów.

1.3.12. Przypadki użycia

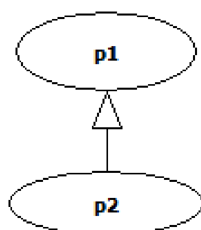
W trakcie procesu analizy systemu przeprowadzanej pod kątem jego przewidywanej funkcjonalności zawsze osiągamy etap, w którym należy zdefiniować wymagania stawiane systemowi przez użytkowników. Definiowane takich wymagań pomaga zrozumieć, jak system powinien lub będzie się zachowywał w kluczowych fazach swojego działania. Definiowanie przypadków użycia pomaga również wyznaczyć granicę oddziaływania systemu i świata zewnętrznego. W najprostszym ujęciu użytkownik zwany aktorem jest reprezentantem świata zewnętrznego. Aktor komunikuje się z systemem poprzez zdefiniowane w jego obrębie przypadki użycia. Przypadek użycia jest dobrze określoną interakcją między użytkownikiem, a systemem komputerowym. Odwzorowuje wybrane (lub wszystkie) funkcje systemu w sposób jaki będą je widzieć przyszli użytkownicy. Pozwala zapomnieć o strukturze (architekturze) systemu, jego detalach logicznych i technicznych oraz nakazuje skoncentrować się na zewnętrznych funkcjach systemu. Na rysunku 1.37 pokazano sposób komunikacji aktora z system poprzez realizację najprostszego modelu przypadków użycia.

Przypadki użycia mogą pozostawać w relacjach dziedziczenia lub zależności. Dziedziczenie pozwala na współdzielenie większości zachowania ogólnego przypadku użycia.



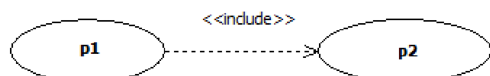
Rysunek 1.37. Najprostszy model przypadków użycia

Na rysunku 1.38 pokazano najprostszy schemat dziedziczenia, w którym jeden przypadek użycia (p2) dziedziczy zachowanie innego (p1) i może dodawać lub modyfikować część tego zachowania. Przypadki użycia mogą pozostawać w relacjach zależności opisujących tzw. przebiegi podstawowe lub/i opcjonalne. Na rysunkach 1.39 i 1.40 pokazano odpowiednio sposoby ich oznaczania.



Rysunek 1.38. Dziedziczenie przypadku użycia

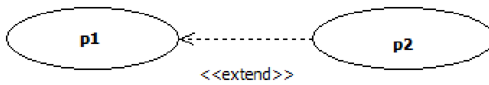
W przebiegu podstawowym (sekwencyjnym) p1 jest przypadkiem bazowym i zawsze występuje jako pierwszy w kolejności działania: p1 zawsze włącza (lub używa) p2. Stereotyp <<include>> wskazuje na wspólny fragment wielu przypadków użycia wykorzystywanych w przebiegach podstawowych, w których wszystkie operacje powinny być zawsze wykonywane.



Rysunek 1.39. Przebieg podstawowy

W przebiegu opcjonalnym (alternatywnym) przypadek występujący jako pierwszy w kolejności działania (p1) jest czasami rozszerzany o p2 (inaczej mówiąc p2 czasami rozszerza p1). Stereotyp <<extend>> ze strzałką prowadzoną od przypadku użycia, który czasami rozszerza inny przypadek użycia

opisuje operacje nie zawsze wykonywane (opcjonalne).



Rysunek 1.40. Przebieg opcjonalny

Model przypadków użycia dostarcza specyficznego spojrzenia na system – spojrzenia z pozycji aktorów, którzy go używają lub będą używać. Nie włącza zbyt wielu szczegółów, co pozwala wnioskować o funkcjonalności systemu na odpowiednio wysokim poziomie. Podstawowym (choć nie jedynym) zastosowaniem modelu przypadków użycia jest dialog z przyszłymi użytkownikami zmierzający do sformułowania poprawnych wymagań odnośnie funkcjonowania systemu. Diagramy przypadków użycia mogą być z powodzeniem stosowane do opisu interakcji użytkownika z konkretną częścią systemu, która została określona poprzez podanie odpowiedniej przestrzenia nazw (np. z głównym oknem aplikacji środowiska graficznego). W takich sytuacjach bardzo często wykorzystywane są różne odmiany tzw. diagramów hybrydowych, łączących elementy diagramów obiektów oraz odpowiadające im przypadki użycia.

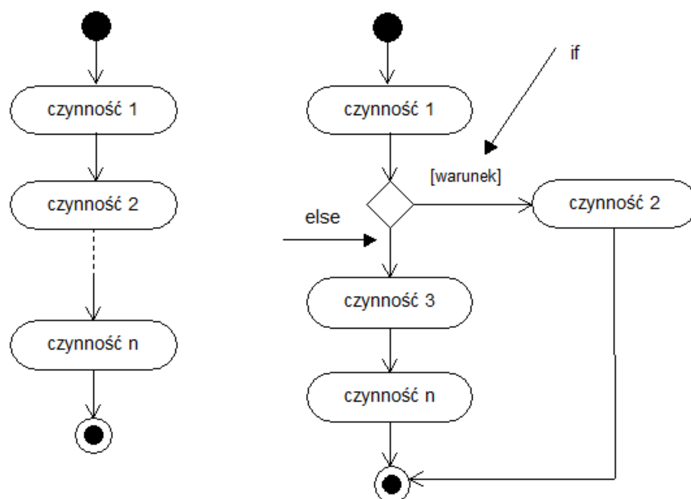
1.3.13. Diagramy czynności

Diagramy czynności wykorzystywane są podczas sporządzania analizy sytemu i służą do modelowania przepływu sterowania między wykonywanymi czynnościami. Czynność jest wieloetapowym działaniem, którego wynikiem jest pewna akcja, składająca się z niepodzielnych kroków prowadzących do zmiany stanu systemu lub przekazania pewnej wartości do innych części systemu. Akcją może być obliczenie wyrażenia arytmetycznego, wywołanie operacji, wysłanie sygnału, utworzenie lub zniszczenie obiektu.

Podczas modelowania dynamicznych aspektów sytemu diagramy czynności można wykorzystywać w dwojaki sposób. Po pierwsze, wykorzystujemy je do modelowania przepływu czynności między poszczególnymi częściami systemu. W tego rodzaju przypadkach diagram czynności można skojarzyć z klasą, komponentem, interfejsem, przypadkiem użycia itp. Po drugie, diagramy te mogą być pomocne do modelowania operacji. Na tego rodzaju diagramach pojęcie operacji może być dwojako rozumiane. Modelowanym bytem może być pojedyncza funkcja lub zespół funkcji i/lub instrukcji wykonywanych np. w głównej funkcji `main()`, gdzie przedstawiać można szczegóły przeprowadzanych obliczeń.

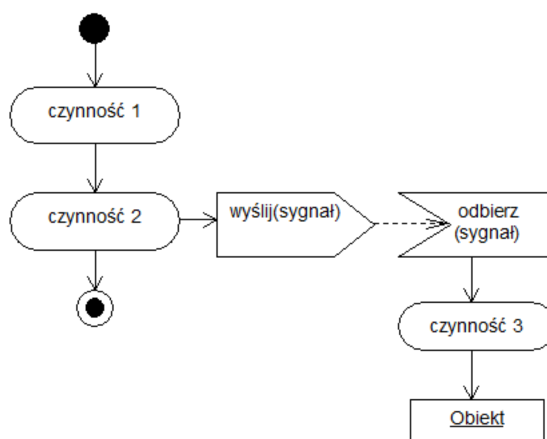
Diagramy czynności wykorzystywane do modelowania operacji są bardzo podobne do używanych dawniej schematów blokowych. Zawartość diagramów czynności może być przedstawiona w sposób ogólny lub szczegółowy. Elementami szczegółowego diagramu czynności dla modelowanej operacji są pojedyncze kroki obliczeniowe, instrukcje decyzyjne oraz rozgałęzienia. Na diagramach czynności wyróżniamy punkt startowy, który jest reprezentowany przez czarne

wypełnione koło oraz punkt końcowy przedstawiany w postaci kółka z czarną kropką w środku. Ogólny obraz dwóch podstawowych typów diagramów czynności pokazano na rysunku 1.41.



Rysunek 1.41. Idea ogólnego oraz szczegółowego diagramu czynności

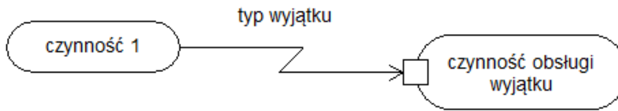
Na diagramach czynności można oznaczać punkty wysłania i odebrania sygnału, symbolizujące wystąpienie zdarzenia wyjściowego i zdarzenia wejściowego. W pierwszym przybliżeniu przez sygnał można rozumieć komunikat, zmieniający stan obiektu, który go otrzymuje (rysunek 1.42).



Rysunek 1.42. Wysłanie i otrzymanie sygnału

Ponieważ diagramy czynności doskonale nadają się do modelowania przepływu czynności w obrębie operacji (funkcji, procedury), dlatego też można je

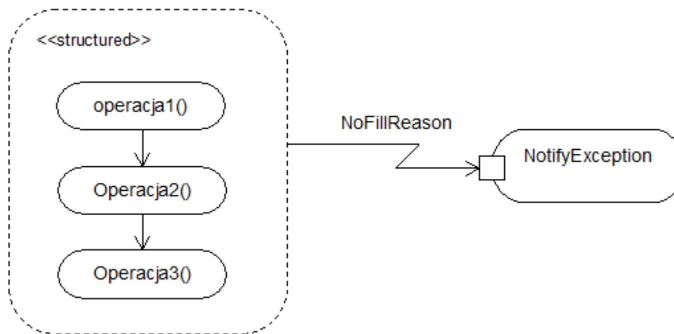
wzbogacić o czynności obsługi wyjątków, tak jak pokazano to na rysunku 1.43.



Rysunek 1.43. Oznaczanie występowania wyjątków

Na tego typu diagramach obrazujemy również mechanizmy strukturalnej obsługi wyjątków. Diagram z rysunku 1.44 odpowiada poniższemu fragmentowi kodu C++.

```
try {
  operacja1();
  Operacja2();
  Operacja3();
}
catch(NoFillReason &reason)
NotifyException(reason);
```



Rysunek 1.44. Oznaczenie strukturalnej obsługi wyjątków

1.3.14. Mechanizmy rozszerzania

Pomimo iż UML określany jest jako zunifikowany język modelowania, jednak nie jest on na tyle ogólny, aby przewidzieć potencjalne problemy mogące powstać w trakcie budowania modelu dowolnego systemu. Standard UML umożliwia jego rozbudowywanie w kontrolowany sposób, dopuszczając stosowanie następujących mechanizmów rozszerzających:

- stereotypy,
- notatki,
- metki,
- ograniczenia.

Stereotypy umożliwią rozszerzania notacji UML. Notatki umożliwiają rozszerzanie listy właściwości dowolnego bloku konstrukcyjnego UML. Ograniczenie jest przedstawiane jako łańcuch znaków zapisanych w nawiasach klamrowych i może być umieszczane niemal w dowolnym miejscu diagramu umożliwiając

rozszerzanie bloku konstrukcyjnego poprzez określenie takich jego elementów, jak: warunki wywołań operacji czy warunki występowania powiązań. Metki mogą być rozszerzalnymi elementami modelu, w którym używany jest stereotyp zawierający ich definicję. Oznaczenie metki zawiera znacznik, znak równości (separator) oraz wartość:

```
znacznik = wartość,
```

gdzie: znacznik jest nazwą metki, wartość zaś jest przypisanym literałem. Jednym z najważniejszych zastosowań metek jest określenie właściwości mających znaczenie podczas generacji kodu w oparciu o model. Metki bardzo często wskazują język programowania, w którym będzie implementowana konkretna klasa.

Podsumowanie

Niniejszy rozdział zawiera krótkie wprowadzenie do języka UML oraz tematyki konstruowania i przedstawiania modeli. Zagadnienie te zostały omówione w sposób zwięzły, co oznacza, iż nie wyczerpują aktualnych zasobów wiedzy i służą jedynie temu, aby zachęcić Czytelnika do samodzielnego rozpoczęcia procesu studiowania zagadnień związanych z nowoczesnymi procesami modelowania oprogramowania.

ROZDZIAŁ 2

PROJEKTOWANIE OPROGRAMOWANIA

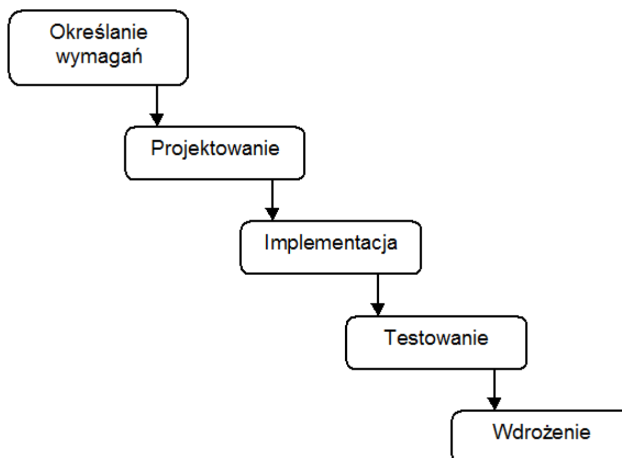
2.1. Proces kaskadowy 56
2.2. Proces iteracyjny 57
2.2.1. Koncepcja początkowa 58
2.2.2. Analiza wymagań 58
2.2.3. Projektowanie 58
2.2.4. Definiowanie klas i obiektów 59
2.2.4.1. Atrybuty 60
2.2.4.2. Metody 60
2.2.4.3. Identyfikacja wymaganych metod 60
2.2.4.4. Właściwości 61
2.2.5. Kontrola hermetyzacji 61
2.2.6. Reguła Podstawiania 63
2.2.7. Projektowanie według kontraktu 63
2.2.8. Model dynamiczny 64
2.2.8.1. Identyfikacja stanów obiektu 64
2.2.9. Implementacja 65
2.2.9.1. Klasy czy struktury? 68
2.2.9.2. Jak rozumieć obiekty? 69
2.3. Architektura sterowana modelem 69
2.3.1. Metamodel 71
Podsumowanie 73

2.1. Proces kaskadowy

Proces obiektowo zorientowanej analizy i projektowania jest dużo bardziej złożony i ważniejszy niż język modelowania. Wynika to z faktu, iż zagadnienia dotyczące wyboru języka modelowania zostały już w dużym stopniu uzgodnione – przemysł zdecydował na używanie UML, natomiast debata na temat wyboru optymalnego procesu wciąż trwa.

Proces projektowania oprogramowania jest z reguły iteracyjny. Oznacza to że opracowując program przechodzimy cały proces wiele razy, dostosowując się do coraz lepszego zrozumienia wymagań. Projekt ukierunkowuje implementację, jednak pojawiające się w trakcie implementacji szczegóły wpływają zwrotnie na projekt. Nigdy nie próbujemy opracować jakiegokolwiek projektu (oczywiście poza przypadkami trywialnymi) w pojedynczym, liniowo-uporządkowanym procesie; zamiast tego powtarzamy rozwijanie fragmentów projektu, wciąż poprawiając jego założenia oraz modyfikując i ulepszając szczegóły implementacji. Opracowywanie iteracyjne należy odróżnić od opracowywania kaskadowego.

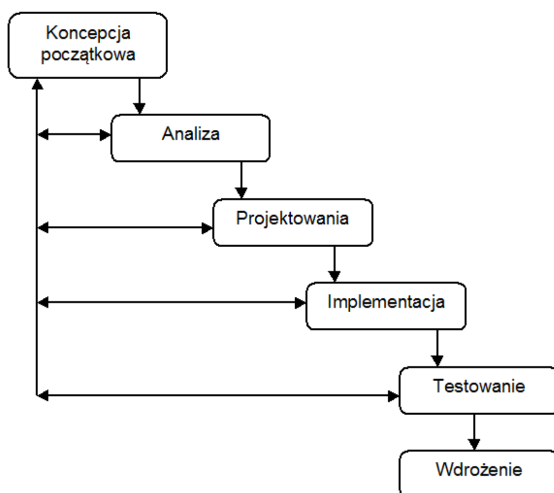
W opracowywaniu kaskadowym wynik jednego etapu staje się wejściem dla następnego, przy czym nie ma powrotu do poprzedniego etapu, tak jak pokazuje to rysunek 2.1. W procesie opracowywania kaskadowego wymagania są szczegółowo przedstawione a następnie te uzgodnione wymagania są przekazywane projektantowi. Projektant tworzy projekt, po czym przekazuje go programiście do implementacji. Z kolei programista udostępnia kod osobie zajmującej się kontrolą jakości, która sprawdza jego działanie i prezentuje go klientowi. Teoretyczne przedstawienie procesu kaskadowego nie sprawia żadnych trudności, inaczej jest w praktyce.



Rysunek 2.1. Opracowanie kaskadowe

2.2. Proces iteracyjny

Opracowywanie iteracyjne rozpoczyna się od koncepcji (idei) obrazującej system. W miarę poznawania szczegółów ta koncepcja podlega ewolucji – może się rozrastać. Gdy wymagania są już dobrze określone, rozpoczyna się faza projektowania. Pracując nad projektem zaczynamy tworzyć prototyp, a następnie wstępną implementację produktu. Zagadnienia pojawiające się podczas opracowywania programu wpływają na zmiany w projekcie i mogą nawet wpłynąć na zrozumienie wymagań. Co najważniejsze, projektujemy i implementujemy jedynie części pełnego produktu, powtarzając na przemian fazy projektowania i implementacji. Chociaż kroki tego procesu są naprzemiennie powtarzane, jednak bardzo trudno jest opisać je na rysunku w sposób cykliczny. Na diagramach przedstawiane są w sposób uproszczony, sekwencyjny: koncepcja początkowa, analiza, projektowanie, implementacja, testowanie, wdrożenie. W rzeczywistości podczas tworzenia pojedynczego produktu wiele razy przechodzimy przez każdy z tych etapów.



Rysunek 2.2. Opracowanie iteracyjne

Istnieją kontrowersje na temat tego, co powinno się dziać na poszczególnych etapach tego procesu, można też spotkać inne jego graficzne prezentacje (nawiasem mówiąc, w literaturze światowej można odszukać kilka odmiennych graficznych prezentacji opracowania iteracyjnego), jednak istota sprawy zawsze pozostaje ta sama [8].

W trakcie pisania książki niektóre przykłady zostały opracowane w omawiany sposób. Nie wszędzie autor szczegółowo dokumentował projekty, gdyż nie jest to celem tego typu publikacji (taką konwencją powinny charakteryzować się nowoczesne podręczniki do inżynierii oprogramowania), jednak zawsze starano się wymodelować najważniejsze elementy kodu wtedy, kiedy omawiano naprawdę zasadnicze pojęcia.

2.2.1. Koncepcja początkowa

Konceptualizacja jest procesem tworzenia pojęć na podstawie ogólnej wiedzy o dziedzinie problemu. Faza konceptualizacji ma na celu sprecyzowanie abstrakcyjnego pojęcia, które stać się ma realnie wytworzonym produktem. Wszystkie programy komputerowe powstają w wyniku przemyśleń dokonywanych przez ich projektantów i twórców. Często mówimy, że osoby takie posiadają wizję systemu, który ma w przyszłości powstać. Najwcześniejszą fazą obiektowo zorientowanej analizy i projektowania jest uchwycenie takiej wizji w formie zwięzłego opisu. Wizja staje się wiodącym celem tworzenia programu, zaś zespół, który zbiera się w celu jej zaimplementowania powinien się do niej odwoływać wraz z postępem prac, a w razie potrzeby nawet ją modyfikować.

2.2.2. Analiza wymagań

Faza konceptualizacji, w której artykułowana jest wizja, jest bardzo krótka. Wiele osób myli opis wizji z wymaganiami, jakie spełniać ma oprogramowanie. Wyraźna wizja jest potrzebna, lecz sama w sobie nie jest wystarczająca. Aby przejść do analizy, należy zrozumieć jak program będzie używany i jak powinien działać. Celem fazy analizy jest wyartykułowanie i wychwycenie tych wymagań. Wynikiem tej fazy jest stworzenie dokumentu z opracowanymi wymaganiami. Pierwszą częścią tego dokumentu jest analiza przypadków użycia produktu.

Przypadek użycia jest ogólnym opisem tego, w jaki sposób system będzie używany. Przypadki użycia nie tylko ukierunkowują analizę, ale także pomagają w określeniu klas i są szczególnie ważne podczas testowania programu.

Tworzenie stabilnego i wyczerpującego zestawu przypadków użycia może być najważniejszym zadaniem w całej analizie.

Kiedy wstępne przypadki użycia zostały już ustalone, należy opracować szczegółowy model dziedziny problemu. Model dziedziny jest dokumentem obejmującym wszystko to co aktualnie wiemy o danej dziedzinie (zagadnieniu, nad którym pracujemy). Jako część modelu dziedziny tworzone są obiekty dziedziny opisujące wszystkie obiekty uwzględnione w przypadkach użycia. Należy zdawać sobie sprawę z faktu, iż *obiekty dziedziny nie są obiektami projektu.* Model dziedziny problemu opisuje sposób funkcjonowania świata rzeczywistego, a nie sposób działania tworzonego systemu komputerowego.

2.2.3. Projektowanie

Faza analizy skupia się na zrozumieniu dziedziny problemu, podczas gdy projektowanie skupia się na stworzeniu optymalnego rozwiązania. Projektowanie jest procesem przekształcenia zrozumienia wymagań w model, który może być zaimplementowany w postaci oprogramowania. Wynikiem tego procesu jest stworzenie dokumentu projektowego. Dokument projektowy formalnie powinien być podzielony na dwie części: *projekt klas* oraz *mechanizmy architektury*. Część projektu klas dzieli się z kolei na projekt sta-

tyczny (diagramy szczegółowo opisujące poszczególne klasy, ich powiązania, zależności i charakterystyki) oraz projekt dynamiczny (diagramy opisujące jak obiekty poszczególnych klas ze sobą współpracują). Projekt mechanizmów architektury zawiera szczegóły dotyczące implementacji artefaktów systemu. W dalszej części rozdziału skupimy się na aspekcie projektowania klas bezpośrednio związanym z wiodącą tematyką książki.

Metodyka obiektowa jest szczególnym rodzajem metodyki wykorzystującym paradygmat obiektowości dla celów modelowania pojęciowego oraz analizy i projektowania systemów informatycznych. Podstawowymi elementami metodyki obiektowej są przedstawione w Rozdziale 1 diagramy struktury statycznej modelu, opisujące:

- klasy,
- interfejsy,
- specyfikacje atrybutów i operacji,
- związki generalizacji,
- związki zależności,
- związki powiązań, agregacji i kompozycji,
- liczebności tych związków,
- związki realizacji interfejsu przez klasę,
- obiekty.

Uzupełnieniem mogą być:

- diagramy struktury dynamicznej, modelujące: sekwencje kreacji i destrukcji obiektów, kolejność przekazywania komunikatów, stany i przejścia międzystanowe,
- diagramy przypadków użycia, których podstawowym celem jest odwzorowanie struktury systemu z punktu widzenia użytkownika.

2.2.4. Definiowanie klas i obiektów

Klasa jest strukturą danych definiującą wewnętrzny stan obiektu (atrybuty), jego zachowanie (operacje), klasy po których dziedziczy lub pozostaje w innych związkach (patrz Rozdział 1). Ogólnie rzecz ujmując można powiedzieć, iż klasa to struktura danych definiująca abstrakcję danych oraz udostępniająca jej całkowitą lub częściową implementację. Należy być świadomym faktu, iż pojedyncza klasa to nie to samo co definiowana przez nią abstrakcja. Klasa jest pojęciem definiującym abstrakcję danych jako przyrostową modyfikację jej klas bazowych. Klasa jest wymagana tylko i wyłącznie w momencie tworzenia abstrakcji. Abstrakcja danych istnieje niezależnie wraz ze swoimi atrybutami i metodami. Wiele z nich z reguły pochodzi z klas bazowych, a nie z klasy, która była bezpośrednio użyta do utworzenia danej abstrakcji.

Można tworzyć wiele obiektów jednej klasy. Obiekty te nazywane są egzemplarzami lub instancjami klasy. Mają one własne tożsamości i własne dane stanu. Wszystkie obiekty danej klasy powinny zachowywać się w sposób,

jaki nakazuje jej definicja.

Nazwa klasy powinna być pisana w formie rzeczownika w liczbie pojedynczej lub rzeczownika z przymiotnikiem i możliwie wiernie odzwierciedlać standardowe słownictwo dziedziny problemu.

Klasa definiuje elementy, które będzie posiadać każdy jej obiekt. W terminologii obiektowej elementy te nazywane są składowymi klasy (ang. *class members*). W Rozdziale 1 w formalizmie UML opisano trzy rodzaje takich składowych.

2.2.4.1. Atrybuty

Atrybuty to wewnętrzne dane abstrakcji, które nie powinny być widoczne na zewnątrz, co odpowiada omówionemu dalej zakresowi prywatnemu. Atrybuty (w terminologii obiektowej nazywane zmiennymi egzemplarza) są komórkami zawierającymi część danych stanu egzemplarza klasy. Atrybut powinien być widoczny wyłącznie w obrębie definicji klasy lub/i wszystkich klas dziedziczących. Każdy obiekt ma własny zbiór atrybutów i uzyskuje do nich dostęp poprzez instrukcje przypisań, operacje dostępu lub operacje wymiany. W analizie obiektowej atrybut powinien być zdefiniowany tak, aby odzwierciedlał zarówno dziedzinę problemu, jak i obowiązki systemu. Każdy atrybut należy umieścić w klasie, którą najlepiej opisuje.

2.2.4.2. Metody

Metody są *wewnętrznym interfejsem* abstrakcji danych i powinny być dostępne dla wszystkich elementów programu odwołujących się do danej abstrakcji, co odpowiada opisanemu dalej zakresowi publicznemu. Metoda to rodzaj funkcji lub procedury, która jest implementacją operacji w kontekście danego obiektu i która ma wyłączny dostęp do jego atrybutów.

Metoda składa się z nagłówka i ciała. Nagłówek jest etykietą i zbiorem argumentów. Składnia większości języków programowania wymaga, aby każdy argument był inną zmienną. Zdefiniowanie metody powoduje uszczegółowienie abstrakcji modelowanego rzeczywistego obiektu wskazując, jakie zachowanie obiektu danej klasy jest przewidywane w systemie. Najczęściej używane są trzy typy klasyfikacji takich zachowań:

- na podstawie bezpośrednich związków przyczynowych,
- pod względem podobieństwa historii rozwoju (zmian w czasie),
- po względem podobieństwa funkcji.

2.2.4.3. Identyfikacja wymaganych metod

Pod względem algorytmicznym metody dzielą się na:

- proste (traktowane często jako niejawne),
- złożone.

Metody algorytmiczne proste stosuje się do każdej klasy i obiektu w modelu. Wykonują one swoje działania od początku pracy sytemu według tego samego

schematu. Rozróżnia się cztery typy metod algorytmicznych prostych:

- metody typu utwórz – zajmują się tworzeniem i inicjowaniem nowego obiektu,
- metody typu powiąż – ustalają powiązania lub/i zależności obiektu z innymi obiektami,
- metody typu pobierz/ustaw – służą do pobierania i ustawiania wartości atrybutów obiektów,
- metody typu zwolnij – obsługują procedury finalizacji i ew. usunięcia obiektu z pamięci.

Metody algorytmiczne złożone dzielą się na dwie kategorie:

- metody typu oblicz – służą do obliczenia wyniku korzystając z wartości atrybutów obiektu,
- metody typu monitoruj – służą do monitorowania systemu zewnętrznego.

Kategorii metod używa się w trakcie projektowania do stwierdzenia, jakie metody są aktualnie niezbędna dla danego obiektu.

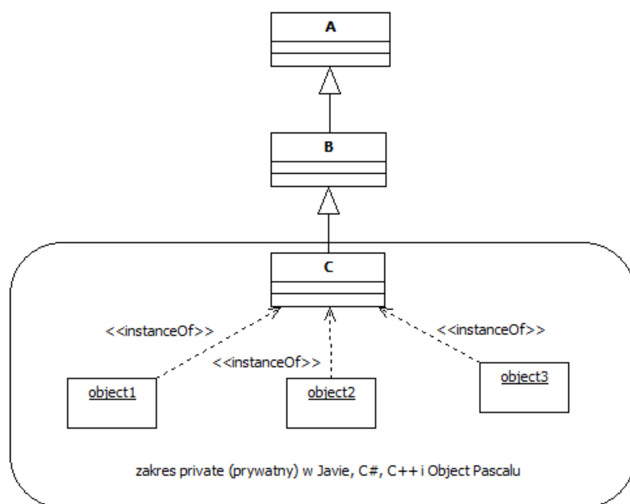
2.2.4.4. Właściwości

Właściwości (ang. *properties*) nie są elementami wbudowanymi wszystkich języków programowania. Właściwość modyfikuje zachowanie obiektu poprzez odpowiednie konstrukcje metod typu pobierz/ustaw.

2.2.5. Kontrola hermetyzacji

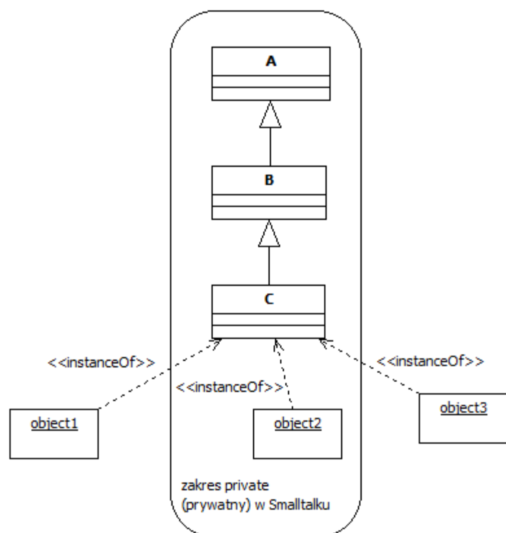
W językach obiektowych kontrola hermetyzacji polega na ograniczeniu dostępu do składowych klasy (atrybutów i metod) zgodnie z projektowanymi wymaganiami architektury systemu. Każda składowa definiowana jest zgodnie z odpowiednim zakresem dostępności. Zakres dostępności definiuje się jako część tekstu programu, w którym składowa jest widoczna, tj. możliwe jest uzyskanie do niej dostępu poprzez podanie jej nazwy. Współczesne języki programowania określają zakres dostępu każdej składowej klasy w momencie jej deklarowania za pomocą słów kluczowych `public`, `private`, `protected` lub `package`. Najbardziej podstawowymi zakresami to zakres prywatny, publiczny i chroniony.

Składowa prywatna (ang. *private member*) jest widoczna wyłącznie w egzemplarzu klasy, w którym została zdefiniowana. Składowa prywatna nie jest dostępna podklasom i ich egzemplarzom. Należy być świadomym faktu, iż prywatność nie jest pojęciem ogólnym i może być różnie interpretowana przez różne języki programowania. Większość nowoczesnych języków interpretuje prywatność w sposób zgodny z powyższą definicją, tak jak pokazano to na rysunku 2.3.



Rysunek 2.3. Znaczenie terminu `private` w Javie, C#, C++ i Object Pascalu

Jednak już np. Smalltalk zakres prywatny interpretuje w sposób przedstawiony na rysunku 2.4.



Rysunek 2.4. Znaczenie terminu `private` w języku Smalltalk

Składowa chroniona (ang. *protected member*) jest dostępna wyłącznie w klasie, w której została zdefiniowana i klasach dziedziczących (oraz wszystkich egzemplarzach tych klas).

Składowa publiczna (ang. *public member*) jest widoczna w całym progra-

mie. W takich językach obiektowych jak Java, C#, C++ i Object Pascal atrybuty w miarę możliwości powinny być prywatne, a metody publiczne, w sensie powyższych definicji.

2.2.6. Reguła Podstawiania

Regułę Podstawiania sformułowała Barbara Liskov w roku 1988. Można ją przedstawić następująco [9]:

Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.

Co w uproszczonej interpretacji sprowadza się do następującej zasady:

Typ C jest podtypem typu B, a B jest podtypem typu A, jeżeli program może używać obiektów typu C zamiast obiektów typu A i B nie zdając sobie z tego sprawy (patrz rys. 2.3).

2.2.7. Projektowanie według kontraktu

Program jest poprawny, jeżeli jego działanie odpowiada specyfikacji projektowej [8]. Jednym ze sposobów dowodzenia poprawności programu jest wnioskowanie oparte na formalnej semantyce. W oparciu o metody formalne Bernard Meyer opracował metodę projektowania poprawnych programów, którą nazwał Projektowanie Według Kontraktu (ang. *design by contract*) DBC i zastosował ja jako element języka Eiffel [10]. Główną cechą tej metody jest to, że abstrakcja danych powinna implikować pewną umową pomiędzy projektantem abstrakcji a jej użytkownikiem. To użytkownicy są zobowiązani zagwarantować, że abstrakcja będzie wywoływana we właściwy sposób. DBC polega na spisaniu umowy danej funkcjonalności (klasy, metody, itp.) z resztą kodu. Umowa taka składa się z trzech punktów:

- warunki początkowe (ang. *preconditions*) – opisują zobowiązania otoczenia wobec funkcjonalności, które muszą być spełnione w chwili rozpoczęcia jej wykonywania,
- warunki końcowe (ang. *postconditions*) – opisują zobowiązania funkcjonalności wobec otoczenia, które muszą być spełnione wraz z zakończeniem działania funkcjonalności,
- niezmienniki (ang. *invariants*) – opisują warunki które muszą być zapewnione przez cały czas trwania umowy.

Niezmienniki bardzo trudno jest implementować i sprawdzać w językach, które nie posiadają wbudowanego mechanizmu DBC, zatem najczęściej określa się warunki początkowe i końcowe. Poniżej zamieszczono nieskomplikowany przykład prostej umowy dla metody `bookLent()` (wypożycz książkę) klasy `TLibrary`, zapisanej w języku OCL [11]:

```
TLibrary::bookLent(accountCounter : integer)
pre : (activeLibraryAccount() = true) and
```

```
        (howManyStayed () > accountCounter)
    post: howManyStayed() = howManyStayed()@pre - accountCounter
```

Oznacza to, że kiedy wypożyczymy książkę z biblioteki, to liczba egzemplarzy, które możemy w przyszłości zamówić i być w ich posiadaniu zmniejszy się o liczbę książek obecnie wypożyczonych (jeżeli oczywiście w międzyczasie nie dokonamy zwrotu).

DBC jest techniką projektowania obiektowego, która jest związana z zasadą Liskov. Regułę Liskov można dość łatwo zinterpretować w kontekście DBC. Interpretacja ta brzmi:

Typ C jest podtypem typu B, a B jest podtypem A, jeśli wymaga nie więcej niż B i A i zapewnia nie mniej.

Przed wywołaniem metody użytkownik zapewnia spełnienie warunków wstępnych, zaś implementacja abstrakcji danych gwarantuje, iż w momencie zakończenia operacji będą spełnione warunki końcowe. Tak właśnie przebiega podział odpowiedzialności. Użytkownik jest odpowiedzialny za warunki wstępna, zaś abstrakcja danych za warunki końcowe.

Stosowanie reguły Liskov przynosi wymierne korzyści w trakcie konstruowania i testowania projektu. Z możliwości udostępnianych przez DBC należy jednak korzystać rozsądnie pamiętając o ważnej zasadzie:

Wszelkiego rodzaju technologie zostały stworzone dla użytkowników – nie zaś odwrotnie.

W językach takich jak Object Pascal, Java, C# i C++, gdzie głównym zadaniem dziedziczenia nie jest rozszerzanie zachowania, lecz polimorfizm typów, może to mieć kolosalne znaczenie dla poprawnego działania systemu. Projektowanie oprogramowania jest sztuką zawierania ciągłych kompromisów, dlatego może się zdarzyć, że reguły Liskov nie da się spełnić, albo wręcz nie będzie to opłacalne. To właśnie projektant powinien prawidłowo ocenić aktualną sytuację.

2.2.8. Model dynamiczny

Oprócz modelowania relacji pomiędzy klasami, bardzo ważne jest także wymodelowanie tego, jak klasy te ze sobą współpracują. Diagramy interakcji występują w dwóch odmianach. Odmiana pokazana na rysunku 1.30 jest nazywana diagramem sekwencji. Innego widoku tych samych informacji dostarczają diagramy współpracy. Diagramy sekwencji kładą nacisk na kolejność zdarzeń w czasie, zaś diagramy komunikacji obrazują współdziałania pomiędzy klasami. Diagram komunikacji można stworzyć bezpośrednio z diagramu sekwencji (patrz rysunek 1.32)

2.2.8.1. Identyfikacja stanów obiektu

Przechodząc do zrozumienia interakcji pomiędzy obiektami, należy zrozumieć różne możliwe stany każdego z obiektów. Przejścia pomiędzy stanami możemy wymodelować na diagramie stanu (lub diagramie zmian stanów).

Każdy obiekt przechodzi przez różne stany od momentu, w którym jest

utworzony do momentu, w którym zostaje usunięty z pamięci. Stan obiektu jest reprezentowany przez wartości jego atrybutów. Każda zmiana wartości atrybutu jest odbiciem zmiany stanu.

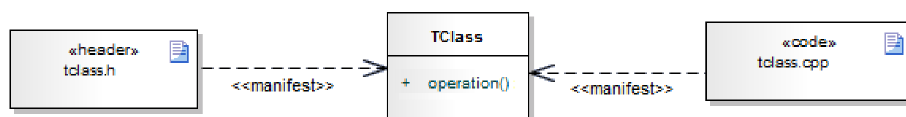
Stan obiektu identyfikuje te wartości atrybutów, które odzwierciedlają zmianę w zachowaniu obiektu. Aby zidentyfikować stany obiektu należy:

- zbadać potencjalne wartości atrybutów,
- stwierdzić, czy w świetle zobowiązań systemu zachowanie obiektu jest różne dla tych potencjalnych wartości.

2.2.9. Implementacja

Standardowo każda klasa powinna mieć swój własny plik nagłówkowy zawierający jej definicją oraz (jeżeli to konieczne) własny plik źródłowy zawierający zapis ciał funkcji składowych. Nazwy tych plików pochodzą od nazwy klasy z odpowiednim rozszerzeniem.

Wiele rozszerzeń języka C++ wymaga pisania wyodrębnionych deklaracji klas w plikach nagłówkowych. Związek pomiędzy klasą (będącą elementem logicznym systemu) a jej plikami implementacyjnymi (będącymi elementami fizycznymi systemu) nazywamy pokazywaniem, manifestacją lub wyrażaniem. Na diagramach UML związek ten w postaci zależności określany jest stereotypem `<<manifest>>`. Elementy logiczne są zawsze niezależne od elementów fizycznych, poprzez które są wyrażane. Artefakt zawierający definicję klasy (plik nagłówkowy) oznacza się stereotypem `<<header>>`, zaś artefakt z implementacją metod klasy oznacza się stereotypem `<<code>>`, tak jak pokazano to na rysunku 2.5 oraz listingach 2.1-2.2.



Rysunek 2.5. Wyrażenie elementu logicznego przez artefakty implementacyjne C++

Listing 2.1 Implementacja pliku nagłówkowego `<<header>>` z diagramu 2.5

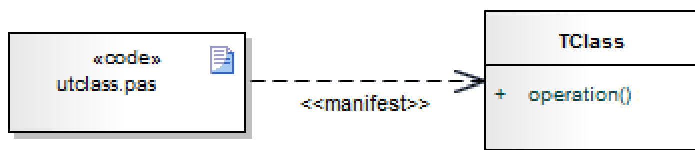
```
//TClass.h
#ifndef __TCLASS_h__
#define __TCLASS_h__
class TClass  {
   public:
       TClass();
       virtual ~TClass();
       void operation();
};
#endif
```

Listing 2.2. Implementacja pliku źródłowego <<code>> z diagramu 2.5

```
//  TClass.cpp
#include "TClass.h"
TClass::TClass() {} //konstruktor domyślny
TClass::~TClass(){} //wirtualny destruktor domyślny
void TClass::operation(){
  //ciało metody
}
```

W tym miejscu warto zauważyć, że niektóre standardy implementacyjne C++ zalecają przechowywanie elementów źródłowych zawierających zapis ciał metod w plikach z rozszerzeniem **.hpp* (np. biblioteka boost). Dla kompilatora C++ nie stanowi to żadnego problemu, gdyż jest to plik tekstowy i może zostać dołączony do pliku **.cpp* z programem głównym (pliku zawierającego definicję funkcji `main()`) albo dyrektywą `#include` albo poprzez dyrektywy wiersza linii poleceń kompilatora.

Podział kodu między nagłówek z deklaracją klasy oraz plik implementacyjny jest ogólnie przyjętą konwencją kodowania w C++. Inne języki programowania inaczej podchodzą do tego zagadnienia. Na rysunkach 2.6-8 zaprezentowano odpowiednio sposoby wyrażania klasy właściwe dla Object Pascala, Javy i C#. Najbardziej zauważalną różnicą pomiędzy C++ a pozostałymi językami jest brak plików nagłówkowych.

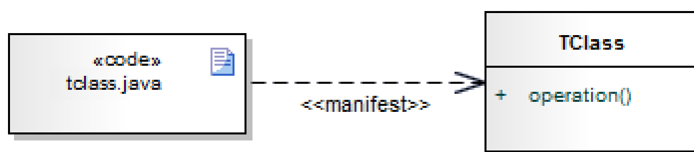


Rysunek 2.6. Wyrażenie elementu logicznego przez artefakt implementacyjny Object Pascala

Listing 2.3. Zawartość typowego artefaktu implementacyjnego Object Pascala

```
unit uTClass;
interface
type
  TClass = class
  public
     procedure operation;
     constructor Create; overload;
     destructor Destroy; override;
  end;
//------------------------------------------------------------
implementation
{implementation of TClass}
```

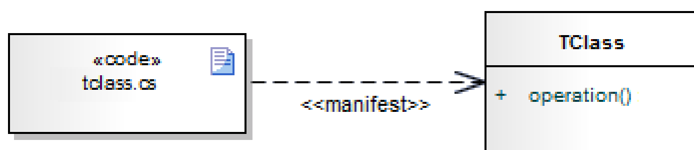
```
constructor TClass.Create;
begin
   inherited Create;
end;
//-----------------------------------------------------------
destructor TClass.Destroy;
begin
   inherited Destroy;
end;
//-----------------------------------------------------------
procedure TClass.operation;
begin
  //ciało procedury
end;
end.
```



Rysunek 2.7. Wyrażenie elementu logicznego przez artefakt implementacyjny Javy

Listing 2.4. Zawartość typowego artefaktu implementacyjnego Javy

```
/* tclass.java*/
public class TClass {
  public TClass(){
     }
     public void finalize() throws Throwable {
   }
     public void operation(){
   }
}
```



Rysunek 2.8. Wyrażenie elementu logicznego przez artefakt implementacyjny C#

Listing 2.5. Zawartość typowego artefaktu implementacyjnego C#

```
//tclass.cs
public class TClass {
```

```
    public TClass(){
        }
        ~TClass(){
        }
   public virtual void Dispose(){
        }
   public void operation(){
        }
}//end TClass
```

Jak zaznaczono we wstępie głównym językiem programowania, który będzie wykorzystywany w tym podręczniku jest C++. Pomimo, iż ogólnie przyjętą zasadą stosowaną podczas kodowania w C++ jest podział kodu między nagłówek z deklaracją klasy oraz plik implementacyjny, to ze względu na prostotę dalszych rozważań, w trakcie książki autor ograniczy się do przedstawiania elementów kodu w postaci jednego artefaktu implementacyjnego. Głównie z tej przyczyny, aby w tekście nie mnożyć liczby listingów i maksymalnie uprościć odwoływanie się do nich. Czytelnicy, którzy niechętnie rezygnują ze sztywnych standardów kodowania bez trudu mogą samodzielnie dokonać dekompozycji zamieszczonych kodów zgodnie z wytycznymi diagramu 2.5.

2.2.9.1. Klasy czy struktury?

W C++, Javie, C# i Object Pascalu klasy definiuje się używając słowa kluczowego `class`. Programując w C++ i C# można użyć słowa `struct`, należy jednak zdawać sobie sprawę z konsekwencji takiego wyboru. W C++ struktury i klasy są w zasadzie równoważne (z wyjątkiem konwencji dotyczących domyślnego poziomu dostępności). Obiekty mogą być tworzone na stosie, gdzie obowiązuje dla nich semantyka wartości (tak jak dla lokalnych zmiennych typów wbudowanych), albo na stercie za pomocą operatora `new`. Utworzone na stercie obiekty są anonimowe, zaś dostęp do nich uzyskuje się poprzez zmienne zawierające wskaźniki do tych obiektów. W Object Pascalu i Javie struktury nie są zdefiniowane. Obiekty tworzone są wyłącznie na bazie klas i są zawsze przechowywane na stercie. Obiekty są zawsze anonimowe, zaś dostęp do nich uzyskuje się wyłącznie poprzez zmienne zawierające referencje do tych obiektów. W Object Pascalu i Javie referencje do obiektów są w rzeczywistości wskaźnikami używanymi w sposób niejawny (są wskaźnikami na miejsce w pamięci, gdzie znajduje się obiekt). W C# obiekty struktur tworzone są na stosie, niezależnie od tego czy użyto operatora `new` czy nie. Dla takich obiektów obowiązuje semantyka wartości. Obiekty klas tworzone są, podobnie jak w Javie i Object Pascalu, zawsze na stercie, zaś dostęp do nich uzyskuje się poprzez referencje. Poniżej przedstawiono ogólne konstrukcje umożliwiające alokowanie obiektów na stercie właściwe odpowiednio dla: C++, Javy, C# i Object Pascala.

C++
```
TClass *object = new TClass();
object->operation();
delete object;
```

Java
```
TClass object = new TClass();
object.operation();
```

C#
```
TClass object = new TClass();
object.operation();
```

Object Pascal
```
var object: TClass;
//...
object := TClass.Create;
object.operation();
object.Free;
```

2.2.9.2. Jak rozumieć obiekty?

Tradycyjnie obiekt definiuje się jako zestaw danych wraz z metodami. W nowoczesnym ujęciu sposób rozumienia problemu obiektów polega na założeniu, że obiekty charakteryzują się posiadaniem określonego rodzaju odpowiedzialności. W trakcie projektowania systemu obiektowego należy zwrócić uwagę na to, aby obiekty były odpowiedzialne za własne działanie oraz aby rodzaje odpowiedzialności były poprawnie określone. Nowoczesne rozumienie obiektu opiera się na wykorzystaniu trzech perspektyw sformułowanych przez Martina Flowera:

- na poziomie *koncepcji* i *analizy* obiekt jest zbiorem określonego typu odpowiedzialności,
- na poziomie *specyfikacji* obiekt jest zbiorem metod (zachowań), które mogą być wywoływane przez metody macierzystego obiektu lub innych obiektów,
- na poziomie *implementacji* obiekt jest wyrażany za pomocą kodu i danych oraz interakcji między nimi.

2.3. Architektura sterowana modelem

Opracowany na początku XXI wieku przez organizację OMG (ang. *Object Management Group*) standard wdrażania oprogramowania sterowanego modelem MDD (ang. *Model-Driven Development*) jest bardzo nowoczesnym i technologicznie zaawansowanym podejściem do ewolucji obiektowo-zorientowanej architektury oprogramowania. MDD wykorzystuje rozwój sterowany modelami, który jest rozszerzeniem paradygmatu architektury sterowanej modelami, MDA (ang. *Model-Driven Architecture*) na wszystkie aspekty ewolu-

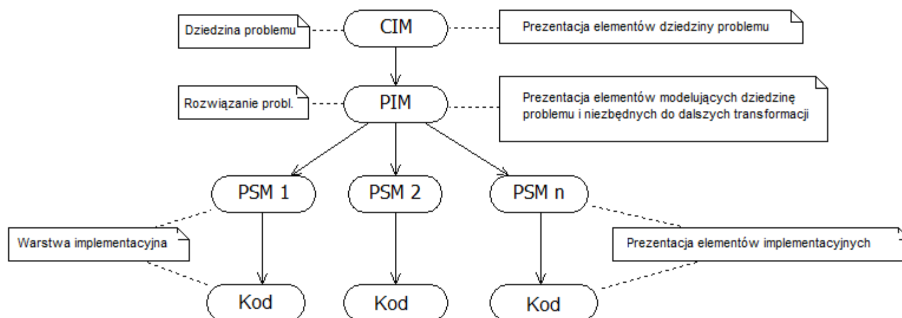
cji systemu [12]. MDA nie jest w ścisłym tego słowa znaczeniu całkowicie nową metodologią tworzenia oprogramowania. W rzeczywistości MDA jest rozwinięciem opracowania iteracyjnego z wprowadzeniem mechanizmów automatycznej transformacji modeli [13]. W praktyce użycie MDA sprowadza się do:

- przeniesienia ciężaru rozwoju systemu na wyższy poziom abstrakcji i nadanie modelowaniu centralnej roli,
- ścisłego oddzielenie warstw systemu,
- automatycznej generacji kodu bezpośrednio z modelu,
- wprowadzeniu mechanizmów automatycznej weryfikacji i walidacji kodu.

Zgodnie z założeniami organizacji OMG wyróżnia się cztery warstwy oprogramowania tworzonego zgodnie z wytycznymi MDA:

- *Computation-Independent Model* (CIM) – model dziedzinowy, nie pozostający w ścisłej relacji z technologia informatyczną;
- *Platform-Independent Model* (PIM) – abstrakcyjna, niezależna od platformy systemowej i programistycznej specyfikacja systemu wykorzystująca metamodel;
- *Platform-Specific Model* (PSM) – model odwzorowany na konkretne rozwiązania wybranej platformy systemowej i programistycznej;
- *Code* – automatycznie generowany kod niskopoziomowy (np. kod Object Pascala, Javy, C#, C++).

Na rysunku 2.9 pokazano schemat cyklu życia oprogramowania tworzonego zgodnie ze standardem MDA.



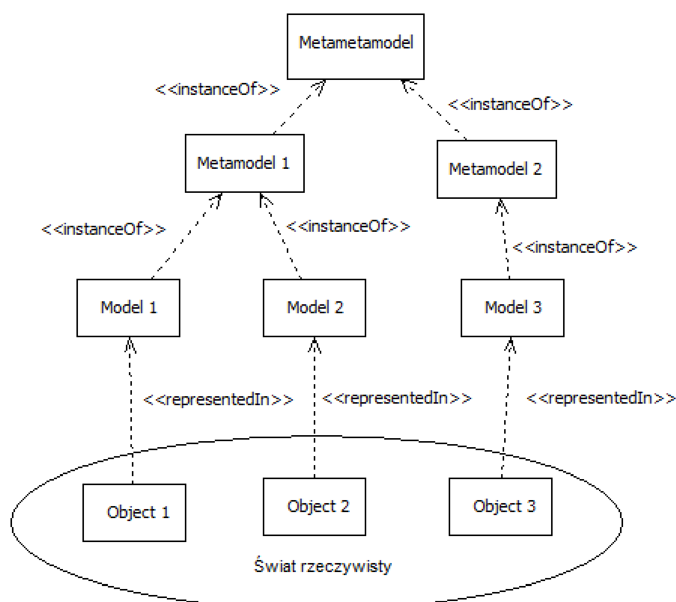
Rysunek 2.9. Relacje między warstwami MDA

Architektura sterowana modelami jest zbiorem kilku standardów opracowanych i rozwijanych przez OMG. Składa się z następujących specyfikacji: UML, Meta-Object Facility (MOF), XML Meta-Data Interchange (XMI), Object-Constraint Language (OCL), Query-View-Transformation (QVT) oraz Common Warehouse Meta-model (CWM) [14-17]. Narzędzia MDA umożliwiają zachowanie wzajemnych relacji pomiędzy modelami, oraz dostarczają mechanizmów

służących do transformacji modeli na podstawie ustalonych wcześniej reguł i standardów [18-20].

2.3.1. Metamodel

UML jako język służący do opisu wszystkich rodzajów programowych artefaktów jest standardem obiektowego modelowania. Tak jak zaprezentowano to w Rozdziale 1, UML posiada notację graficzną, co implikuje fakt, iż powinien być też zaopatrzony w wbudowany mechanizm służący do definiowania tej notacji. Mechanizmem tym jest metamodel. Metamodel jest modelem, który definiuje język i gramatykę służącą do wyrażania innych modeli. Rysunek 2.10 prezentuje przykład stosu metamodeli.



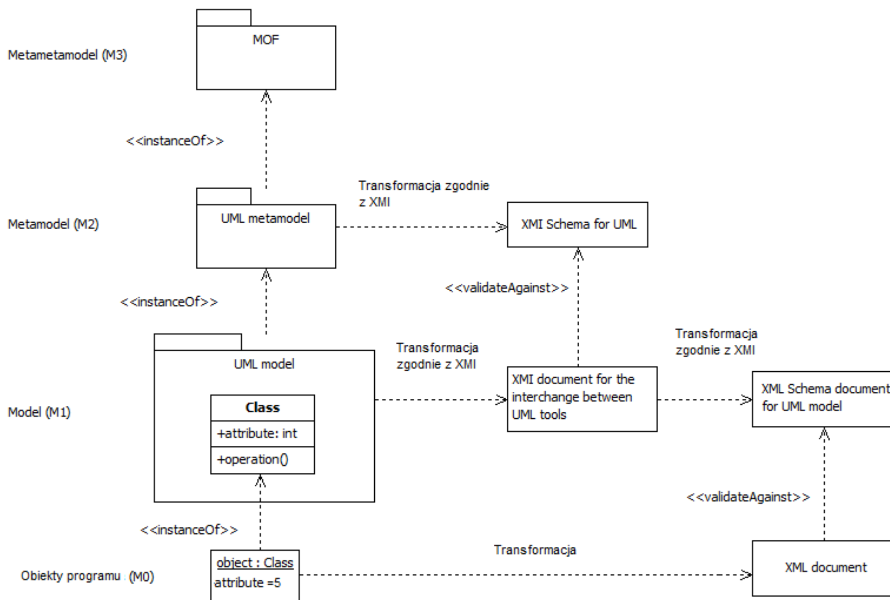
Rysunek 2.10. Stos metamodeli

Na rysunku 2.10 modele reprezentują zjawiska ze świata rzeczywistego i są egzemplarzami swoich własnych metamodeli. W terminologii tej języki są nazywane metamodelami, zaś stworzone w tych językach artefakty (włączając programy wykonywalne) nazywane są modelami.

Metamodele UML definiują strukturę UML. Metametamodel (lub metajęzyk) jest modelem, który definiuje język do opisu metamodeli. UML jest zdefiniowany w formie metametamodelu nazywanego MOF (Meta-Object Facility). MOF charakteryzuje mechanizmy niezbędne do przechowywania i uzyskiwania dostępu do różnych języków. Tak jak pokazuje to rysunek 2.11, UML i MOF stanowią podstawę czteropoziomowego stosu modeli metodyki MDA.

Na poziomie M0 występują konkretne realizacje modeli, na przykład kon-

kretne obiekty generowane w trakcie wykonania programu. Na poziomie M1 występują modele, np. kody Object Pascala, Javy, C#, C++ generowane na podstawie diagramów klas. Na poziomie M2 mamy do czynienia z metamodelami, które odgrywają rolę gramatyk do definiowania modeli np. pojęcia generalizacji, powiązań i zależności klas. Na poziomie M3 występuje metametamodel. W standardzie MDA, MOF umożliwia definiowanie wielu meta-modeli na poziomie M2.



Rysunek 2.11. Struktura warstwowa UML

Standardowymi, zaakceptowanymi przez przemysł metamodelami są: UML i CWM. Głównymi standardami przyjętymi w metodyce MDA są:

- zunifikowany język modelowania UML jako standardowy język do definiowania metamodeli,
- XML Meta-Data Interchange XMI jako standardowy mechanizm wymiany metadanych i metamodeli za pomocą XML pomiędzy różnymi narzędziami wizualnego modelowania,
- Common Warehouse Meta-model CWM jako standardową specyfikacją do definiowania struktury i semantyki metadanych w hurtowniach danych i eksploracji danych,
- mapowanie metametamodeli do języka IDL (MOF-to-IDL mapping) jako standardowy mechanizm dostępu do metadanych za pomocą API (niezależnie od języka programowania i modelu obiektowego),
- mapowanie metametamodeli do języka Java (MOF-to-Java mapping) jako standardowy mechanizm dostępu do metadanych w Javie.

Dodatkowo wielu producentów narzędzi zgodnych z technologią MDD udostępnia profilowane mechanizmy mapowania metametamodeli do C++, Object Pascala, C#, PHP i Pythona jako mechanizmy dostępu do metadanych odpowiednio w C++, Object Pascalu, C#, PHP i Pythonie.

Wprowadzenie standardu XMI pozwoliło na serializację metamodeli, co oznacza możliwość reprezentacji diagramów w formacie XML i ich wzajemną wymianę pomiędzy narzędziami różnych producentów. Do obecnie najpopularniejszych narzędzi wspierających technologie MDD-MDA należą Eclipse, Together oraz Sparx Enterprise Architect.

Podsumowanie

Istnieje wiele metod porządkujących szeroko rozumiany proces wytwarzania oprogramowania. W literaturze omawiane są one głównie w kontekście wspomagającym aktualnie istniejące procesy informacyjno-decyzyjne oraz jako elementy kształtujące długofalową strategię przedsiębiorstwa informatycznego (w szeroko rozumianym tego słowa znaczeniu). Praktyka wskazuje, że dla pojedynczych użytkowników lub niewielkich organizacji bardzo dobrze sprawdza się metodologia oparta na programowaniu iteracyjnym. Garść informacji na temat teoretycznych podstaw technologii MDA powinna okazać się przydatna dla osób zainteresowanych automatyzacją procesu wytwarzania oprogramowania zorientowanego obiektowo.

ROZDZIAŁ 3

DZIEDZICZENIE I DELEGOWANIE

3.1. Mechanizm dziedziczenia 76
3.2. Odwołania i wskaźniki do klas pochodnych 77
3.3. Wskaźniki do funkcji składowych 80
3.4. Funkcje wirtualne 83
3.5. Obiekty funkcyjne w hierarchii dziedziczenia 90
3.6. Statyczny polimorfizm 92
3.7. Dziedziczenie wielokrotne 98
3.8. Informacja czasu wykonywania 104
3.8.1. Operator dynamic_cast 106
3.8.2. Zliczanie obiektów wielokrotnych 108
3.9. Delegowanie 110
3.9.1. Delegowanie z pomocą obiektów funkcyjnych 111
Podsumowanie 112

3.1. Mechanizm dziedziczenia

Mechanizm dziedziczenia (ang. *inheritance*) opiera się na spostrzeżeniu, że abstrakcje danych modelujące wybraną dziedzinę problemu z reguły mają wiele wspólnych elementów. Często zadawane przez programistów pytanie brzmi: *Czy można zaimplementować je bez powtarzania części wspólnych?* Występowanie powtórzeń zawsze oznacza dłuższy i bardziej skomplikowany logicznie oraz fizycznie program. Z punktu widzenia programisty, każdej zmianie powtarzanego fragmentu kodu musi towarzyszyć zmiana we wszystkich jego kopiach. Co więcej, kopie te mogą różnić się w niewielkim stopniu, co dodatkowo komplikuje określenie właściwych modyfikacji.

Główny powodem, dla którego obiektowe języki programowania wprowadziły mechanizm dziedziczenia jest dążenie do maksymalnego ograniczenia występowania problemu powtarzania się kodu oraz to, aby związki pomiędzy abstrakcjami danych stały się maksymalnie przejrzyste. Z podstawowej definicji klasy dziedziczącej można wywnioskować, iż może być potomkiem jednej lub większej liczby innych klas. Wówczas, podstawą zachowań klasy dziedziczącej będą operacje zdefiniowane w klasach dziedziczonych. Mogą zostać one jedynie uzupełnione w macierzystej klasie o pewne rozszerzenia i modyfikacje. W mechanizmie dziedziczenia definicja nowej klasy dokonywana jest poprzez pewien rodzaj transformacji, w której jedna lub większa liczba klas zostaje w sposób hierarchiczny trwale połączona z opisem nowych atrybutów i operacji.

Dziedziczenie jest jedną z najchętniej wykorzystywanych technik obiektowego programowania, jednak praktyka wskazuje, iż powinna być stosowane bardzo rozsądnie. Przede wszystkim należy wykazać się bardzo dobrą znajomością budowy klas nadrzędnych. Należy pamiętać, iż mechanizm dziedziczenia powoduje ustanowienie nowego wewnętrznego interfejsu, co oznacza, że dodawanie nowych klas w hierarchii generalizacji powinno być traktowane jako ustanowienie nowego typu interakcji programu z klasą bazową (klasą korzeniem). Interakcja programu użytkownika z dobrze skonstruowaną hierarchią klas dziedziczących niezmiennie powinna odbywać się za pośrednictwem pewnych standardów opisujących budowę klas liści (klas finalnych), czyli takich, po których nie można już dziedziczyć. Dziedziczenie umożliwia strukturalizowanie kodu tak, aby elementy wspólne modelu były definiowane tylko raz. Jednak ma to również swoja cenę w postaci rozproszenia implementacji na wiele różnych (chociaż współpracujących ze sobą) artefaktów.

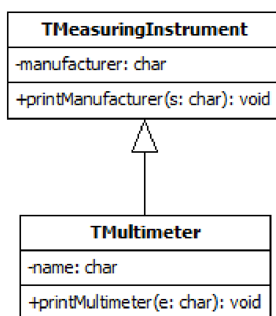
Mechanizm dziedziczenia stosowany w odniesieniu do klas abstrakcyjnych, polimorficznych i konkretnych w znacznym stopniu rozwiązuje problem ponownego wykorzystania kodu. Jednak obecnie coraz większą wagę przywiązuje się do tworzenia systemów generycznych opartych na wzorcach klas [21]. U podstaw programowania generycznego leży założenie, które pozwala podzielić elementy programu na dwie grupy: obiekty mające swój stan oraz algorytmy operujące na obiektach, ale nieposiadające stanu. Dodatkowym założeniem (tylko pozornie paradoksalnym) jest to, iż dane i algorytmy nie zawierają wiedzy na swój temat, zaś systemy generyczne mogą mieć wiele

konkretyzacji. Konkretyzacja systemu generycznego polega na nadaniu wartości parametrom uogólnionym wzorca klasy. W dalszej części rozdziału zostanie pokazane, w jaki sposób jest to realizowane przy użyciu mechanizmów dziedziczenia i wzorców klas.

W analizie i projektowaniu zorientowanym obiektowo występują dwa bardzo ważne pojęcia: *wczesne* oraz *późne* wiązanie. Z wczesnym wiązaniem (ang. *early binding*) mamy do czynienia w sytuacjach, gdy funkcje w trakcie kompilacji programu wiąże się z określonymi obiektami. Przykładem wczesnego wiązania będą np. sytuacje, w których w głównej funkcji `main()` programu C++ wywołujemy funkcje standardowe i przeładowane oraz funkcje przedefiniowywanych standardowych operatorów. Sytuacje, gdy wywoływane funkcje wiązane są z określonymi obiektami w trakcie działania programu, określamy mianem późnego wiązania (ang. *late binding*). Przykładem późnego wiązania są funkcje wirtualne, klasy polimorficzne i abstrakcyjne. Późne wiązanie jest skutkiem działania metod zadeklarowanych ze słowem kluczowym `virtual` (metody wirtualne). Wielką zaletą technik związanych z późnym wiązaniem jest możliwość stworzenia prawdziwego interfejsu użytkownika wraz z odpowiednią biblioteką klas, którą można niemal swobodnie uzupełniać i modyfikować.

3.2. Odwołania i wskaźniki do klas pochodnych

Na podstawie wiadomości z kursowych wykładów programowania w języku C++ możemy wywnioskować, iż wskaźnik określonego typu nie może wskazywać na dane odmiennych typów. Niemniej jednak od tej reguły istnieje pewne bardzo ważne odstępstwo. Rozpatrzmy sytuację, w której zaimplementowaliśmy w programie pewną klasę zwaną klasą bazową oraz klasę z niej dziedziczącą — czyli klasę pochodną (potomną), tak jak pokazano to na rysunku 3.1.



Rysunek 3.1. Przyrząd pomiarowy jest generalną formą miernika uniwersalnego

Okazuje się, że wskaźniki do klasy bazowej mogą również w określonych sytuacjach wskazywać na reprezentantów lub elementy klasy pochodnej. Załóżmy, iż w programie zaimplementowaliśmy klasę bazową

`TMeasuringInstrument` z publiczną funkcją składową odczytującą nazwę producenta przyrządów pomiarowych. Następnie stworzymy klasę pochodną `TMultimeter` z publiczną funkcją składową odczytującą nazwę oraz typ konkretnego przyrządu. W funkcji `main()` zadeklarujemy zmienną `ptrMI` jako wskaźnik do klasy `TMeasuringInstrument`:

```
TMeasuringInstrument *ptrMI;
```

Zadeklarujmy również po jednym egzemplarzu (obiekcie) klas `TMeasuringInstrument` i `TMultimeter`:

```
TMeasuringInstrument instrument;
TMultimeter multimeter;
```

Okazuje się, że zmienna deklarowana jako wskaźnik do typu bazowego `TMeasuringInstrument` może wskazywać nie tylko na obiekty klasy bazowej, ale również i pochodnej:

```
ptrMI=&instrument;
ptrMI=&multimeter;
```

Za pomocą tak określonego wskaźnika `ptrMI` można uzyskać dostęp do wszystkich elementów klasy `TMultimeter` odziedziczonych po klasie `TMeasuringInstrument`, tak jak pokazano to na listingu 3.1.

Listing 3.1. Implementacja diagramu 3.1

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

class TMeasuringInstrument  // klasa bazowa
{
    char  manufacturer[40];
  public:
    void printManufacturer(char *s) {
       strcpy( manufacturer, s);
       cout <<  manufacturer << endl;
    }
};
//-----------------------------------------------------------
class TMultimeter: public TMeasuringInstrument // klasa
                                               // pochodna
{
    char name[5];
  public:
```

```
    void printMultimeter(char *e) {
      strcpy(name, e);
      cout << name << endl;
    }
};
//--------------------------------------------------------------
int main()
{
  // ptrMI-wskaźnik do klasy TMeasuringInstrument (bazowej)
  TMeasuringInstrument *ptrMI;
 // instrument-egzemplarz klasy TMeasuringInstrumen
  TMeasuringInstrument instrument;
  // ptrMultimeter-wskaźnik do klasy TMultimeter (pochodnej)
  TMultimeter *ptrMultimeter;
  // multimeter-egzemplarz klasy TMultimeter
  TMultimeter multimeter;
  // wskaźnik ptrMI wskazuje na egzemplarz
  // klasy TMeasuringInstrument
  ptrMI=&instrument;
  ptrMI->printManufacturer("Texas Instruments");
  // wskaźnik ptrMI wskazuje na egzemplarz
  // klasy TMultimeter, będącej klasą pochodną względem
  // klasy bazowej TMeasuringInstrument
  ptrMI=&multimeter;
  ptrMI->printManufacturer("Lake Shore");
  // funkcja printMultimeter() jest elementem
  // klasy pochodnej. Dostęp do niej uzyskujemy
  // za pomocą wskaźnika ptrMultimeter
  ptrMultimeter = &multimeter;
  ptrMultimeter->printMultimeter("Voltmeter K6; 2.5 kV");
  // uzyskanie dostępu do funkcji składowej klasy pochodnej
  // za pomocą wskaźnika do klasy bazowej
  ((TMultimeter *)ptrMI)->printMultimeter("Ammeter FG55;
                                          20 A");
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Texas Instruments
Lake Shore
Voltmeter K6; 2.5 kV
Ammeter FG55; 20 A
```

Śledząc powyższe zapisy, z łatwością przekonamy się, iż wskaźnik do klasy bazowej może równie dobrze wskazywać na te elementy klasy pochodnej, które

są zdefiniowane również w klasie bazowej, z tego względu, że klasa `TMultimeter` dziedziczy publiczne elementy klasy `TMeasuringInstrument`. Jednak, używając w prosty sposób wskaźnika do klasy bazowej, nie można uzyskać dostępu do tych elementów, które występują jedynie w klasie pochodnej. W przypadku, gdy zażądalibyśmy uzyskania dostępu np. do funkcji składowej `printMultimeter()` klasy pochodnej, należałoby wykorzystać zmienną `ptrMI` będącą jawnym wskaźnikiem do klasy `TMultimeter`. Jeżeli mimo wszystko ktoś zdecydowałby się, aby za pomocą wskaźnika do klasy bazowej uzyskać dostęp do jakiegoś elementu klasy pochodnej, będzie musiał wykonać w odpowiedni sposób operację rzutowania typów:

```
((TMultimeter *)ptrMI)->printMultimeter("Ammeter FG55;
                                          20 A");
```

Poprzez wykorzystanie zewnętrznej pary nawiasów informujemy kompilator, iż rzutowanie łączone jest ze wskaźnikiem `ptrMI`, a nie z wartością funkcji `printMultimeter()`. Występowanie wewnętrznej pary nawiasów określa sytuację, w której wskaźnik `ptrMI` do klasy bazowej rzutowany jest na typ klasy pochodnej `TMultimeter`.

3.3. Wskaźniki do funkcji składowych

Programowania zgodne z paradygmatem obiektowym zakłada, iż działający program składa się ze zbioru obiektów. Każdy obiekt zawiera wartości przypisane do atrybutów określających jego stan oraz operacje pozwalające zmieniać stan lub odczytywać informację o stanie.

W językach ogólnego przeznaczenia (takich jak C++) stosowanie obiektowego stylu programowania może prowadzić do pojawienia się sytuacji, w których programista staje przed koniecznością uzyskania informacji o stanie obiektu oraz konieczności ewentualnej zmiany jego stanu za pomocą funkcji zewnętrznych, zamiast funkcji składowych klasy. Zgodnie ze starszym standardem ANSI języka C++ uzyskanie wskaźnika do funkcji składowej klasy możliwe jest jedynie poprzez odpowiednie zadeklarowania (z użyciem operatora dostępu pośredniego) wskaźnika do klasy, a następnie poprzez wykorzystanie operatora adresowego `&` odczytanie adresu funkcji składowej. Warto zdawać sobie sprawę z faktu, iż w ten sposób nie można uzyskać wskaźnika do funkcji składowej klasy dziedziczącej po którejś ze swoich klas bazowych [22].

Posługiwanie się wskaźnikami wskazującymi na funkcje składowe klasy jest jednym ze sposobów zmiany zachowania funkcji składowych klasy w czasie wykonywania programu. Standardowa deklaracja takiego wskaźnika ma postać:

```
typedef typ (Class::*ptr)(param);
```

gdzie `typ` jest typem wartości powrotnej funkcji składowej klasy, `Class` jest nazwą klasy, w której jest zadeklarowana funkcja, `ptr` jest wskaźnikiem do

funkcji składowej klasy, zaś `param` jest wykazem parametrów funkcji składowej. Tak skonstruowany wskaźnik `ptr` jest następnie wykorzystywany do zadeklarowania innego wskaźnika `ptrFunc`, inicjowanego wskazaniem na żądaną funkcję składową `Func()`:

```
ptrFunc() = &Class::Func;
```

Jako przykład rozpatrzmy przyrząd zwany zasilaczem (jest on niezbędny w każdym laboratorium naukowym). W ogólności, każdy zasilacz może znajdować się w jednym z dwóch podstawowych stanów: włączonym (ON) lub wyłączonym (OFF). W rzeczywistości przejścia pomiędzy tymi dwoma dyskretnymi stanami realizowane są poprzez sygnał pochodzący od włącznika (przycisku), który wyzwala odpowiednią reakcje układu sterującego przyrządem. Program, którego kod przedstawiono w ramach listingu 3.2 wykorzystuje wskaźnik do funkcji składowej `switchOnOff()`, któremu następnie przypisywany jest adres jednej z funkcji: `off()` lub `on()`.

Listing 3.2. Emulacja zasilacza laboratoryjnego

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

class TPowerSupply {
  private:
    bool state;
  public:
   ~TPowerSupply() {};
    void (TPowerSupply::*switchOnOff) ();
    void off() { state = false; cout << "Current OFF\n";}
    void on() { state = true; cout << "Currnt ON\n"; }
    void controlFunc(int i) {
        switch (i) {
          case 0: switchOnOff = &TPowerSupply::off; off();
              break;
          case 1: switchOnOff = &TPowerSupply::on; on();
              break;

        }
    }
};
//----------------------------------------------------------
int main()  {
   TPowerSupply powerSupply;
   typedef void (TPowerSupply::*ptrPS) (int);
   ptrPS ptr = &TPowerSupply::controlFunc;
```

```
   int option;
   while(true) {
     cout << "Enter 0-OFF, 1-ON, 2-exit:\n\n";
     cin >> option;
     if (option == 2)
       exit(EXIT_SUCCESS);
       else {
         (powerSupply.*ptr)(option);
         // lub
         //(powerSupply.*(powerSupply.switchOnOff))();
       }
   }
   cin.get();
   return 0;
}
```

Wynik działania programu:

```
Enter 0-OFF, 1-ON, 2-exit:

1
Currnt ON
Enter 0-OFF, 1-ON, 2-exit:

0
Current OFF
Enter 0-OFF, 1-ON, 2-exit:
```

Dwa specyficzne dla C++ operatory `.*` (kropka gwiazdka) i `->*` (strzałka gwiazdka) określane są mianem operatorów wskaźnikowych do elementów składowych klas. W odróżnieniu od zwykłych operatorów kropka (`.`) i strzałka (`->`) umożliwiają tworzenie zmiennych wskazujących do elementów klas, a nie do ich egzemplarzy. Tego typu operatory wskaźnikowe definiują adresy względne, pod którymi można znaleźć wszystkie elementy wybranej klasy. W funkcji `main()` z listingu 3.2 tworzony jest wskaźnik `ptrPS`, wskazujący na funkcję jako jedyny element składowy klasy `TPowerSupply`. Po utworzeniu egzemplarza `powerSupply` klasy `TPowerSupply` pobrany zostaje adres funkcji `TPowerSupply::controlFunc`. Operator rozróżniania zakresu (`::`) określa klasę, do której funkcja przynależy. W celu określenia aktualnego stanu zasilacza używamy wskaźnika `ptrPS` — aby bez problemu wywołać funkcję należącą do nazwanego obiektu `powerSupply` klasy `TPowerSupply`.

W przypadku, gdy potrafimy uzyskać dostęp do wskaźnika do klasy (lub innego obiektu), możliwe jest zastosowanie operatora (`->*`). W przykładzie zaprezentowanym na listingu 3.3 funkcja składowa `controlFunc()` ma postać identyczną jak poprzednio. Jednak w funkcji `main()` tym razem deklarowany jest wskaźnik `powerSupply` do klasy `TPowerSupply`. Dostęp do funkcji składowej klasy uzyskujemy za pomocą operatora (`->*`).

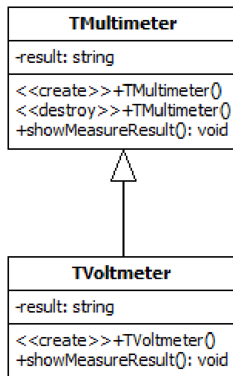
Listing 3.3. Deklaracja wskaźnika do klasy finalnej

```
int main()  {
   TPowerSupply *powerSupply = new TPowerSupply;
   typedef void (TPowerSupply::*ptrPS) (int);
   ptrPS ptr = &TPowerSupply::controlFunc;
   int option;
   while(true) {
     cout << "Enter 0-OFF, 1-ON, 2-exit:\n\n";
     cin >> option;
     if (option == 2)
       exit(EXIT_SUCCESS);
       else {
         (powerSupply->*ptr)(option);
         (powerSupply->*(powerSupply->switchOnOff))();
       }
   }
   delete powerSupply;
   cin.get();
   return 0; }
```

3.4. Funkcje wirtualne

Funkcją wirtualną (ang. *virtual function*) nazywamy taką funkcję, która jest zadeklarowana w klasie bazowej za pomocą słowa kluczowego `virtual`, a następnie w takiej samej postaci redefiniowana również w klasach pochodnych. Funkcje takie bardzo często określa się mianem funkcji kategorii *virtual*. Ponownie definiując funkcję wirtualną w klasie pochodnej, możemy (ale nie musimy) powtórnie umieszczać słowo `virtual` przed jej nazwą. Funkcje wirtualne mają bardzo ciekawą właściwość. Charakteryzują się mianowicie tym, iż podczas wywoływania dowolnej z nich za pomocą odwołania lub wskaźnika do klasy bazowej wskazującego na egzemplarz klasy pochodnej, aktualna wersja wywoływanej funkcji każdorazowo ustalana jest w trakcie wykonywania programu z rozróżnieniem typu wskazywanej klasy. Klasy, w których zdefiniowano jedną lub więcej funkcji wirtualnych, nazywamy klasami polimorficznymi.

Jako praktyczny sposób wykorzystania klas polimorficznych rozpatrzmy przykład, gdzie zadeklarowano nieskomplikowaną klasę bazową `TMultimeter` reprezentującą urządzenia laboratoryjne zwane miernikami uniwersalnymi z funkcją `showMeasureResult()` kategorii `virtual`, której jedynym zadaniem jest wyświetlenie hipotetycznego wyniku pomiaru napięcia prądu. Ponieważ funkcja jest rzeczywiście funkcją wirtualną, możemy ją z powodzeniem redefiniować (tzn. zdefiniować jej kolejną wersję) w klasie pochodnej `TVoltmeter` (woltomierz), dziedziczącej publiczne elementy klasy `TMultimeter`. Sytuację tę ilustruje diagram z rysunku 3.2 oraz odpowiadający mu przykład z listingu 3.4.



Rysunek 3.2. Miernik uniwersalny jest generalną formą woltomierza

Listing 3.4. Przykład implementacji klas polimorficznych

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

class TMultimeter {//miernik uniwersalny
  private:
    string result;
  public:
    TMultimeter() {//konstruktor klasy miernik uniwersalny
      showMeasureResult();
    }
    virtual ~TMultimeter(){};  //destruktor wirtualny
    virtual void showMeasureResult() {
      result = "215 V";
      cout << "Pomiar miernika uniwersalnego: "<< result
                                               << endl;
    }
};
//-----------------------------------------------------------
class TVoltmeter : public TMultimeter {//woltomierz
  private:
    string result;
  public:
    TVoltmeter() {//konstruktor klasy woltomierz
      showMeasureResult();
    }
    void showMeasureResult() {
      result = "215.5 V";
      cout << "Pomiar woltomierza: " << result << endl;
    }
```

```
};
//-------------------------------------------------------------
int main()
{
  TVoltmeter voltmeter;//wywołanie funkcji
                       //showMeasureResult()
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Pomiar miernika uniwersalnego: 215 V
Pomiar woltomierza: 215.5 V
```

Jak łatwo zauważyć, w celu wywołania funkcji `showMeasureResult()` w głównej funkcji `main()` zawarto jedynie deklarację nazwanego obiektu klasy pochodnej. Wynika to z faktu, iż zarówno klasa bazowa, jak i pochodna zawierają odpowiednio zaimplementowane konstruktory.

Na listingu 3.5 zamieszczono przykład ilustrujący ideę posługiwania się wskaźnikami do klas polimorficznych, co w efekcie pozwala na pominięcie jawnych implementacji ciał konstruktorów odpowiednich klas.

Listing 3.5. Implementacja wskaźników do klas polimorficznych

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

class TMultimeter {  //miernik uniwersalny
  private:
    string result;
  public:
    TMultimeter() {}
    virtual ~TMultimeter(){};  //destruktor wirtualny
    virtual void showMeasureResult() {
      result = "215 V";
      cout << "Pomiar miernika uniwersalnego: "<< result
                                              << endl;
    }
};
//-------------------------------------------------------------
class TVoltmeter : public TMultimeter {    //woltomierz
  private:
    string result;
  public:
    TVoltmeter(){}
    void showMeasureResult() {
```

```
      result = "215.5 V";
      cout << "Pomiar woltomierza: " << result << endl;
    }
};
//-----------------------------------------------------------
int main()
{
  TMultimeter multimeter;
  TMultimeter *ptrMultimeter;
  TVoltmeter voltmeter;
  ptrMultimeter = &multimeter;
  // wywołanie funkcji showMeasureResult() klasy TMultimeter
  ptrMultimeter->showMeasureResult();
  ptrMultimeter=&voltmeter;
  // wywołanie funkcji showMeasureResult() klasy TVoltmeter
  ptrMultimeter->showMeasureResult();
  cin.get();
  return 0;
}
```

W podanym przykładzie w klasie bazowej (`TMultimeter`) definiowana jest funkcja wirtualna `showMeasureResult()`, po czym jej kolejna wersja zdefiniowana jest względem klasy pochodnej (`TVoltmeter`). W głównej funkcji `main()` zawarto w kolejności deklarację obiektu `multimeter` klasy bazowej, wskaźnika `ptrMultimeter` do klasy bazowej i obiektu `voltmeter` klasy pochodnej. Dzięki instrukcjom:

```
ptrMultimeter = &multimeter;
```

wskaźnik `ptrMultimeter` uzyskuje adres obiektu klasy bazowej, co w konsekwencji pozwala na wykorzystanie jej do wywołania funkcji `showMeasureResult()` z klasy bazowej:

```
ptrMultimeter -> showMeasureResult();
```

W analogiczny sposób można dokonać wywołania funkcji `showMeasureResult()` klasy pochodnej, posługując się adresem obiektu klasy pochodnej. Ponieważ funkcja `showMeasureResult()` jest w swoich klasach funkcją wirtualną, zatem w trakcie działania programu decyzja o tym, która wersja tej funkcji jest aktualnie wywoływana, zapada na podstawie określenia typu egzemplarza klasy aktualnie wskazywanego przez wskaźnik `ptrMultimeter` wskazujący na klasę bazową.

Każda klasa powinna deklarować destruktor wirtualny [21]. Destruktory wirtualne pozwalają odpowiednio usunąć obiekt wskazywany wskaźnikiem do

klasy, po której on dziedziczy (klasy-przodka). Jeżeli klasa ma chociaż jedną metodę wirtualną, powinna mieć wirtualny destruktor; jednak ogólnie rzecz biorąc nic nie stoi na przeszkodzie, żeby destruktor był zawsze wirtualny. Co prawda cierpi na tym nieznacznie wydajność programu, ale za to kod jest bezpieczniejszy i mniej podatny na błędy. Jeżeli żądamy, aby niemożliwym było generowanie wyjątku z destruktora należy opatrzyć go klauzulą `throw()`, np.:

```
virtual ~TMultimeter() throw() {};
```

Podczas pracy z funkcjami wirtualnymi możliwe jest również wykorzystywanie parametru jako odwołania do klasy bazowej. Odpowiednio konstruowane odwołania do klasy bazowej umożliwiają wywołanie funkcji wirtualnej z jednoczesnym przekazaniem jej argumentu. Przedstawiony na diagramie z rysunku 3.3 model jest modyfikacją modelu z poprzedniego ćwiczenia. Tak jak pokazano w przykładzie z listingu 3.6 zadeklarowano w nim klasę bazową, dwie klasy pochodne oraz funkcję przeładowaną, zawierającą — poprzez parametr formalny `m` — odwołanie do klasy bazowej:

```
//-----------------------------------------------------------
void showMeasureResult(TMultimeter &m)
{
  m.showMeasureResult();
  return;
}
//-----------------------------------------------------------
```

Dzięki tak skonstruowanemu odwołaniu aktualna wersji funkcji `showMeasureResult()`, która powinna być w danym momencie działania programu wywołana, ustalana jest w głównej funkcji `main()` na podstawie typu, do którego odwołuje się jej parametr aktualny.

Listing 3.6. Odwołanie do klasy polimorficznej

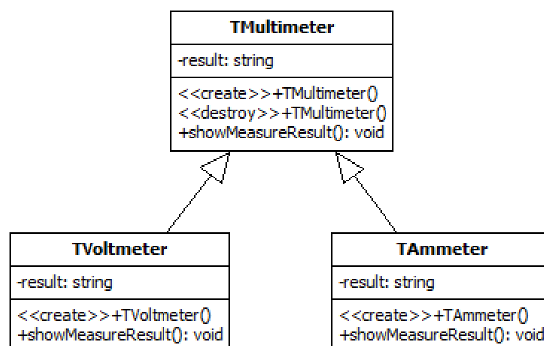
```
#include <iostream>
using namespace std;

class TMultimeter {//miernik uniwersalny
  private:
    string result;
  public:
    TMultimeter() {}
    virtual ~TMultimeter(){};  //destruktor wirtualny
    virtual void showMeasureResult() {
      result = "215 V";
      cout << "Pomiar miernika uniwersalnego: "<< result
```

```
                                                        << endl;
    }
};
//-----------------------------------------------------------
class TVoltmeter : public TMultimeter {//woltomierz
  private:
    string result;
  public:
    TVoltmeter(){}
    void showMeasureResult() {
      result = "215.5 V";
      cout << "Pomiar woltomierza: " << result << endl;
    }
};
//-----------------------------------------------------------
class TAmmeter : public TMultimeter {//woltomierz
  private:
    string result;
  public:
    TAmmeter(){}
    void showMeasureResult() {
      result = "0.5 A";
      cout << "Pomiar amperomierza: " << result << endl;
    }
};
//-----------------------------------------------------------
void showMeasureResult(TMultimeter &m)
// odwołanie do klasy bazowej
{
  m.showMeasureResult();
  return;
}
//-----------------------------------------------------------
int main()
{
  TMultimeter multimeter;
  TVoltmeter voltmeter;
  TAmmeter ammeter;
  // wywołanie funkcji showMeasureResult() klasy
  // TMultimeter
  showMeasureResult(multimeter);
  // wywołanie funkcji showMeasureResult() klasy TVoltmeter
  showMeasureResult(voltmeter);
  // wywołanie funkcji showMeasureResult() klasy TAmmeter
  showMeasureResult(ammeter);
```

```
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Pomiar miernika uniwersalnego: 215 V
Pomiar woltomierza: 215.5 V
Pomiar amperomierza: 0.5 A
```



Rysunek 3.3. Woltomierz i amperomierz są specjalnymi formami miernika uniwersalnego

Bardzo często funkcje zawierające odwołania do klas polimorficznych mają (chociaż niekoniecznie) takie same nazwy, jak funkcje wirtualne względem danej klasy. Chociaż funkcje te mogą mieć takie same nazwy, nie należy utożsamiać ich z funkcjami przeładowanymi. Pomiędzy konstrukcją funkcji przeładowywanych i ponownym definiowaniem funkcji wirtualnych istnieją poważne różnice, np. prototypy funkcji wirtualnych muszą być identyczne, funkcje przeładowane zaś mają różną liczbę lub typ parametrów.

Wskaźniki do wszystkich metod wirtualnych deklarowanych w klasie bazowej oraz klasach pochodnych przechowywane są w tablicy metod wirtualnych VMT (ang. *Virtual Method Table*). Każda zdefiniowana klasa polimorficzna i abstrakcyjna ma swoją własną tablicę VMT. Tablice takie zawierają również klasy, w których jawnie nie zdefiniowano funkcji wirtualnej — wystarczy, że klasa dziedziczy je po swojej klasie bazowej. Każda tablica VMT wypełniona jest listą wszystkich metod wirtualnych właściwych swojej klasie, co w konsekwencji pozwala C++ kompilować odwołania do nich jako bardzo szybkie przeszukiwanie odpowiedniej tablicy VMT. Warto zdawać sobie sprawę z faktu, iż ze względu na swoją specyfikę tablice VMT doskonale nadają się do identyfikowania poszczególnych klas. Również deklarowane odwołania do poszczególnych klas są w rzeczywistości wskaźnikami do VMT danej klasy.

3.5. Obiekty funkcyjne w hierarchii dziedziczenia

Obiekty funkcyjne są specyficznymi wyrażeniami (idiomami) językowymi stosowanymi w programach C++. Po względem składniowym obiekty funkcyjne są bytami zachowującymi się podobnie jak funkcje, przy czym mogą być kreowane i udostępniane innym częściom programu podobnie jak standardowe obiekty. W implementacji C++ obiekt funkcyjny jest wystąpieniem klasy, w której zdefiniowano operator wywołania funkcji `()`. Jako idiom języka C++ obiekt funkcyjny może być wykorzystywany wszędzie tam, gdzie istnieje konieczność używania wskaźników do funkcji składowych klasy. Manipulowanie obiektami funkcyjnymi jest realizacją nowoczesnej idei programowania obiektowego, zgodnie z którą należy operować wyłącznie obiektami reprezentującymi byty fizyczne lub/i abstrakcyjne.

W przykładach z listingów 3.2 oraz 3.3 symulowanie zmian stanu urządzenia (zasilacza) było możliwe dzięki wykorzystaniu wskaźników do funkcji składowych klasy `TPowerSupply`. Podejście taki jest zgodne ze starszymi standardami języka C++. Należy być świadomym faktu, iż najnowsze wersje standardu języka C++ inaczej adresują wskaźniki do struktur danych a inaczej wskaźniki do funkcji [21]. W konsekwencji wynik porównywania wskaźników do funkcji może być nieokreślony i zależny od aktualnej implementacji.

Aby uniknąć tego typu dwuznaczności zaproponujemy odmienne rozwiązanie opierające się na przedefiniowaniu struktury logicznej dziedziny problemu. Na rysunku 3.4. pokazano statyczny diagram klas, gdzie wymodelowano klasę bazową `TPowerSupply`, w której `switchOnOff()` nie jest wskaźnikiem do funkcji składowej klasy, lecz wirtualną funkcją składową, która również powinna posiadać swoją implementację w klasie dziedziczącej. Ponadto, zdefiniowano obiekt funkcyjny poprzez przeładowanie operatora wywołania funkcji `operator()`. Obiekt funkcyjny symuluje włącznik zasilacza.

Przedstawiona w przykładzie z listingu 3.7 implementacja diagramu z rysunku 3.4 znacznie upraszcza proces konstruowania anonimowego obiektu zasilacza sprowadzając całą operację do jednej linii kodu:

```
TPowerSupply *ps= new TPowerSupply(option, true);
```

W klasie `TPowerSupply`, której wystąpieniami są obiekty funkcyjne, zdefiniowano funkcję wirtualną `switchOnOff()`, przedefiniowaną w klasie pochodnej. Wskaźnik `powerSupply` jest inicjowany konstruktorem klasy `TConcretePowerSupply`, w którym funkcja wirtualna `switchOnOff()` wybiera odpowiednią funkcję składową: `off()` lub `on()` „wyłączającą” lub „włączającą” urządzenie.

Listing 3.7. Przykład implementacji obiektu funkcyjnego

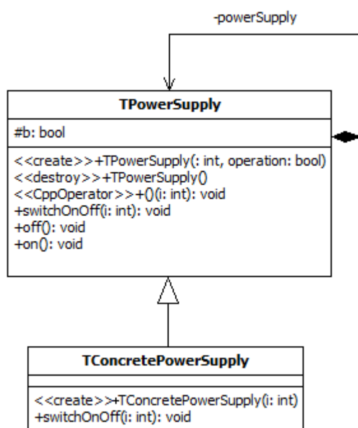
```
#include <iostream.h>
using namespace std;
```

```
class TPowerSupply {
  public:
    TPowerSupply(int, bool operation = true);
    virtual ~TPowerSupply() { };
    virtual void operator() (int i) {
      switchOnOff(i);
    }
    virtual void switchOnOff(int i) {
      powerSupply->switchOnOff(i);
    }
    void off() { cout << "Current OFF" << endl; }
    void on() { cout << "Current ON" << endl; }
  protected:
    bool b;
  private:
    TPowerSupply* powerSupply;
};
//-----------------------------------------------------------
class TConcretePowerSupply: public TPowerSupply{
  public:
    TConcretePowerSupply(int i): TPowerSupply(i, false) {
       cout<<"Konstruowanie konkretnego zasilacza...\n";
    }
    void switchOnOff(int i) {
       switch (i) {
          case 0: off();
              break;
          case 1: on();
              break;
        }
    }
};
//-----------------------------------------------------------
//Konstruktor klasy TPowerSupply
TPowerSupply::TPowerSupply(int i, bool operation) {
  if(b != i)
    this->b = i;
  if(operation == true) {
    powerSupply = new TConcretePowerSupply(i);
    cout << "Zasilacz w stanie = "<<powerSupply->b<<endl;
  }
}
//-----------------------------------------------------------
int main()  {
```

```
  int option;
  while(true) {
    cout << "Enter 0-OFF, 1-ON, 2-exit:\n\n";
    cin >> option;
    if (option == 2)
      exit(EXIT_SUCCESS);
      else {
        TPowerSupply *ps= new TPowerSupply(option, true);
        delete ps;
      }
  }
  return 0;
}
```

Wynik działania programu:

```
Enter 0-OFF, 1-ON, 2-exit:

1
Konstruowanie konkretnego zasilacza...
Zasilacz w stanie = 1
Enter 0-OFF, 1-ON, 2-exit:

0
Konstruowanie konkretnego zasilacza...
Zasilacz w stanie = 0
Enter 0-OFF, 1-ON, 2-exit:
```



Rysunek 3.4. Klasa bazowa modelująca obiekt funkcyjny

3.6. Statyczny polimorfizm

Głównym sposobem realizowania polimorfizmu w implementacjach C++ (a także w innych obiektowych językach programowania) jest korzystanie funkcji wirtualnych. Warto jednak zauważyć, iż przy rozbudowanych klasach liczba wykorzystywanych funkcji wirtualnych może osiągnąć pokaźne rozmiary, co skutkuje spadkiem wydajności programu, np. poprzez zwiększanie się rozmiaru

pojedynczego obiektu klasy. Rozmiar ten zwiększa się wraz ze wzrostem liczby używanych wskaźników, co może mieć duże znaczenie, w przypadku gdy gromadzone są duże ilości obiektów.

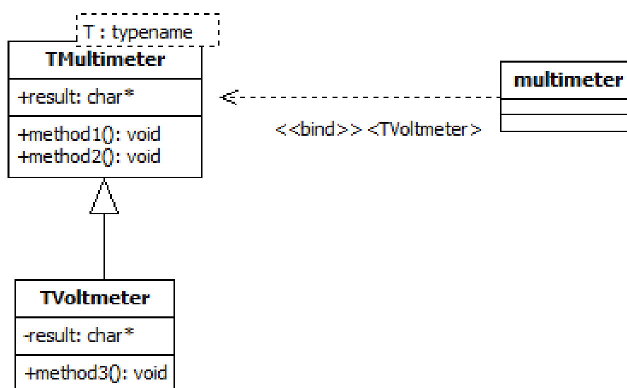
Sposobem na rozwiązanie tego problemu jest wykorzystanie mechanizmów określanych mianem statycznego polimorfizmu lub symulowanego późnego wiązania (ang. *simulated late binding*). Podstawą statycznego polimorfizmu jest opisany przez Jamesa Copliena wzorzec CRTP (ang. *curiously recurring template pattern*), który został spopularyzowany m.in. przez WTL (ang. *Windows Tempale Library*) oraz ATL (ang. *Active Template Library*) [23]. Pozwala on na przekazanie za pomocą parametru uogólnionego do klasy bazowej informacji o typie klasy pochodnej. Mechanizm ten opiera się na jednym z dwóch idiomów:

```
class derived : public base<derived> {};
```

lub

```
template<typename base>
class derived : public base {};
```

Pokazane wyżej konstrukcje umożliwiają wykorzystanie przez klasę bazową informacji o typie klasy pochodnej do wywołania odpowiedniej metody. Przedstawia to przykład z listingu 3.8 będący implementacją diagramu klas z rysunku 3.5.



Rysunek 3.5. Woltomierz jest typem pochodnym szablonu miernik uniwersalny

Listing 3.8. Implementacja diagramu 3.5

```
#include <iostream>
using namespace std;
template<typename T>
class TMultimeter {
```

```
  private:
    char* result;
  public:
    void method1() {
      T *ptr = static_cast<T*>(this);
      return ptr->method3();
    }
    void method2(){
      result = "215.5 V";
      cout<<"Pomiar miernika uniwersalnego: "<<result<<endl;
    };
};
//-----------------------------------------------------------
class TVoltmeter : public TMultimeter<TVoltmeter> {
  private:
     char* result;
  public:
     void method3() {
       result = "215 V";
       cout<<"Pomiar woltomierza: "<<result<<endl;
     };
};

//-----------------------------------------------------------
int main() {
  TMultimeter<TVoltmeter> multimeter;
  //lub
  //TVoltmeter multimeter;
  multimeter.method1();
  multimeter.method2();
  cin.get();
  return 0;
}
```

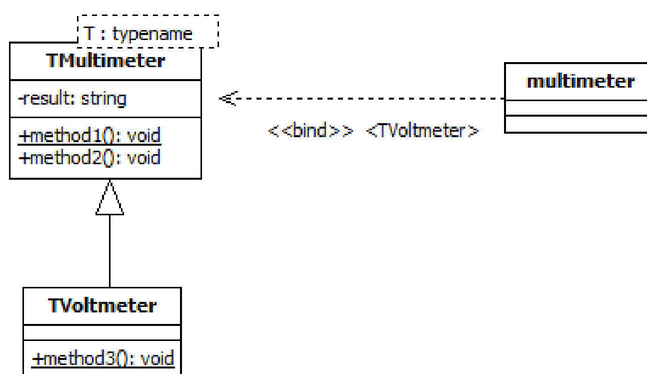
Wynik działania programu:

```
Pomiar woltomierza: 215 V
Pomiar miernika uniwersalnego: 215.5 V
```

Klasa bazowa `TMultimeter` jest w stanie wywołać funkcje klasy pochodnej dzięki temu, że ma informację o nich. Po rzutowaniu wskaźnika `this` na odpowiedni typ `T`, zostanie wywołana funkcja składowa właśnie tego typu. Rzutowanie odbywa się za pomocą operatora `static_cast`, który wykonuje tzw. rzutowanie niepolimorficzne, co między innymi oznacza możliwość wykonania każdej standardowej konwersji typów bez konieczności sprawdzania jej poprawności [22]. W omawianym przykładzie, dzięki operatorowi

static_cast, uzyskamy wskaźnik ptr do typu T będącego parametrem szablonu ze wskaźnika this do typu bazowego (TMultimeter*). Warto zauważyć, iż w prezentowanym kodzie nie są wykorzystywane funkcje wirtualne oraz inne dynamiczne elementy implementacji języka. Dzięki temu mogące pojawiać się różnego rodzaju problemy związane z wywoływaniem wskaźników do funkcji stają się nieistotne.

Jedną z najciekawszych rzeczy na jakie pozwala opisany wyżej mechanizm jest polimorfizm statycznych funkcji składowych klasy. Na rysunku 3.6 pokazano diagram analogiczny do poprzedniego, z tym że aktualnie wykorzystane są statyczne funkcje składowe.



Rysunek 3.6 . Polimorfizm z operacjami statycznymi

Listing 3.9 Implementacja diagramu 3.6

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

template<typename T>
class TMultimeter {
  private:
    string result;
  public:
    static void method1() {
      return T::method3();
    }
    void method2(){
      result = "215.5 V";
      cout << "Pomiar miernika uniwersalnego: "<< result
                                                << endl;
    };
};
//-----------------------------------------------------------
class TVoltmeter : public TMultimeter<TVoltmeter> {
```

```
  public:
    static void method3() {
      string result = "217 V";
      cout << "Pomiar woltomierza: "<< result << endl;
    };
};
//-----------------------------------------------------------
int main() {
  TMultimeter<TVoltmeter> multimeter;
  multimeter.method1();
  multimeter.method2();
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Pomiar woltomierza: 217 V
Pomiar miernika uniwersalnego: 215.5 V
```

Dzięki implementacji pokazanej na listingu 3.9, statyczne funkcje składowe klasy bazowej zdolne są do wywoływania statycznych funkcji składowych klasy pochodnej. Zabieg ten pozwala na korzystanie w statycznych funkcjach składowych z zalet polimorfizmu, który jest podstawowym mechanizmem programowania zorientowanego obiektowo.

Testując przykłady z listingów 3.8 oraz 3.9 bez trudu można zauważyć, iż, główną zaletą wykorzystania statycznego polimorfizmu jest możliwość nieskomplikowanego wywoływania funkcji składowych klasy pochodnej z wnętrza klasy bazowej. Ta niewątpliwa zaleta ma też i drugą stronę: nie można w bezpieczny sposób tworzyć obiektów różnych klas pochodnych dziedziczących po tych samych klasach bazowych, gdyż w tej sytuacji klasy bazowe są zupełnie różnymi klasami. Dlatego też jakiekolwiek rzutowania na klasę bazową są bezcelowe, gdyż nie można pozbyć się informacji o klasie pochodnej. Sytuację tę obrazuje model klas z rysunku 3.7. Listing z przykładu 3.10 jest jego implementacją.

Listing 3.10. Implementacja diagramu 3.7

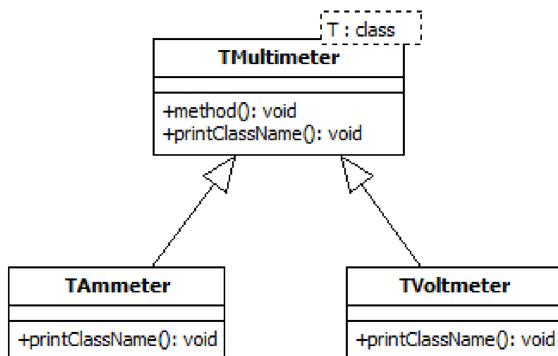
```
#include <iostream>
using namespace std;

template <class T>
class TMultimeter
{
  public:
    void method()  {
        T* ptr = static_cast<T*>(this);
```

```
        ptr->printClassName();
    }

    void printClassName() {
      cout << "Operacje klasy TMultimeter\n";
    }
};
//------------------------------------------------------------
class TAmmeter : public TMultimeter<TAmmeter>
{
  public:
    void printClassName() {
       cout << "operacje klasy TAmmeter\n";
    }

};
//------------------------------------------------------------
class TVoltmeter : public TMultimeter<TVoltmeter>
{
  public:
    void printClassName() {
       cout << "operacje klasy TVoltmeter\n";
    }
};
//------------------------------------------------------------
int main() {
  TMultimeter<TAmmeter> ammeter;
  TMultimeter<TVoltmeter> voltmeter;
  ammeter.method();
  voltmeter.method();
  cin.get();
  return 0;
}
```

Wynik działania powyższego programu:

```
operacje klasy TAmmeter
operacje klasy TVoltmeter
```

udowadnia niemożliwość uzyskania informacji zawartych w klasie bazowej.

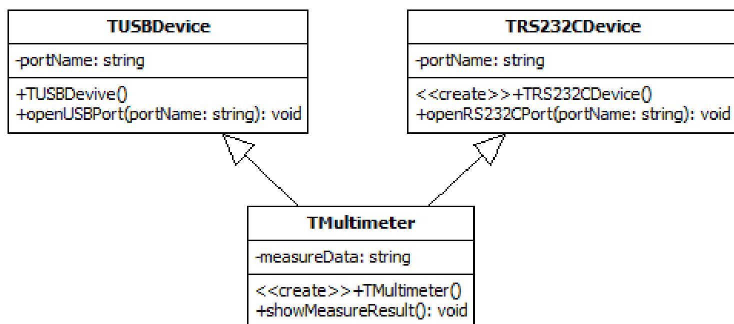


Rysunek 3.7. Statyczny polimorfizm. Dziedziczenie różnych klas pochodnych po wspólnej klasie bazowej

3.7. Dziedziczenie wielokrotne

Dziedziczenie wielokrotne (ang. *multiple inheritance*) nazywane także dziedziczeniem wielobazowym lub wielodziedziczeniem polega na dziedziczeniu po więcej niż jednej klasie bazowej. Dziedziczenie wielokrotne implementowane jest przez niewielką liczbę języków programowania. W większości obiektowych języków programowania (Java, C#, Object Pascal) dopuszczalne jest wyłącznie dziedziczenie jednokrotne, zaś do uzyskania efektu, który w C++ osiąga się poprzez dziedziczenie wielokrotne używa się interfejsów.

Wielodziedziczenie znajduje zastosowanie w sytuacjach, gdy żądamy aby dany obiekt był w tym samym programie obiektem wielotypowym. Posłużmy się przykładem miernika uniwersalnego, którego układy elektroniczne mogą transmitować dane pomiarowe zarówno w standardzie RS 232C jak i USB [24,25]. Na rysunku 3.8 zaprezentowano statyczny diagram klas modelujący sytuację, w której klasa `TMultimeter` posiada wszystkie atrybuty i operacje swoich klas bazowych `TUSBDevice` oraz `TRS232CDevice`.



Rysunek 3.8. Dziedziczenie wielobazowe

Dzięki zastosowaniu wielokrotnego dziedziczenia obiekt klasy `TMultimeter` będzie wielotypowy, gdyż klasa `TMultimeter` łączy dwie zupełnie niezależne klasy opisujące różne protokoły transferu danych. Transfer oparty na protokole USB nie ma nic wspólnego ze standardową transmisją danych opisaną protokołem RS 232C, a więc ich łączenie z punktu widzenia logicznego i programistycznego jest bezpieczne. W odniesieniu do powyższego modelu, w kodzie programu mogą pojawić się zapisy pokazane na listingu 3.11.

Listing 3.11. Implementacja diagramu 3.8

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

class TUSBDevice {
  private:
    string portName;
  public:
    TUSBDevive() {}
    void openUSBPort(string& portName) {
      this->portName = portName;
      cout << "Otwarcie portu USB: "<< portName << endl;
    }
};
//-----------------------------------------------------------
class TRS232CDevice {
  private:
    string portName;
  public:
    TRS232CDevice(){}
    void openRS232CPort(string& portName) {
      this->portName = portName;
      cout << "Otwarcie portu RS232C: " << portName << endl;
    }
};
//-----------------------------------------------------------
class TMultimeter : public TUSBDevice, public TRS232CDevice
{
  private:
    string measureData;
  public:
    TMultimeter(){}
    void showMeasureResult() {
      measureData = "0.5 A";
      cout << "Transfer danych: " << measureData << endl;
    }
};
```

```
//---------------------------------------------------------
void TxDUSB(TUSBDevice& usb, string portName)
{
  usb.openUSBPort(portName);
  return;
}
//---------------------------------------------------------
void TxDRS232C(TRS232CDevice& rs, string portName)
{
  rs.openRS232CPort(portName);
  return;
}
//---------------------------------------------------------
int main()
{
  TMultimeter multimeter;
  TxDUSB(multimeter, "\\USB\Dev...");
  TxDRS232C(multimeter, "COM2");
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Otwarcie portu USB: \USBDev...
Otwarcie portu RS232C: COM2
```

upewnia nas, iż obiekt `multimeter` jest jednocześnie typu `TUSBDevice` i `TRS232CDevice`.

Dziedziczenie wielobazowe to metoda dającą w praktyce duże możliwości, jednak należy z niej korzystać z rozwagą. Wielodziedziczenie bardzo dobrze sprawdza się w sytuacjach, w których należy połączyć dwie całkowicie niezależne klasy, jednak znacznie trudniej jest je zastosować wówczas, gdy łączone klasy mają ze sobą coś wspólnego. Przykładowo, tworzenie klasy opisującej miernik uniwersalny z klas opisujących woltomierz i amperomierz może być ryzykowne, ponieważ zarówno woltomierz jak i amperomierz są przyrządami pomiarowymi. Obie klasy mogą nawet dziedziczyć po wspólnej klasie opisującej przyrząd pomiarowy (patrz rys. 3.1).

Kolejną trudność można napotkać przy problemie wspólnej implementacji, tj. w sytuacji, w której dwie klasy bazowe mają wspólnego przodka definiującego obiekt stanowy mający atrybuty, tak jak pokazano to na rysunku 3.9, gdzie wymodelowano klasę bazową `TBase`, dwie klasy pochodne `TDerived1` i `TDerived2` dziedziczące po klasie bazowej i dodatkowo trzecią klasę pochodną `TDervied3`, dziedziczącą elementy publiczne klas `TDerived1` i `TDerived2`. W każdej z klas zdefiniujmy po jednej funkcji zwracającej pewną wartość całkowitą. Przyjęte założenia implementuje listing 3.12.

Listing 3.12. Implementacja diagramu 3.9

```
// Program nie zostanie skompilowany !
#include <iostream>
using namespace std;

class TBase {
  public:
    int i;
    int printBase(){
      cout << "Jestem klasa bazowa" << endl;
      return i;
    }
};
//-------------------------------------------------------
class TDerived1 : public TBase {
  public:
    int j;
    int printD1() {
      cout << "Jestem 1 klasa pochodna" << endl;
      return j;
    }
};
//-------------------------------------------------------
class TDerived2 : public TBase {
  public:
    int k;
    int printD2() {
      cout << "Jestem 2 klasa pochodna" << endl;
      return k;
    }
};
//-------------------------------------------------------
// klasa TDerived3 dziedziczy klasy TDerived1 i TDerived2,
// i zawiera dwie kopie klasy TBase
class TDerived3 : public TDerived1, public TDerived2 {
  public:
    int l;
    int printD3() {
      cout << "Jestem 3 klasa pochodna" << endl;
      return l;
    }
};
//-------------------------------------------------------
int main()
```

```
{
  TDerived3 object;
  object.i = 100;
  object.j = 200;
  object.k = 300;
  object.l = 400;
  cout << object.printD1() << endl;
  cout << object.printD2() << endl;
  cout << object.printD3() << endl;
  cin.get();
  return 0;
}
```

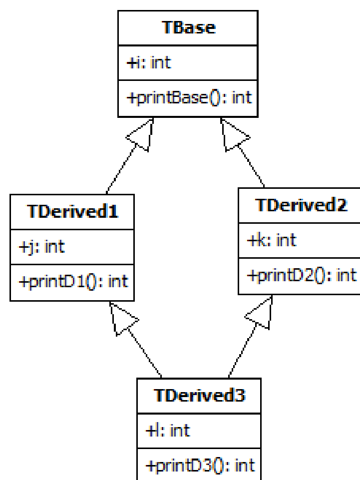
Podczas próby uruchomienia powyższego programu spotka nas przykra niespodzianka polegająca na tym, że program się po prostu nie skompiluje. Wynika to z faktu, iż jego konstrukcja jest niejednoznaczna, ponieważ np. wywołanie:

```
object.i = 100;
```

jest dla kompilatora C++ niejednoznaczne:

```
[C++ Error] Unit.cpp(44): E2014 Member is ambiguous:
                                'TBase::i' and 'TBase::i'
```

z tego powodu, że każdy egzemplarz klasy `TDerived3` zawiera dwie kopie elementów składowych klasy `TBase`.



Rysunek 3.9. Idea wielobazowego dziedziczenia

Ponieważ w tej sytuacji istnieją dwie kopie atrybutu `i` (deklarowanego w klasie bazowej), kompilator nie ma najmniejszej wiedzy na temat, którą kopię atrybutu ma wykorzystać — tę odziedziczoną przez klasę `TDerived1` czy tę z klasy `TDerived2`.

Jeżeli dwie lub większa liczba klas dziedziczy z tej samej klasy bazowej, możemy zapobiec sytuacji, w której kopia klasy bazowej jest powielana w klasach potomnych w sposób niekontrolowany. Istnieje kilka sposobów, aby przeciwdziałać takiemu „samopowielaniu się” klasy bazowej. Najprostszym rozwiązaniem jest zastosowanie koncepcji wirtualnego dziedziczenia po klasie bazowej, tak jak pokazano to na listingu 3.13.

Listing 3.13. Implementacja wirtualnych klas bazowych

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

class TBase {
  //...
};
//--------------------------------------------------------
class TDerived1 : virtual public TBase {
  //...
};
//--------------------------------------------------------
class TDerived2 : virtual public TBase {
  //...
};
//--------------------------------------------------------
class TDerived3 : public TDerived1, public TDerived2 {
  //...
};
//--------------------------------------------------------
int main()
{
  TDerived3 object;
  //...
  cin.get();
  return 0;
}
```

Testując przedstawiony kod, natychmiast zauważymy, iż w klasie `TDerived3` istnieje teraz już tylko jedna kopia klasy bazowej, gdyż klasy `TDerived1` i `TDerived2` dziedziczą klasę bazową jako klasę wirtualną, co skutkuje utworzeniem tylko jednej kopii klasy `TBase`. Klasy wirtualne należy wykorzystywać w C++ tylko wtedy, gdy w programie istnieje konieczność

wielokrotnego dziedziczenia klas. Gdy klasa bazowa definiująca obiekt stanowy (mający atrybuty) dziedziczona jest jako wirtualna, w programie tworzona jest tylko jedna jej kopia. Przez pojęcie wielokrotnego dziedziczenia zawsze rozumiemy sytuację, w której jedna klasa jednocześnie dziedziczy po większej liczbie klas.

3.8. Informacja czasu wykonywania

Nowoczesne środowiska programisty C++, opierają się na informacjach udostępnianych przez kompilator. Informacja czasu wykonywania RTTI (ang. *Run-Time Type Information*) opisuje wybrane aspekty wykorzystywania w programie typów danych, włącznie z elastycznymi klasami polimorficznymi i abstrakcyjnymi.

W celu ustalenia typu obiektu w czasie działania programu, w najprostszym przypadku należy do programu włączyć plik nagłówkowy *typeinfo.h* oraz użyć operatora (słowa kluczowego) `typeid(object)`. Słowo `object` oznacza nazwę zmiennej, której typ ma zostać zidentyfikowany. W pliku nagłówkowym *typeinfo.h* deklarowanych jest kilka typów oraz funkcji umożliwiających uzyskanie łatwego dostępu do właściwości danego obiektu. Plik ten w szczególności zawiera definicję klasy `type_info`:

```
class type_info
{
public:
   virtual ~type_info();
   int operator==(const type_info& rhs) const;
   int operator!=(const type_info& rhs) const;
   int before(const type_info& rhs) const;
   const char* name() const;
};
```

Przedefiniowane operatory == oraz `!=` umożliwiają porównywanie typów danych. Funkcja składowa `before()` zwraca wartość prawdziwą, jeżeli wywołujący ją obiekt `T1` znajduje się przed obiektem `T2` użytym jako argument aktualny, zgodnie z kolejnością ich porównywania:

```
typeid ( T1 ).before(typeid( T2 ));
```

Funkcji tej używamy głównie w odniesieniu do własności lub funkcji wewnętrznych. Funkcja składowa `name()` zwraca wskaźnik do szukanej nazwy typu. Jeżeli funkcja `name()` została użyta z operatorem `typeid()`, którego argumentem jest zmienna zadeklarowana jako wskaźnik do bazowej klasy polimorficznej, to zwróci ona typ aktualnie wskazywanej klasy (lub jej egzemplarza — nawet jeżeli jest to egzemplarz którejś z klas pochodnych).

Przykład nieskomplikowanej aplikacji konsolowej wykorzystującej operator `typeid()`, funkcje `name()`, `before()` oraz przedefiniowane operatory przedstawiono na listingu 3.14.

Listing 3.14. Praktycznego wykorzystanie elementów klasy `type_info`

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

class TBase {
  private:
    //...
  public:
    TBase(){};
    virtual ~TBase(){};
    //tworzy klasę polimorficzną
    virtual void func(){};
};
//-----------------------------------------------------------
class TDerived: public TBase
{
  public:
    TDerived(){};
  //...
};
//-----------------------------------------------------------
int main()
{
  TDerived *ptrDerived = new TDerived;
  cout << " Wskaźnik ptrDerived wskazuje na typ: ";
  cout << typeid(*ptrDerived).name() << endl;
  cout << " Funkcja func() jest typu: ";
  cout << typeid(ptrDerived->func()).name() << endl;

  cout << typeid(TBase).name();
  cout << " jest przodkiem klasy " <<
          typeid(*ptrDerived).name() << ": " <<
          (typeid(TBase).before(typeid(TDerived))
          ? true : false) << endl;

  try {
    if (typeid(*ptrDerived) == typeid(TDerived )) {
      cout << "Nazwa klasy: " << typeid(*ptrDerived).name()
          <<  "\nTyp funkcji składowej: " <<
               typeid(ptrDerived->func()).name();
```

```
    }
    if (typeid(*ptrDerived) != typeid(TBase))
      cout << "\nWskaźnik nie odpowiada typowi klasy\
                 bazowej\n";
  }
  catch (bad_typeid) {
      cout << "Błędny rezultat wykonania wyrażenia typeid";
  }
  delete ptrDerived;
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
 Wskaznik ptrDerived wskazuje na typ: TDerived
 Funkcja func() jest typu: void
TBase jest przodkiem klasy TDerived: 1
Nazwa klasy: TDerived
Typ funkcji skladowej: void
Wskaznik nie odpowiada typowi klasy bazowej
```

Testując powyższy algorytm, musimy zauważyć, że operator `typeid()` użyto ze wskaźnikiem do bazowej klasy polimorficznej, co w konsekwencji pozwoliło ustalić aktualną nazwę typu wskazywanego przez wskaźnik `ptrDerived`. W analogiczny sposób ustalono typ funkcji tworzącej klasę polimorficzną. Informację czasu wykonania należy stosować w odniesieniu do każdej bazowej klasy polimorficznej wchodzącej w skład projektu, nawet jeśli jej definicja znajduje się w odrębnych plikach nagłówkowych. Również dosyć często w celu prześledzenia poprawnego ustalania w czasie wykonywania programu typu obiektu za pomocą instrukcji `typeid` stosujemy prosty mechanizm przechwytywania wyjątków (patrz Rozdział 7). Warto pamiętać, iż kiedy operand `typeid()` zawiera wskaźnik pusty, zostaje wygenerowany wyjątek `bad_typeid`, który wystarczy odpowiednio przechwycić, np. instrukcją `catch`.

3.8.1. Operator dynamic_cast

Operator `dynamic_cast` wykonuje rzutowanie czasu wykonywania, co oznacza, iż poprawność tej operacji zawsze sprawdzania jest w trakcie działania programu. W przypadku, gdy operacja rzutowania typów nie jest możliwa do zrealizowania, całość wyrażenia, w którym występuje omawiany operator, przyjmuje wartość zerową. Ponieważ operator `dynamic_cast` wykonuje rzutowanie czasu wykonywania, należy używać go głównie do przekształcania typów polimorficznych. Oznacza to, iż np. w przypadku, gdy pewna klasa `TDerived` jest klasą potomną innej klasy polimorficznej `TBase`, to posługując się operatorem `dynamic_cast`, zawsze można przekształcić wskaźnik `ptrDerived` do typu `TDerived` na wskaźnik `ptrBase` do typu `TBase`.

Jako przykład rozpatrzmy klasę polimorficzną `TBase` oraz dziedziczącą po niej klasę `TDerived`, tak jak pokazano to na listingu 3.15.

Listing 3.15. Polimorficzne rzutowanie typów

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

class TBase {
  public:
    TBase(){};
    virtual ~TBase(){};
    //tworzy klasę polimorficzną
    virtual void func() {}
};
class TDerived: public TBase {
  public:
    TDerived(){};
  //...
};
//--------------------------------------------------------
int main () {
  try {
     TBase* ptrBase = new TDerived;
     TBase* ptrBase1 = new TBase;
     TDerived* ptrDerived;

     ptrDerived = dynamic_cast<TDerived*>(ptrBase);
     if (ptrDerived == 0)
        cout << "1. Wskaźnik pusty" << endl;
     ptrDerived = dynamic_cast<TDerived*>(ptrBase1);
     if (ptrDerived == 0)
       cout << "2. Wskaźnik pusty" << endl;
  } catch (exception& my_ex) {
      cout << "Exception: " << my_ex.what();
  }
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
2. Wskaznik pusty
```

Śledząc powyższe zapisy, musimy zauważyć, iż użycie operatora `dynamic_cast` pozwoliło na odszukanie wszystkich istniejących wskaźników

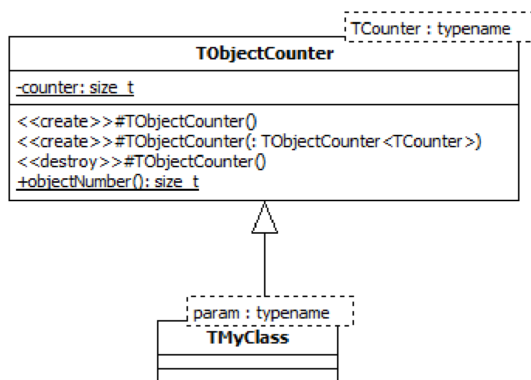
do klasy bazowej. W przypadku, gdy wartość wyrażenia:

```
ptrDerived = dynamic_cast<TDerived*>(ptrBase);
```

będzie równa zeru, oznaczać to będzie, iż operacja rzutowania nie jest możliwa do przeprowadzenia.

3.8.2. Zliczanie obiektów wielokrotnych

W praktyce często stajemy przed koniecznością bieżącego monitorowania liczby obiektów wielokrotnych tworzonych w trakcie wykonywania programu. W tym celu często wykorzystujemy mechanizmy RTTI udostępniane przez kompilator. Jednak są to operacje czasochłonne i niewygodne do stosowania w trakcie pracy z szablonami klas. W tym kontekście można wykorzystać mechanizmy zaimplementowane we wzorcu CRTP konstruując szablon pokazany na rysunku 3.10, a którego implementację przedstawiono w przykładzie z listingu 3.16.



Rysunek 3.10. Klasa parametryzowana licznika obiektów

Listing 3.16. Implementacja diagramu 3.10

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

template <typename TCounter>
class TObjectCounter {
  private:
   static size_t counter; //statyczny licznik
                          //istniejących obiektów
  protected:
    TObjectCounter() { //konstruktor
     ++TObjectCounter<TCounter>::counter;
    }
```

```
   //konstruktor kopiujący
   TObjectCounter (TObjectCounter<TCounter> const&) {
     ++TObjectCounter<TCounterType>::counter;
   }
   ~TObjectCounter() {  //destruktor
     --TObjectCounter<TCounter>::counter;
   }
  public:
   // zwraca liczbę istniejących obiektów
   static size_t objectNumber() {
     return TObjectCounter<TCounter>::counter;
   }
 };
//-----------------------------------------------------------
// wartość początkowa licznika obiektów
template <typename TCounterType>
 size_t TObjectCounter<TCounterType>::counter = 0;
 //----------------------------------------------------------
template <typename param>
class TMyClass : public TObjectCounter<TMyClass<param> > {
  //...
};
//-----------------------------------------------------------
int main() {
   TMyClass<int> i1, i2, i3;
   TMyClass<double> d1, d2;
   cout << "liczba obiektów typu TMyClass<int>: "
        << i1.objectNumber() << endl;
   cout << "liczba obiektów typu TMyClass<double>: "
        << d1.objectNumber() << endl;
   cin.get();
   return 0;
 }
```

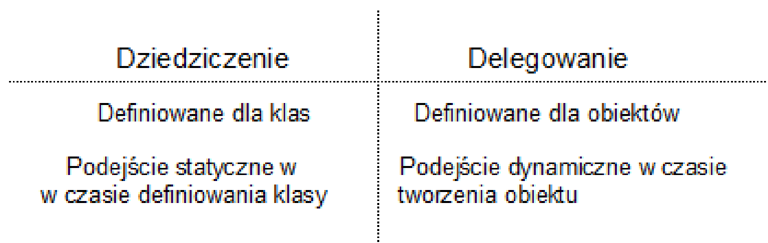
Wynik działania programu:

```
liczba obiektow typu TMyClass<int>: 3
liczba obiektow typu TMyClass<double>: 2
```

Posługując się pokazanym szablonem `TObjectCounter` w sposób bardzo wydajny można w trakcie działania programu śledzić liczbę obiektów wielokrotnych tworzonych na bazie wybranej klasy parametryzowanej.

3.9. Delegowanie

Dziedziczenie jest jednym z mechanizmów służących do ponownego wykorzystania gotowych funkcji przy definiowaniu nowych funkcji systemu. Właściwe z logicznego punktu widzenia korzystanie z dziedziczenia może jednak sprawiać poważne trudności ze względu na to, iż implikuje ścisły logiczny związek pomiędzy klasą bazową i klasami pochodnymi. Praktyka programistyczna wskazuje, iż w bardzo wielu przypadkach wygodniej jest wykorzystywać mniej restrykcyjne podejścia. Jednym z takich podejść definiowanych na poziomie obiektów jest delegowanie. Na rysunku 3.11 pokazano zestawienie dwóch podstawowych mechanizmów rozszerzania funkcjonalności systemu.



Rysunek. 3.11. Dwa podstawowe sposoby rozszerzania funkcjonalności systemu.

Mechanizm wiążący funkcję oraz kontekst jej wywołania nazywamy delegowaniem. Jest on powszechnie stosowany w platformie .NET (C#, VB), zaś odpowiednie rozszerzenia dla języka C++ wprowadziła także firma Borland w środowisku C++Builder (polecenie `__closure`) [22].

Języki z platformy .NET automatycznie tworzą delegatów, gdy przekazywana jest jako argument operacji metoda obiektu. Delegat (ang. *delegate*) jest typem, którego przeznaczeniem jest referencja (wskazanie) na metodę konkretnego obiektu, a nie na funkcję składową klasy. Od chwili przypisania metody do delegacji – zachowuje się ona jak każda inna metoda. Do delegata może być przypisana dowolna metoda, która ma identyczną z delegatem sygnaturę oraz typ zwracanej wartości. Dzięki delegatom można programowo zmieniać wywoływane metody, jak również wprowadzać nowy kod do istniejących klas, jeżeli tylko znana jest sygnatura delegacji. Zdolność parametryzacji czyni delegata idealnym instrumentem dla funkcji typu `callback()` oraz w obsłudze zdarzeń sterujących pracą interfejsu użytkownika.

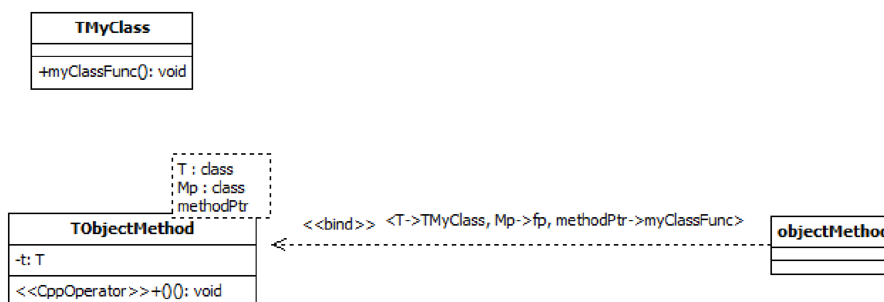
Delegowanie to metoda zapewniająca bardzo duże możliwości dynamicznego strukturalizowania systemu. Umożliwia budowanie hierarchii, której elementami nie są klasy, ale obiekty [26]. Zamiast wykorzystywać dziedziczenie, nakazujemy obiektowi delegowanie odpowiedzialności do innego obiektu w momencie jego konstruowania. Delegacja daje podobne możliwości jak dziedziczenie, z tym że w trakcie delegowania tworzona jest hierarchia obiektów, a nie klas, i hierarchia ta może zostać zmodyfikowana w dowolnym momencie działania programu.

3.9.1. Delegowanie z pomocą obiektów funkcyjnych

Możliwości przedefiniowywania operatorów w C++ dla własnych typów obiektów sprawiają, że obiekty te mogą zachowywać się w bardzo różny sposób. Mogą na przykład „udawać" pewne wbudowane konstrukcje językowe, nierzadko wykonując ich zadania lepiej i wygodniej. Przykładów na to można podać co najmniej kilka. Poniżej opisano najważniejszy z nich.

Obiekt może zachowywać się jak funkcja, czyli udostępniać możliwość wywołania siebie z określonymi parametrami. Takie twory często nazywa się funktorami lub obiektami funkcyjnymi i są standardowo używane podczas pracy z zasobami biblioteki STL. Działają one przy tym w bardzo prosty sposób, zwyczajnie przeciążając operator wywołania funkcji `()`. Jest on na tyle elastyczny, że może przyjmować dowolne parametry i zwracać dowolne wyniki, co pozwala nawet na stworzenie więcej niż jednego sposobu wywoływania danego obiektu. Obiekty funkcyjne mają składnię wywołania funkcji, ale są też pełnoprawnymi obiektami, mogą więc mieć swój własny typ, stan, konstruktory, destruktory i inne metody, jak również typy stowarzyszone. Te dodatkowe informacje pozwalają na implementowanie wielu ciekawych rozwiązań niedostępnych dla zwykłych funkcji i wskaźników do nich. Jednym z bardziej interesujących zastosowań operatora wywołania funkcji jest implementacja w C++ brakującego w standardzie mechanizmu delegatów, czyli wskaźników do metod obiektów.

Na rysunku 3.12 pokazano jeden ze sposobów konstruowania szablonowego obiektu funkcyjnego `TObjectMethod` z funkcją operatorową `operator()`. Na listingu 3.17 zaprezentowano implementację diagramu. Delegatem jest funkcja operatorowa zawierająca definicję wskaźnika `methodPtr`, który jest wskaźnikiem do funkcji składowej pewnej klasy `T` będącej parametrem uogólnionym klasy `TObjectMethod`. W chwili utworzenia obiektu funkcyjnego `objectmethod()`, tzn, w momencie wpisania parametrów aktualnych do `TObjectMethod`, wskaźnik `methodPtr` będzie wskazywał na funkcję zadeklarowaną w klasie będącej aktualnym parametrem szablonu.



Rysunek 3.12. Generyczny delegat

Listing 3.17. Implementacja diagramu z rysunku 3.12

```
#include <iostream>
```

```
using namespace std;
class TMyClass {
  public:
    void myClassFunc() {
      cout << "Metoda obiektu została wywołana\n"; }
};
//-----------------------------------------------------------
template<class T, class Mp, Mp methodPtr>
class TObjectMethod {
  private:
    T t;
  public:
   void operator()() {
    (t.*methodPtr)();
   }
};
//-----------------------------------------------------------
int main() {
  typedef void (TMyClass::*fp)();
  TObjectMethod<TMyClass, fp, &TMyClass::myClassFunc>
                objectMethod;
  objectMethod(); // Wywołanie metody obiektu
  cin.get();
  return 0;
}
```

Analizując powyższe zapisy, dojdziemy do wniosku, iż użycie składni z funkcją operatorową `operator()` przekształca typ funkcyjny w metodę. Metody składają się z dwóch części: wskaźnika kodu i wskaźnika danych. Pobranie adresu metody jest równoważne z pobraniem wskaźnika kodu metody (przypominającego zwykły wskaźnik do funkcji) i wskaźnika danych będącego referencją do obiektu (lub egzemplarza) klasy, w której została zdefiniowana określona metoda.

Podsumowanie

Programowanie obiektowe jest jedną z metod strukturalizowania programów. W porównaniu z innymi paradygmatami programowania wyróżnia się przede wszystkim stosowaniem mechanizmów opartych na polimorfizmie i dziedziczeniu. Polimorfizm nie jest jednak niepowtarzalną cechą paradygmatu obiektowego, dlatego w tym rozdziale skupiono się na odniesieniu dziedziczenia do innych technik programowania. Omówiono też różne podejścia umożliwiające praktyczne wykorzystanie technik związanych ze śledzeniem informacji czasu wykonywania.

ROZDZIAŁ 4

INTERFEJSY

4.1. Interfejsy a klasy abstrakcyjne 114
4.2. Zliczanie odwołań do interfejsu 117
4.2.1. Identyfikator interfejsu 117
4.3. Delegowanie obiektów poprzez wspólny interfejs 124
4.4. Programowanie obiektowe z użyciem interfejsów 127
4.4.1. Wirtualna realizacja interfejsu C++ 129
4.5. Składniki 131
Podsumowanie 133

4.1. Interfejsy a klasy abstrakcyjne

Jedną z różnic występujących pomiędzy klasami abstrakcyjnymi a interfejsami jest to, że pierwsze z nich pozwalają w systemie na określenie wspólnych zachowań i stanu. Oznacza to, że jeżeli wszystkie klasy pochodne dysponują pewnym identycznym stanem lub zachowaniem, to można go zdefiniować w klasie abstrakcyjnej (klasa abstrakcyjna może mieć atrybuty). Rozróżnienie klas abstrakcyjnych i interfejsów jest szczególnie ważne w takich językach obiektowych jak Java, C# lub Object Pascal, w których niedozwolone jest stosowanie wielodziedziczenia. W celu zachowania uniwersalności pojęć powinniśmy przyjąć, iż nie należy używać klas abstrakcyjnych, jeżeli w rzeczywistości nie jest to niezbędne, gdyż każda klasa może mieć tylko jedną bezpośrednią klasę bazową.

Należy zauważyć, iż istnieje jeszcze inny powód, dla którego projektanci szczególnie wyraźnie postrzegają różnice pomiędzy klasami abstrakcyjnymi a interfejsami (chociaż oba te pojęcia dostarczają mechanizmów korzystania z polimorfizmu). Klasy abstrakcyjne postrzegane są jako wydajny sposób grupowania powiązanych ze sobą elementów systemu. W tym przypadku szczególny nacisk kładzie się na to, w jaki sposób klasy pochodne (konkretne elementy) mają być używane. Oznacza to konieczność ustalenia prawidłowych zasad hermetyzacji implementacji – obiektów świadczących usługi. W praktyce ustalanie takiej odpowiedzialności polega na odwróceniu analizy. Najpierw analizowane są obiekty świadczące usługi (implementacje), zaś w dalszej kolejności określa się możliwe sposoby tworzenia ich abstrakcji, dzięki której obiekty używające tych implementacji nie będą powiązane z żadnymi ich szczegółami. Aby podczas projektowania móc w podobny sposób korzystać z interfejsów, należy odpowiedzieć na pytanie: *Jaki wspólny interfejs powinny posiadać obiekty, aby mogły być używane w identyczny sposób?*

Kolejny zabieg myślowy wykonywany przez projektantów koncentruje się na obiektach, które będą używać klas pochodnych lub obiektach klas realizujących interfejs. Proces ten sprowadza się do określenia interfejsu, jaki powinny mieć obiekty, aby wykorzystywanie ich było najprostsze. W tym celu analizę rozpoczyna się od obiektu użytkownika. W dalszej kolejności używane obiekty dzieli się na mniejsze elementy. Oznacza to, iż w przypadku gdy jeden obiekt musi używać innych obiektów różnych typów (czyli wielu abstrakcji) określa się interfejsy dla każdego z nich. Obiekty dysponujące wspólnym stanem (wspólnymi wartościami atrybutów) lub zachowaniem będą dziedziczyć po klasie abstrakcyjnej. Natomiast wszystkie inne obiekty nie dysponujące wspólnym stanem lub zachowaniem powinny realizować interfejs.

Interfejs jest typem składającym się wyłącznie z funkcji czysto wirtualnych. W odróżnieniu od takich języków obiektowych jak Java, C# czy Object Pascal interfejsy C++ *emulowane* są za pomocą klas abstrakcyjnych całkowicie pozbawionych atrybutów. Podobnie jak w Object Pascalu i Javie, w C++ podczas pisania interfejsu można użyć słowa `interface`, należy jednak pamiętać, że jest to jedynie makrodefinicja (wprowadzona w celu zachowania

zgodności z językiem IDL), której użycie wymaga włączenia pliku nagłówkowego *objbase.h*. Na listingu 4.1 pokazano najprostszy przykład konstruowania interfejsów.

Listing 4.1. Przykładowa implementacja dwóch interfejsów w C++

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

class IA // interfejs IA
{
  public:
    virtual ~IA(){};
    virtual void funIA() = 0; //funkcja interfejsu IA
};
//-----------------------------------------------------------
class IB // interfejs IB
{
  public:
    virtual ~IB(){};
    virtual void funIB() = 0; //funkcja interfejsu IB
};
//-----------------------------------------------------------
// Realizacja interfejsów przez klasę TMyComp
class TMyComp : public IA, public IB
{
  public:
  // Implementacja interfejsu IA
   virtual void funIA();
  // Implementacja interfejsu IB
   virtual void funIB();
};
//-----------------------------------------------------------
void TMyComp::funIA()
{
  cout << "Funkcja interfejsu IA" << endl;
}
void TMyComp::funIB()
{
  cout << "Funkcja interfejsu IB" << endl;
}
//-----------------------------------------------------------
int main()
{
  TMyComp *pMyComp = new TMyComp;
  // wskaźnik do interfejsu IA
```

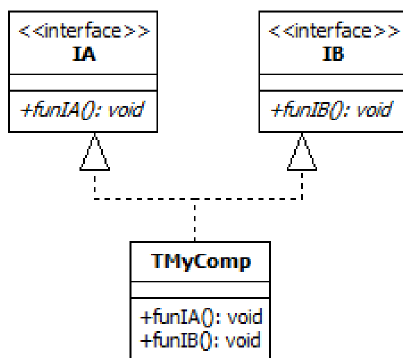
```
  IA *pIA = pMyComp;
  pIA->funIA();
  // wskaźnik do interfejsu IB
  IB *pIB = pMyComp;
  pIB->funIB();
  delete pMyComp;
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Funkcja interfejsu IA
Funkcja interfejsu IB
```

Najprostszy związek pomiędzy klasą a interfejsem nazywamy realizacją (chociaż klasy oraz interfejsy mogą pozostawać również w innych relacjach). Klasa implementuje każdą z operacji interfejsu poprzez deklarację operacji o tej samej nazwie, tych samych argumentach i sposobie wywoływania. C++ automatycznie dopasowuje operacje klasy do operacji danego interfejsu. Nie jest możliwe uzyskanie dostępu do elementów klasy drogą inną niż poprzez funkcje interfejsu. Na rysunku 4.1 pokazano diagram odpowiadający przykładowi z listingu 4.1, w którym klasa `TMyComp` realizuje dwa interfejsy `IA` oraz `IB`.

Nowe interfejsy mogą być deklarowane poprzez dziedziczenie po interfejsach istniejących. Tak jak wszystkie klasy w systemie mogą dziedziczyć po klasie bazowej (klasie `{root}`), wszystkie interfejsy dziedziczą po `IUnknown`. `IUnknown` jest odpowiednikiem innego interfejsu `IInterface` z modułu *system.hpp*. Budując aplikacje niezależne od platformy systemowej, należy używać `IInterface`. `IUnknown` rezerwowany jest dla platform Win32 oraz Win64.



Rysunek 4.1. Realizacja interfejsów

4.2. Zliczanie odwołań do interfejsu

Interfejs `IUnknown` deklaruje trzy metody: `AddRef()`, `Release()` i `QueryInterface()`. Pierwsze dwie zarządzają zliczaniem odwołań do interfejsu w czasie życia obiektu implementującego interfejs. `AddRef()` zwiększa licznik odwołań do interfejsu, natomiast `Release()` zmniejsza go. `QueryInterface()` kontaktuje się z innymi interfejsami, które może implementować obiekt. Funkcja ta zwraca jako wynik 32-bitową liczbę typu `HRESULT`. Jeżeli funkcja zostanie prawidłowo wykonana zwraca wartość dodatnią lub zero (jako `S_OK`), zaś w przeciwnym wypadku wartość ujemną (`E_NOINTERFACE`). Warto pamiętać, iż istnieją dwie makrodefinicje: `SUCCEEDED` oraz `FAILED` zdefiniowane w pliku *WinError.h*, które ułatwiają sprawdzanie wartości `HRESULT`:

```
#define SUCCEEDED(Status) ((HRESULT) (Status) >= 0)
#define FAILED(Status) ((HRESULT) (Status) < 0)
```

Zadaniem funkcji `QueryInterface()` jest zwrócenie wskaźnika do interfejsu o podanym identyfikatorze `GUID`. Do jej użycia niezbędny jest identyfikator interfejsu. W celu wydobycia identyfikatora zawsze można podać nazwę interfejsu. C++ automatycznie konwertuje nazwę interfejsu na jego identyfikator. W celu utworzenia początkowo niezainicjowanego obiektu programista powinien prawidłowo skonstruować własną funkcję `CreateInstance()`.

4.2.1. Identyfikator interfejsu

Pełniejszą realizację interfejsu można uzyskać poprzez wydobycie jego identyfikatora. Typ strukturowy `GUID`, zdefiniowany w pliku nagłówkowym *guiddef.h*, przechowuje globalnie unikatowy identyfikator (ang. *Globally Unique Identifier*).

```
#ifndef GUID_DEFINED
#define GUID_DEFINED
#if defined(__midl)
typedef struct {
    unsigned long  Data1;
    unsigned short Data2;
    unsigned short Data3;
    byte           Data4[ 8 ];
} GUID;
#else
typedef struct _GUID {
    unsigned long  Data1;
    unsigned short Data2;
```

```
    unsigned short Data3;
    unsigned char  Data4[ 8 ];
} GUID;
#endif
#endif
```

System operacyjny Windows, generując nowy identyfikator `GUID`, gwarantuje, że będzie on unikatowy w skali wszystkich identyfikatorów tego typu, wygenerowanych na całym świecie. C++ używa wartości `GUID` do identyfikowania interfejsów. Typową formą `GUID` jest `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`, gdzie każdy `x` reprezentuje cyfrę szesnastkową. Istnieje wiele narzędzi do generacji `GUID` w formie kanonicznej. Użytkownicy systemów operacyjnych Windows mogą użyć narzędzia firmy Microsoft o nazwie *GuidGen* (jest ono częścią MS Visual C++). Użytkownicy Linuxa mogą użyć polecenia `/usr/bin/uuidgen`. Można go znaleźć w pakiecie *libuuid1* (Debian).

Poszczególne elementy wygenerowanego identyfikatora należy wpisać w postaci szesnastkowej do poszczególnych pól struktury `GIUD`, tak jak pokazano to na listingu 4.2 obrazującego przykładową konstrukcję prostego kalkulatora służącego do obliczania pól powierzchni podstawowych figur geometrycznych (prostokąta oraz trójkąta).

Listing 4.2. Kod modułu definiującego interfejsy z implementacją zarządzania czasem ich życia

```
#ifndef __unit1_H__
#define __unit1_H__
#include <Windows.h>
using namespace std;
// interfejs IA
class IA : public IUnknown
{
  public:
    virtual void __stdcall dane(double x, double y) = 0;
    virtual void __stdcall funIA() = 0;
};
//------------------------------------------------------------
// interfejs IB
class IB : public IUnknown
{
  public:
    virtual void __stdcall dane(double x, double y) = 0;
    virtual void __stdcall funIB() = 0;
};
//------------------------------------------------------------
```

```
class IB2 : public IUnknown
{
  public:
    virtual void __stdcall dane(double x, double y) = 0;
    virtual void __stdcall funIB2() = 0;
};
//-----------------------------------------------------------
// deklaracje identyfikatorów GUID interfejsów
//['{E7D1FDE1-8A61-11D9-8C68-00E07D843852}']
static const GUID iidIA =
       { 0xE7D1FDE1, 0x8A61, 0x11D9, { 0x8C, 0x68, 0x00,
          0xE0, 0x7D, 0x84, 0x38, 0x52 } };
//['{E7D1FDE2-8A61-11D9-8C68-00E07D843852}']
static const GUID iidIB =
       { 0xE7D1FDE2, 0x8A61, 0x11D9, { 0x8C, 0x68, 0x00,
          0xE0, 0x7D, 0x84, 0x38, 0x52 } };
//-----------------------------------------------------------
// Klasa komponentu
class TMyComp : public IA, public IB
{
  private:
    double a, b;
    LONG Addend; // licznik odwołań do interfejsu

  public:
    // Implementacja interfejsu IUnknown
    virtual HRESULT __stdcall
            QueryInterface(const IID& iid, void **Obj);
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();
    void __stdcall dane(double x, double y)
                       {a = x; b = y;}

    // Implementacja interfejsu IA
    void __stdcall funIA()
      {cout << "Funkcja interfejsu IA oblicza pole
             prostokąta = " << a*b << endl; }

    // Implementacja interfejsu IB
    void __stdcall funIB()
      {cout << "Funkcja interfejsu IB oblicza pole
               trójkąta = " << 0.5*a*b << endl;}
};
#endif
```

Listing 4.3. Program kliencki

```
#include <iostream>
#include "unit1.h"
//Metoda CreateInstance() służy do tworzenia komponentu
//MyComp i pobierania od niego wskaźnika do interfejsu
//Iunknown schodząc po lewej stronie drzewa realizacji
IUnknown *CreateInstance()
{
  IUnknown *ptr =
  static_cast<IUnknown*>(static_cast<IA*>(new TMyComp));
  ptr->AddRef();
  return ptr;
}
//-----------------------------------------------------------
//Prosimy interfejs IUnknown o zwrócenie wskaźników
//do interfejsów IA oraz IB
HRESULT __stdcall TMyComp::QueryInterface(const IID& iid,
                                          void **ppvObject)
{
  if (iid == iidIA)
    //Klient wywołuje elementy interfejsu IA
    *ppvObject = static_cast<IA*>(this);
    else
      if (iid == iidIB)
      //Klient wywołuje elementy interfejsu IB
       *ppvObject = static_cast<IB*>(this);
    else
      if (iid == IID_IUnknown)
         //Klient pyta o interfejs IUnknown
       *ppvObject =
          static_cast<IUnknown*>(static_cast<IA*>(this));
    else {
      //Klient pyta o interfejs, którego nie znamy
      ppvObject = NULL;
      return E_NOINTERFACE;
    }
  static_cast<IUnknown*>(*ppvObject)->AddRef();
  return S_OK;
}
//-----------------------------------------------------------
ULONG __stdcall TMyComp::AddRef()
{
  return InterlockedIncrement(&Addend);
}
```

```
//--------------------------------------------------------
ULONG __stdcall TMyComp::Release()
{
  if (InterlockedDecrement(&Addend) == 0) {
    delete this;
    return 0;
  }
  return Addend;
}
//--------------------------------------------------------
int main()
{
  HRESULT hr;
  // Tworzenie komponentu
  IUnknown *pUnknown = CreateInstance();
  // Korzystamy z interfejsu IA
  IA *pIA = NULL;
  hr = pUnknown->QueryInterface(iidIA, (void**)&pIA);
  if (SUCCEEDED(hr)) {
    pIA->dane(2,2);
    pIA->funIA();
    pIA->Release();
  }
  // Korzystamy z interfejsu IB
  IB *pIB = NULL;
  hr = pUnknown->QueryInterface(iidIB, (void**)&pIB);
  if (SUCCEEDED(hr)) {
    pIB->dane(2,2);
    pIB->funIB();
    pIB->Release();
  }
  pUnknown->Release(); // delete pUnknown;
  system("PAUSE");
  return 0;
}
```

Wynik działania programu:

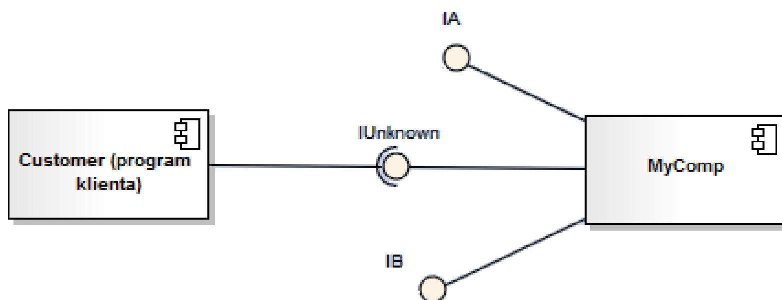
```
Jestem fa() i obliczam pole prostokata =4
Jestem fb() i obliczam pole trojkata =2
Aby kontynuować, naciśnij dowolny klawisz . . .
```

Wywołania funkcji `AddRef()` i `Release()` dokonywane są w celu umożliwienia zarządzania czasem życia interfejsów. W celu użycia mechanizmu automatycznego zliczania odwołań należy zadeklarować wskaźnik do typu bazowego interfejsu. Po jego zainicjowaniu wywołujemy metodę `AddRef()` z

funkcją `InterlockedIncrement()` zwiększającą licznik odwołań `Addend` do interfejsu. Kiedy obiekt kończy swoje operacje, wywoływana jest funkcja `Release()` z funkcją `InterlockedDecrement()` zmniejszającą licznik odwołań do interfejsu. Wymienione funkcje można zaimplementować w sposób dowolny. W powyższym przykładzie zastosowano rozwiązanie preferowane przez klasę `TInterfacedObject` z modułu *system.hpp*.

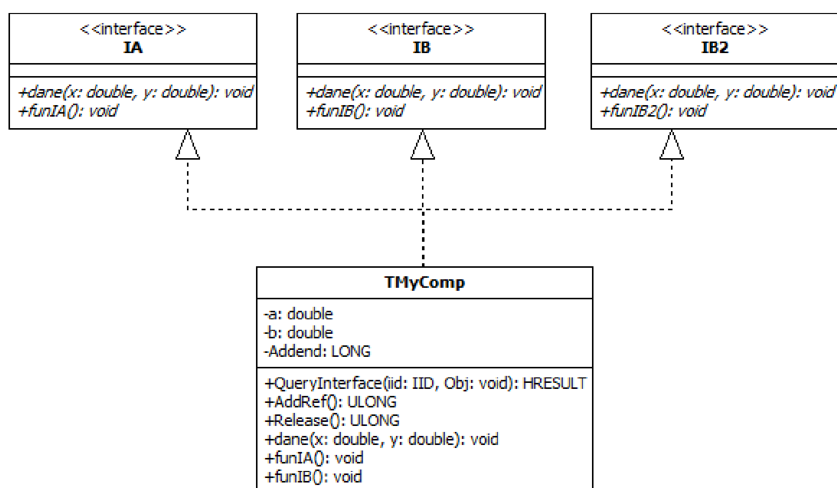
Funkcji `InterlockedIncrement()` oraz `InterlockedDecrement()`, zdefiniowanych w module *sysutils.hpp*, należy używać tylko na platformie Linuksowej. Aplikacje Windows używają analogicznych funkcji API zdefiniowanych w module *windows.hpp* lub *winbase.h*. Należy zwrócić uwagę, iż w przykładzie z listingu 4.3 zamiast standardowej metody rzutowania typów wykorzystano operator `static_cast` pozwalający wykonać rzutowanie niepolimorficzne (o czym wspomniano już w poprzednim rozdziale).

Praktyczne wykorzystywanie tak skonstruowanych interfejsów jest czynnością niezbyt skomplikowaną. Program klienta posiada funkcję `main()`, w której deklarowane są wskaźniki do istniejących interfejsów. Metoda `CreateInstance()` służy do tworzenia nowego komponentu `MyComp` i pobierania od niego wskaźnika do interfejsu bazowego `IUnknown`, tak jak pokazano to na diagramie komponentów z rysunku 4.2.

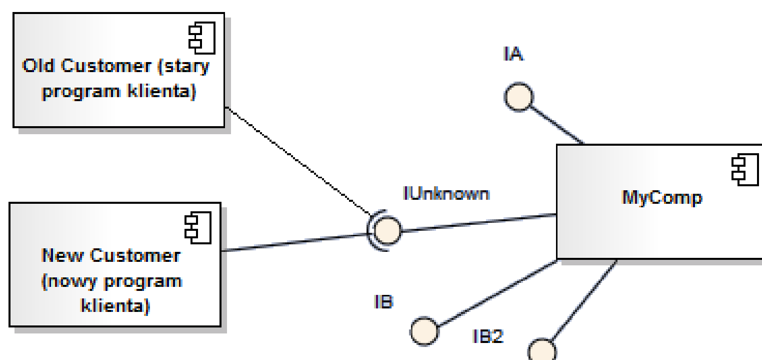


Rysunek 4.2. Diagram komponentów odpowiadający przykładowi 4.3

Kiedy programista potrzebuje dodać nową metodę do istniejącego interfejsu, np. `IB`, tworzy nową wersję interfejsu `IB2`, tak jak pokazano to na rysunku 4.3. Nowo utworzony interfejs powinien być zaimplementowany z nowym identyfikatorem `iidIB2`. Po tej modyfikacji programista ma do dyspozycji nową wersję komponentu (kalkulatora) `MyComp`. Uprzednio skonstruowany program kliencki który nie ma wiedzy o nowej metodzie `funIB2()` interfejsu może dalej używać starszej wersji komponentu. Nie ma potrzeby rekompilacji starego klienta. Nowy program kliencki powinien posiadać wiedzę o nowej metodzie `funIB2()` – w konsekwencji może używać nowej wersji komponentu, tak jak pokazano to na rysunku 4.4.



Rysunek 4.3. Dodawanie nowej metody do interfejsu



Rysunek 4.4. Dodawanie nowych funkcji do istniejących interfejsów

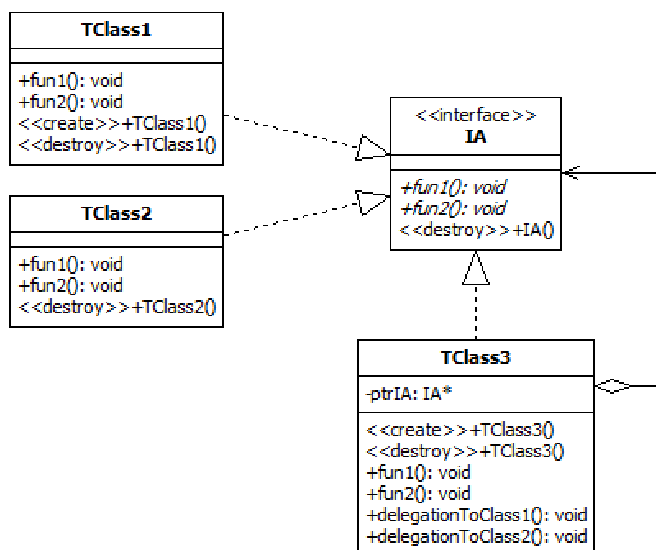
Podczas pracy z identyfikowanymi interfejsami programista powinien pamiętać, iż zawsze należy stworzyć nową wersję interfejsu, gdy modyfikacji podlega:

- liczba funkcji w interfejsie,
- kolejność funkcji w interfejsie,
- liczba parametrów w funkcji interfejsu,
- kolejność parametrów w funkcji interfejsu,
- typ parametrów w funkcji interfejsu,
- typ wartości zwracanej przez funkcję interfejsu.

4.3. Delegowanie obiektów poprzez wspólny interfejs

Opisany w poprzednim rozdziale mechanizm delegowania operacji można z powodzeniem zastosować również w przypadku bardziej złożonej struktury systemu zawierającej interfejsy. W takiej sytuacji mało opłacalnym podejściem byłoby delegowanie pojedynczych funkcji. Funkcje systemu, które powinny być delegowane zapisywane są w interfejsie, który to interfejs jest z kolei realizowany przez wszystkie elementy systemu.

Załóżmy, że w systemie występują klasy: `TClass1` i `TClass2` realizujące wspólny interfejs oraz klasa `TClass3`, tak jak pokazano to na rysunku 4.5.



Rysunek 4.5. Delegowanie interfejsu

W klasie `TClass3` zdefiniowano operacje `delegationToClass1()` oraz `delegationToClass2()`, takie że wywołanie ich określa odpowiednio obiekty klas do których obiekt klasy `TClass3` będzie delegował dalsze wywołania. W efekcie, `TClass1` oraz `TClass2` będą spełniać rolę „superklas" klasy `TClass3`. Zrealizowanie tego mechanizmu poprzez wspólny interfejs daje pewność, iż wywoływane metody zostały prawidłowo zaimplementowane w każdej z klas. Łańcuch kolejnych delegacji poprzez wspólny interfejs może mieć dowolną długość. Wystarczy aby obiekt klasy, który ma być delegowany będzie miał interfejs w swojej ścieżce realizacji, tak jak pokazano to na listingu 4.4. Delegowanie operacji interfejsu możliwe jest dzięki zagregowaniu interfejsu z klasą, w której zaimplementowane są operacje delegujące. Dzięki temu operacje delegujące z klasy `TClass3` mają dostęp do wskaźników wskazujących na inne obiekty w systemie realizujące wspólny interfejs.

Listing. 4.4. Implementacja diagramu z rysunku 4.5

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

class IA {
   public:
     virtual void fun1() = 0;
     virtual void fun2() = 0;
     virtual ~IA() {}
 };
//-----------------------------------------------------------
class TClass1 : public IA {
   public:
     void fun1() { cout << "TClass1::fun1()\n"; }
     void fun2() { cout << "TClass1::fun2()\n"; }
     TClass1() { cout << "Konstruktor TClass1\n"; }
     ~TClass1() { cout << "Destruktor TClass1\n"; }
 };
//-----------------------------------------------------------
class TClass2 : public IA {
   public:
     void fun1() { cout << "TClass2::fun1()\n"; }
     void fun2() { cout << "TClass2::fun2()\n"; }
     ~TClass2() { cout << "Destruktor TClass2\n"; }
 };
//-----------------------------------------------------------
class TClass3 : public IA {
   public:
     // Konstruktor TClass3 wywołuje konstruktor TClass1
     TClass3() : ptrIA( new TClass1() ) {
       cout << "Konstruktor TClass3\n";
     }
     // Wirtualny destruktor
     virtual ~TClass3() { delete ptrIA; }
     void fun1() { ptrIA->fun1(); }
     void fun2() { ptrIA->fun2(); }
     void delegationToClass1() {
       delete ptrIA; ptrIA = new TClass1();
     }
     void delegationToClass2() {
       delete ptrIA; ptrIA = new TClass2();
     }
   private:
     // Agregacja interfejsu IA
```

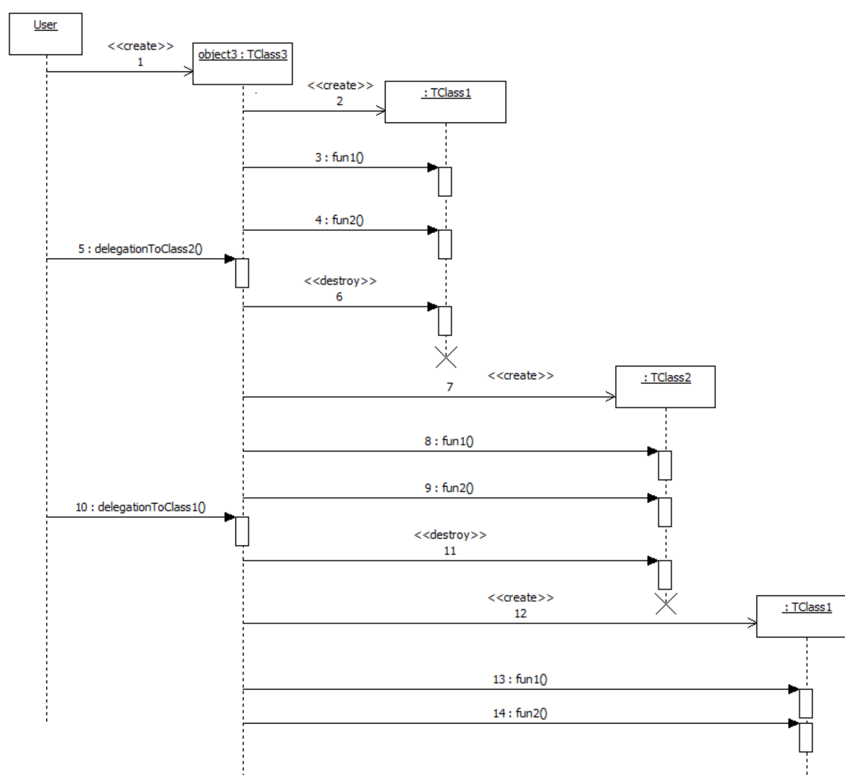
```
      IA* ptrIA;
  };
//------------------------------------------------------------
int main() {
   // Tworzony jest obiekt anonimowy klasy TClass1,
   // a następnie obiekt object3 klasy TClass3
   TClass3 object3;
   // Wywoływane są metody klasy TClass1
   object3.fun1();
   object3.fun2();
   // Wywoływany jest destruktor klasy TClass1
   // i tworzony obiekt anonimowy klasy TClass2
   object3.delegationToClass2();
   // Wywoływane są metody klasy TClass2
   object3.fun1();
   object3.fun2();
   //Łańcuch delegacji może być powtarzany wielokrotnie
   object3.delegationToClass1();
   object3.fun1();
   object3.fun2();
   cin.get();
   return 0;
}
```

Wynik działania programu:

```
Konstruktor TClass1
Konstruktor TClass3
TClass1::fun1()
TClass1::fun2()
Destruktor TClass1
TClass2::fun1()
TClass2::fun2()
Destruktor TClass2
Konstruktor TClass1
TClass1::fun1()
TClass1::fun2()
```

Na rysunku 4.6 pokazano diagram sekwencji odpowiadający implementacji kodu z listingu 4.4 i odzwierciedlający działanie programu po uruchomieniu. Diagram ten w sposób jednoznaczny wyjaśnia ideę korzystania z mechanizmów wielokrotnych delegacji obiektów realizujących wspólny interfejs.

Z przebiegu diagramu sekwencji pokazanego na rysunku 4.6 z łatwością odczytamy, iż w trakcie delegowania tworzona jest hierarchia obiektów, a nie klas, i hierarchia ta może zostać zmodyfikowana w dowolnym momencie działania programu.



Rysunek 4.6. Diagram sekwencji odpowiadający implementacji kodu z listingu 4.4

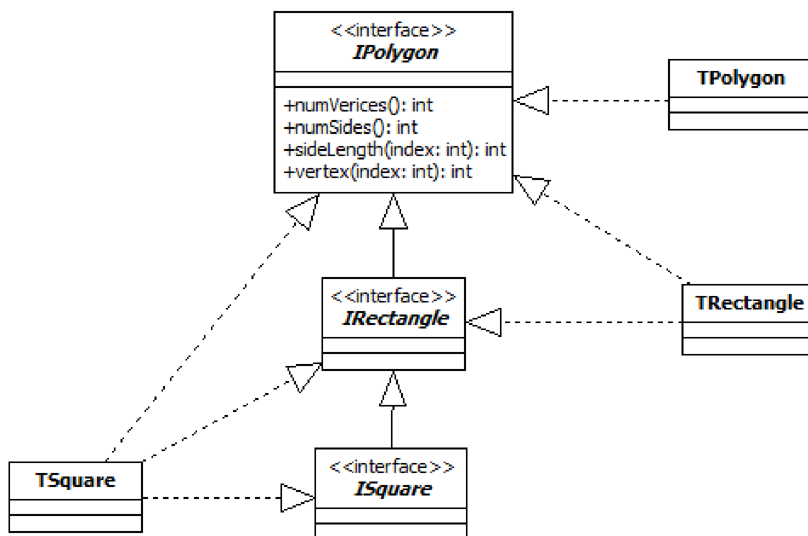
4.4. Programowanie obiektowe z użyciem interfejsów

Najważniejszą funkcją interfejsów jest odseparowanie dziedziczenia typów od dziedziczenia klas. Dziedziczenie klas jest wydajnym narzędziem, pozwalającym na wielokrotne wykorzystywanie kodu. Klasa potomna w prosty sposób dziedziczy atrybuty, metody i właściwości klasy bazowej unikając konieczności ponownego implementowania wcześniej zdefiniowanych elementów składowych. W językach opartych na silnym modelu typów danych, kompilator traktuje klasę jako typ danych, zatem dziedziczenie klas jest równoważne z dziedziczeniem typów. Jednak w świecie rzeczywistym klasy i typy są bytami zupełnie odrębnymi.

Wiele książek w celach poglądowych opisuje związek dziedziczenia jako związek bytności, np. klasa prostokąt (ang. *rectangle*) jest klasą kwadrat (ang. *square*). Jednak w świecie realnym tego rodzaju związek bytności nie występuje. Kwadrat oczywiście jest prostokątem, ale nie oznacza to, iż można odziedziczyć klasę `TSquare` po klasie `TRectangle`. Podobnie, prostokąt jest wielokątem (ang. *polygon*), co jednak nie oznacza, że klasa `TSquare` dziedziczy po klasie `TPolygon`. Dziedziczenie klas skutkuje tym, że klasa potomna

przechowuje wszystkie atrybuty zadeklarowane w klasie bazowej, chociaż w większości zastosowań informacja taka jest całkowicie zbyteczna. Przykładowo, obiektowi klasy kwadrat wystarczy jeden atrybut do zapisania długości wszystkich swoich boków. Obiekt klasy prostokąt powinien przechowywać informacje na temat długości dwóch boków, zaś wielokąt długości wszystkich boków oraz dodatkowo kąty między nimi zawarte.

Rozwiązaniem zaistniałej sytuacji jest odseparowanie dziedziczenia typów od dziedziczenia klas. Dziedziczenie typów realizowane jest poprzez interfejsy, natomiast dziedziczenie klas ogranicza się do dziedziczenia atrybutów i operacji. Oznacza to, że interfejs `ISquare` dziedziczy po `IRectangle`, który z kolei dziedziczy po `IPolygon`. W zupełnym oderwaniu od interfejsów, klasa `TSquare` realizuje `ISquare`, zaś `TRectangle` oraz `TPolygon` realizują odpowiednio `IRectangle` oraz `IPolygon`, tak jak zaprezentowano to na rysunku 4.7.

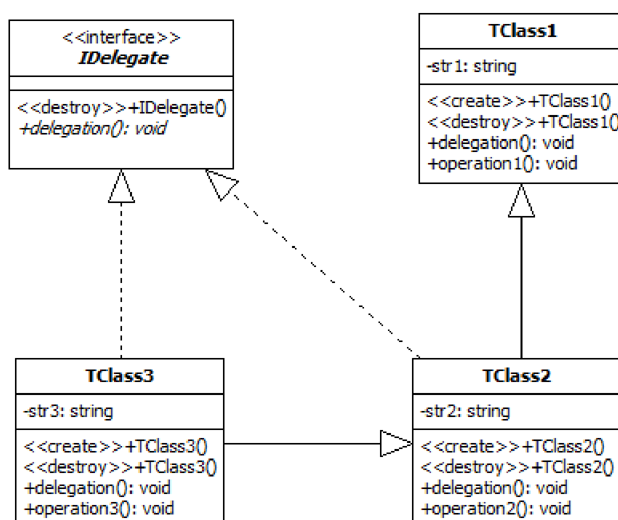


Rysunek 4.7. Realizacja interfejsów

Z rysunku 4.7 bez trudu odczytamy, iż klasa realizuje interfejsy, które mogą dziedziczyć po sobie. Dziedziczenie interfejsów ma jedynie na celu zaoszczędzenie ponownego wypisywania deklaracji operacji. W przypadku, gdy klasa realizuje dany interfejs, nie oznacza to, że może realizować jego interfejsy nadrzędne. Klasa realizuje jedynie te interfejsy, które zostały wymienione w jej deklaracji (i ew. deklaracji klas nadrzędnych). Zatem, mimo że interfejs prostokąta dziedziczy po interfejsie wielokąta, to klasa prostokąt musi jawnie zawierać deklaracje interfejsów prostokąta i wielokąta.

4.4.1. Wirtualna realizacja interfejsu C++

Jak zaznaczono we wstępie obecnego rozdziału, C++ jest jednym z niewielu obiektowych języków programowania, w którym interfejsy są emulowane poprzez szczególny rodzaj klas abstrakcyjnych. Ta właściwość języka implikuje możliwość wirtualnego realizowania interfejsów. Na rysunku 4.8 przedstawiono sytuację, w której klasa `TClass3` realizuje interfejs `IDelegate` i jednocześnie dziedziczy po klasie `TClass2`, która też realizuje ten sam interfejs. W praktyce oznacza to, iż `TClass3` realizuje dwie kopie tego samego interfejsu, co prowadzi do wystąpienia opisanego w Rozdziale 3 zjawiska dwuznaczności. Aby wyeliminować ten efekt klasy `TClass2` oraz `TClass3` powinny realizować `IDelegate` jako interfejs wirtualny, tak jak zaprezentowano to na listingu 4.5.



Rysunek 4.8. Wirtualna realizacja interfejsu

Listing 4.5. Implementacja delegowania poprzez wirtualną realizację interfejsu

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

class IDelegate {
  public:
    virtual ~IDelegate() {};
    virtual void delegation() = 0;
};
//-----------------------------------------------------------
class TClass1 {
  private:
    string str1;
```

```
  public:
    TClass1() : str1("TClass1") {cout << str1 << endl;};
    virtual ~TClass1(){};
};
//-----------------------------------------------------------
class TClass2 : virtual public IDelegate, public TClass1 {
  private:
    string str2;
  public:
    TClass2() : str2("TClass2") { };
    virtual ~TClass2() {};
    void delegation() {
        this->operation2();
    };
    void operation2() {
      cout << this->str2 << endl;
    };
};
//-----------------------------------------------------------
class TClass3 : virtual public IDelegate,  public TClass2 {
  private:
    string str3;
  public:
    TClass3() : str3("TClass3"){};
    ~TClass3(){};
    void delegation() {
       this->operation2();
     this->operation3();
    };
    void operation3() {
       cout << this->str3 << endl;
    };
};
//-----------------------------------------------------------
void fun(IDelegate &obj)
{   obj.delegation();
};
//-----------------------------------------------------------
int main() {
   TClass3 object3;
   fun(object3);
   cin.get();
   return 0;
}
```

Wynik działania programu:

```
TClass1
TClass2
TClass3
```

4.5. Składniki

Coraz powszechniejsze stosowanie technik komponentowych zmienia oblicze programowania. W literaturze przedmiotu można dostrzec dwa podejścia opisujące to zagadnienie [26].

Pierwszym z nich jest umożliwienie programowania aplikacji nie tylko zawodowym programistom, ale też przeciętnym użytkownikom. Użytkownik ma do dyspozycji zbiór komponentów charakteryzujących się wysokim poziomem abstrakcji oraz intuicyjnym, graficznym interfejsem, który umożliwia samodzielne wykonywanie wielu czynności programistycznych.

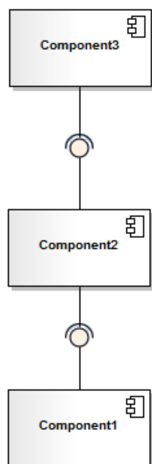
Drugim podejściem jest zmiana w pracy programisty. Im więcej komponentów ma do dyspozycji programista, tym mniejsza jest ziarnistość systemu. Podstawowymi elementami, z którymi pracuje programista nie są już drobnoziarniste operacje właściwe językowi programowania, ale rozbudowane elementy architektury składnikowej, które można niemal dowolnie konfigurować. Jest to wyraźnie dostrzegalne w nowoczesnych zintegrowanych środowiskach łączących język programowania, kompilator, narzędzia wizualnego modelowania oraz wizualnego tworzenia interfejsu użytkownika. Należy jednak zauważyć, iż technologia ta nie upowszechnia się w sposób wystarczająco szybki. Głównym powodem jest kwestia specyfikacji zachowań komponentów. Obecnie dostępne są bardzo złożone i charakteryzują się obszerną dokumentację, co w dużym stopniu ogranicza ich pole zastosować, zwłaszcza jeżeli weźmie się pod uwagę naturalną niechęć człowieka do częstej zmiany narzędzi, którymi się posługuje.

W ujęciu tradycyjnym, kompozycja hierarchiczna wydaje się naturalną zasadą strukturalizowania systemu. Jednak w rzeczywistości nie jest to do końca prawdą. Okazuje się, iż podejście natury jest zupełnie inne – jest niekompozycyjne. Porównajmy dwa typy diagramów komponentów, do których prowadzi każdy z wymienionych kierunków. Na tego typu diagramach komponenty reprezentują pewien zasób wiedzy o dziedzinie problemu, zaś interfejsy reprezentują wiedzę o istnieniu innych komponentów. W systemie kompozycyjnym diagram taki przedstawiany jest w postaci hierarchicznej, tak jak pokazano to na rysunku 4.9.

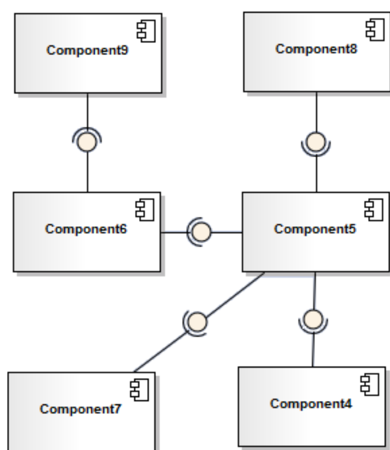
Każdy komponent jest połączony wyłącznie z najbliższymi sąsiadami. Dzięki temu system może być z łatwością dekomponowany na niezależne części, połączone z innymi poprzez dobrze określone interfejsy.

W systemie niekompozycyjnym, diagram komponentów ma zupełnie odmienną strukturę. Tak, jak pokazano na rysunku 4.10 jest mocno rozgałęziony i nielokalny, co oznacza, że każdy komponent może być połączony z wieloma

innymi, oraz że wzajemne połączenia komponentów prowadzą do wielu różnych części diagramu.



Rysunek 4.9. Hierarchiczna struktura komponentów



Rysunek 4.10. Niekompozycyjna struktura komponentów

Dekomponowanie systemu pokazanego na rysunku 4.10 przebiega w sposób o wiele bardziej arbitralny. Interfejsy pomiędzy częściami systemu są rozbudowane, a co za tym idzie rozważanie pracy komponentów niezależnie od ich relacji z innymi częściami systemu jest o wiele trudniejsze.

Nie ulega wątpliwości, iż kompozycja hierarchiczna jest bardziej atrakcyjna dla człowieka. Wynika to z podstawowej własności pamięci ludzkiej, jaką jest rozmiar pamięci krótkotrwałej [27]. Człowiek może sprawnie operować jednocześnie stosunkowo niewielkim zasobem pojęć. Podział dużego projektu jest więc niezbędny, aby zrozumieć ideę funkcjonowania części, które obejmie umysłem jedna osoba. Konstrukcje spotykane w przyrodzie nie napotykają na

takie ograniczenia. Obowiązującą zasadą jest selekcja naturalna – nowe systemy powstają poprzez podział, łączenie lub modyfikacje starszych wersji. Jedynym kryterium oceny jest to, jak system funkcjonuje w swoim naturalnym środowisku. Sukces osiągają te systemy, które charakteryzują się dużą zdolnością do ponownego użycia, tzn. takie, które posiadają zdolność do występowania w jak największej liczbie różnych implementacji. Prowadzi to do struktury charakteryzującej się niewielką kompozycyjnością.

W czystej postaci każde z omawianych pojęć prowadzi do zachowań ekstremalnych. Działalność ludzka jest ukierunkowana przede wszystkim na szybkie efekty, a to oznacza iż zdominowana jest przez redukcjonizm w przeciwieństwie do bytów naturalnych, które mają charakter eksploracyjny i holistyczny. Nowe narzędzia programistyczne oraz nowe platformy systemowe pozwalają stosować bardziej naturalne podejście do projektowania systemów.

Podsumowanie

Programowanie oparte na obiektach (ang. *object-base programming*) to programowanie obiektowe OOP bez dziedziczenia. Często postrzega się je jako programowanie komponentowe uzupełnione o składnię i związki klas. Dzięki temu uzyskuje się wygodną notację, która pomaga w hermetyzowaniu stanu obiektu i sprawnym definiowaniu operacji właściwych dla danego stanu. Bez dziedziczenia, abstrakcja danych staje się dużo prostsza – nie występują problemy z przedefiniowywaniem operacji, operatorów i wielodziedziczeniem. Osobom zainteresowanym szerszym omówieniem paradygmatu programowania komponentowego należy polecić książkę Clemensa Szyperskiego zawierającą przegląd najnowszych technik komponentowych [28]. W książce omówiono dokładnie pojęcie komponentu i interfejsu oraz trzy platformy komercyjne: platformę organizacji OMG ze swoim standardem CORBA, platformę Microsoft ze standardami COM, DCOM, OLE i Active X oraz platformę firmy Sun Microsystems oferującą język Java i JavaBeans.

ROZDZIAŁ 5

WZORCE PROJEKTOWE PROGRAMOWANIA OBIEKTOWEGO

5.1. Podział wzorców 136
5.2. Singleton 137
5.3. Fabryka abstrakcyjna 140
5.4. Prototyp 143
5.5. Metoda wytwórcza 146
5.6. Budowniczy 149
5.7. Adapter 151
5.8. Dekorator 155
5.9. Fasada 158
5.10. Pełnomocnik 160
5.11. Kompozyt 165
5.12. Most 166
5.13. Pyłek 167
5.14. Metoda szablonu 168
5.15. Strategia 171
5.16. Obserwator 174
5.17. Stan 179
5.18. Wizytator 182
5.19. Polecenie 186
5.20. Łańcuch odpowiedzialności 188
5.21. Interpreter 189
5.22. Iterator 193
5.23. Memento 195
5.24. Mediator 196
Podsumowanie

5.1. Podział wzorców

W trakcie pracy nad różnego rodzaju projektami często napotykamy na powtarzające się problemy. Wzorce to próba systematyzacji takich doświadczeń i próba ułożenia ich w postaci poręcznych i sprawdzonych rozwiązań. Obowiązująca do dnia dzisiejszego ogólna definicja wzorca została podana przez Alexandera, Ishikawę, i Silversteina w 1977 roku [29]: *Każdy wzorzec opisuje problem, który ciągle pojawia się w naszej dziedzinie, a następnie określa zasadniczą część jego rozwiązania w taki sposób, by można było zastosować je nawet milion razy za każdym razem w nieco inny sposób*.

Wzorce opisują ogólne metody rozwiązywania typowych, często spotykanych problemów. Istnieją cztery główne kategorie wzorców związanych z procesem projektowania oprogramowania [30]:

- wzorce analityczne (ang. *analytical patterns*) – umiejscowione na poziomie opisu rzeczywistości,
- wzorce architektoniczne (ang. *architectural patterns*) – umiejscowione na poziomie integracji komponentów. Są to wzorce wysokiego poziomu określające strukturę i zachowanie systemu jako całości,
- wzorce projektowe programowania obiektowego (ang. *design patterns*) – umiejscowione na poziomie interakcji między klasami będącymi podstawowymi elementami programu zorientowanego obiektowo. Są to wzorce pośredniego poziomu, w ogólności określające strukturę i zachowanie komponentów oraz zestawów klas systemu,
- wzorce programowania lub idiomy (ang. *programming patterns*, *idioms*) – umiejscowione na poziomie języka programowania. Są to wzorce niskiego poziomu opisujące rozwiązania dla konkretnych, często spotykanych problemów implementacyjnych.

Wzorce projektowe programowania obiektowego identyfikują i opisują pewną abstrakcję, której poziom znajduje się powyżej poziomu abstrakcji pojedynczej klasy. Pierwszy katalog wzorców projektowych programowania obiektowego z objaśnieniami został opublikowany w 1994 roku [31]. Zgodnie z klasyfikacją opracowaną przez Ericha Gammę i współautorów dokonano podziału wzorców projektowych programowania obiektowego według następujących kategorii:

- wzorce konstrukcyjne (ang. *creational design patterns*) wykorzystywane do pozyskiwania obiektów zamiast bezpośredniego tworzenia egzemplarzy klas,
- wzorce strukturalne (ang. *structural design patterns*) pomagające łączyć obiekty w większe struktury,
- wzorce operacyjne (ang. *behavioral design patterns*) służące do definiowania komunikacji pomiędzy obiektami oraz kontrolowania przepływu danych w złożonym programie.

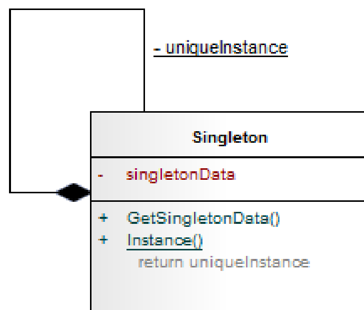
W podgrupie wzorców konstrukcyjnych, opisujących różne schematy tworzenia obiektów wyróżniono następujące wzorce: Budowniczy (ang. *builder*), Metoda fabrykująca (ang. *factory method*), Prototyp (ang. *prototype*), Singleton (ang. *singleton*) oraz Fabryka abstrakcyjna (ang. *abstract factory*). W podgrupie wzorców strukturalnych występują: Adapter (ang. *adapter*), Dekorator (ang. *decorator*), Fasada (ang. *facade*), Kompozyt (ang. *composite*), Most (ang. *bridge*), Pełnomocnik (ang. *proxy*) i Pyłek (ang. *flyweight*). W skład wzorców operacyjnych wchodzą: Łańcuch odpowiedzialności (ang. *chain of responsibility*), Polecenie (ang. *command*), Interpreter (ang. *interpreter*), Iterator (ang. *iterator*), Mediator (ang. *mediator*), Memento (ang. *memento*), Obserwator (ang. *observer*), Stan (ang. *state*), Strategia (ang. *strategy*), Metoda szablonu (ang. *template method*) oraz Wizytator (ang. *visitor*).

Wraz z upowszechnianiem się systemów oferujących możliwości programowania wielowątkowego i współbieżnego pojawiły wzorce współbieżności opisujące często spotykane problemy związane z praktycznymi metodami współdziałania i współdzielenia wielu obiektów, wątków lub procesów. W tej grupie znajdują się takie wzorce jak: Aktywny obiekt, Asynchroniczne sterowanie przez zdarzenia, Udaremnianie, Blokada z podwójnym zatwierdzaniem, Ochraniane wstrzymywanie, Obiekt monitorujący, Blokada zapisu i odczytu, Zarządca procesów, Pula wątków, Pamięć dla wątków oraz Reaktor.

Wzorce projektowe programowania obiektowego są bardzo często wykorzystywane jako pewnego rodzaju podstawa myślowa, która jednak nie zawsze w sposób niemodyfikowalny jest implementowana w konkretnym języku programowania. Często wzorzec pozostaje jedynie w umyśle programisty stanowiąc wygodną podstawę dla jego własnej inwencji twórczej. W dalszej części rozdziału opisano 23 podstawowe, najczęściej w praktyce wykorzystywane wzorce projektowe programowania obiektowego. Opis wzorców współbieżności wykracza poza ramy tematyczne niniejszego podręcznika i powinien być nieodłącznym elementem współczesnych wykładów z zakresu programowania współbieżnego i wielowątkowego.

5.2. Singleton

Singleton jest najprostszym wzorcem projektowym umożliwiającym stworzenie obiektowej alternatywy dla zmiennych globalnych, nieobecnych w wielu językach obiektowych: zapewnienie istnienia w aplikacji tylko jednego obiektu danej klasy. Obiekt ten udostępniany jest poprzez odpowiednią metodę statyczną. Na rysunku 5.1 pokazano diagram omawianego wzorca składającego się z jednej klasy zarządzającej swoim własnym obiektem.



Rysunek 5.1. Singleton

Implementację singletonu przeprowadza się w sposób pokazany na listingu 5.1. Prywatny atrybut statyczny `uniqueInstane` wskazuje na klasę singletonu. Zarządzaniem anonimowym obiektem klasy zajmuje się publiczna metoda statyczna o nazwie `Instance()`. Cechą charakterystyczną klasy `Singleton` jest posiadanie prywatnego konstruktora i publicznego destruktora. Umieszczenie konstruktora w sekcji prywatnej definicji klasy ma za zadanie bezpośrednie uniemożliwienie wywoływania go poziomu programu głównego.

Listing 5.1. Przykładowa implementacja singletonu w C++

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

class Singleton {
  private:
     static Singleton* uniqueInstance;
     Singleton();
  public:
     static Singleton* Instance();
     string getSingletonData();
     string singletonData;
     ~Singleton();
};
//-----------------------------------------------------------
Singleton::Singleton() {
//  this->name = "Singleton test 1";
}
//-----------------------------------------------------------
Singleton::~Singleton(){
  delete Singleton::uniqueInstance;
  Singleton::uniqueInstance = NULL;
}
//-----------------------------------------------------------
string Singleton::getSingletonData() {
```

```
  return singletonData;
}
//-----------------------------------------------------------
Singleton* Singleton::Instance(){
  if (Singleton::uniqueInstance == NULL) {
    uniqueInstance = new Singleton;
  }
  return uniqueInstance;
}
//-----------------------------------------------------------
Singleton* Singleton::uniqueInstance = NULL;
//-----------------------------------------------------------
int main()
{
  //Singleton* singleton = Singleton::Instance();
  //cout << singleton->getSingletonData() << endl;
  Singleton::Instance()->singletonData = "Singleton test 2";
  cout << Singleton::Instance()->singletonData << endl;
  cin.get();
  return 0;
}
```

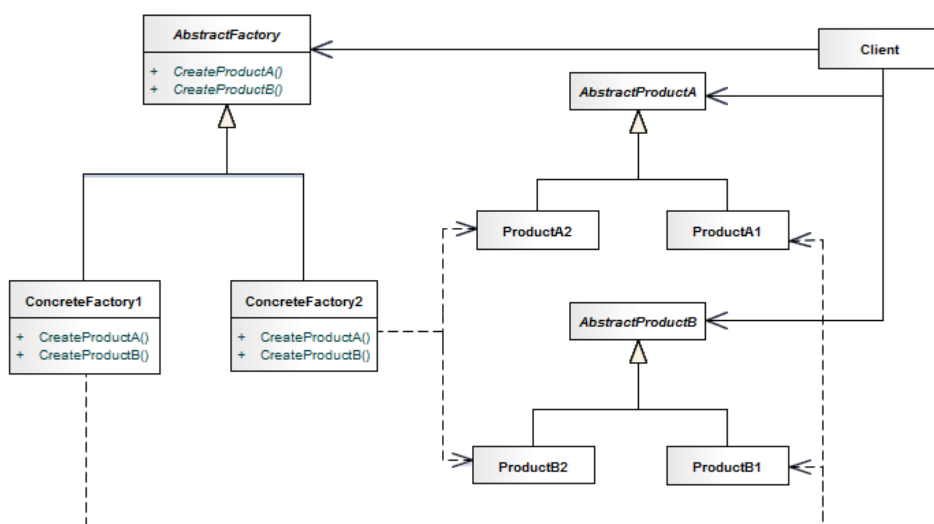
Wynik działania programu:

```
Singleton test 2
```

Jeżeli statyczny atrybut przechowujący egzemplarz klasy wskazuje na wartość pustą `NULL` (co w praktyce oznacza, że anonimowy obiekt klasy dotąd nie został utworzony), wówczas anonimowy obiekt jest tworzony i zapamiętywany pod postacią wskaźnika `uniqueInstane`. Dzięki temu, niezależnie od tego, który raz wywoływana jest metoda `Instance()`, zawsze zwraca utworzony i jedyny anonimowy obiekt klasy `Singleton`. Aby uniemożliwić użytkownikowi samodzielne tworzenie obiektów singletonu z pominięciem metody statycznej, klasa `Singleton` uniemożliwia dostęp do konstruktora z zewnątrz, zwykle czyniąc go prywatnym. Jak łatwo można zauważyć, omawiany wzorzec jest prostym sposobem na zapewnienie, że zostanie utworzony dokładnie jeden obiekt żądanej klasy, który będzie dostępny dla wszystkich pozostałych obiektów programu. Singleton jest obiektem bezstanowym, tzn. sposób działania metody statycznej `Instance()` nie zależy od stanu, w jakim znajduje się program: użytkownik otrzymuje anonimowy obiekt klasy na żądanie, niezależnie od tego, czy został on utworzony wcześniej, czy nie. Singleton pozwala także stosować mechanizmy polimorfizmu. Uzupełnienie diagramu 5.1 o klasę nadrzędną nie wymaga modyfikacji wywołania po stronie programu użytkownika.

5.3. Fabryka abstrakcyjna

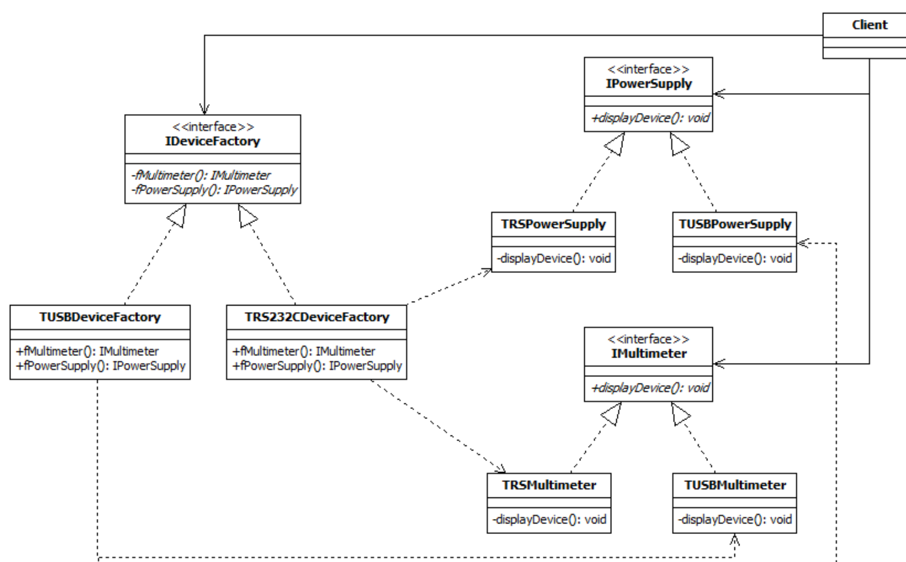
Wzorzec fabryki abstrakcyjnej umożliwia enkapsulację grupy operacji fabrykujących dotyczących tego samego zagadnienia. Tak jak pokazano to na rysunku 5.2 w podstawowej reprezentacji wzorzec składa się z trzech głównych ról: fabryki abstrakcyjnej (`AbstractFactory`) deklarującej abstrakcję dla operacji tworzących abstrakcyjne produkty i realizowanych w odpowiednich klasach (`ConcreteFactory1`, `ConcreteFactory2`), abstrakcyjnych produktów (`AbstractProductA`, `AbstractProductB`) deklarujących abstrakcje dla par konkretnych produktów (`ProductA1`, `ProductA2`, `ProductB1`, `ProductB2`), które będą tworzone przez konkretną fabrykę oraz z roli klienta (programu głównego) implementującego żądane abstrakcje.



Rysunek 5.2. Statyczny diagram klas wzorca fabryki abstrakcyjnej

Zazwyczaj fabryka abstrakcyjna jest budowana w postaci interfejsu. Następnie (w oprogramowaniu klienta) tworzone są konkretne implementacje fabryki. Konkretne obiekty tworzone są poprzez wywołanie metod interfejsu (a nie implementacji). W ten sposób od implementacji fabryki zależny jest tylko fragment kodu tworzący daną fabrykę. Fabryka pozwala na tworzenie zestawów obiektów dopasowanych do konkretnych zastosowań (np. różnych funkcjonalności, platform, itp.). Każda z konkretnych fabryk realizuje odmienny zestaw klas, ale zawsze posiadają one pewien zdefiniowany zespół interfejsów. Każda z konkretnych fabryk pozwala na tworzenie kolekcji obiektów zajmujących się realizacją pewnego zagadnienia ze względu na spójne kryterium wyboru. Przykładowo, można wyobrazić sobie interfejs `IDeviceFactory` deklarujący operacje fabrykujące dwie grupy urządzeń: mierniki uniwersalne oraz zasilacze. Interfejs ten realizowany jest przez dwie klasy odpowiedzialne za fabrykacje

odpowiednio mierników uniwersalnych zaopatrzonych w mechanizmy transmisji danych opartych na protokole RS 232C i USB, oraz zasilaczy zaopatrzonych w te same mechanizmy transmisji danych. Użytkownik chce skonfigurować urządzenia parami w ten sposób, aby mieć możliwość komunikowania się z nimi za pośrednictwem jednego protokołu. W tym celu deklaruje interfejs opisujący zasilacze oraz interfejs opisujący mierniki uniwersalne, a następnie tworzy ich implementacje odpowiednio dla urządzeń transmitujących dane w standardzie RS 232C i oddzielnie dla urządzeń obsługujących standard USB. Jeżeli urządzenia mają być np. skonfigurowane zgodnie z obsługiwanym protokołem RS 232C, wystarczy utworzyć fabrykę właściwą dla tych urządzeń `TRS232CDeviceFactory`, a następnie żądanej funkcji przekazać np. interfejs `IPowerSupply` opisujący zasilacze, tak jak pokazano to na rysunku 5.3 oraz listingu 5.2.



Rysunek 5.3. Abstrakcyjna fabryka laboratoryjnych przyrządów pomiarowych

Listing 5.2. Implementacja diagramu 5.3

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

class IMultimeter {
  public:
    virtual void displayDevice() = 0 ;
    virtual ~ IMultimeter(){};
};
//-----------------------------------------------------------
class IPowerSupply {
```

```
  public:
    virtual void displayDevice() = 0 ;
    virtual ~ IPowerSupply(){};
};
//-----------------------------------------------------------
class TRSMultimeter : public IMultimeter {
  private:
    void displayDevice() {
        cout << "RS 232C Multimeter" << endl ;
    }
};
//-----------------------------------------------------------
class TRSPowerSupply : public IPowerSupply {
  private:
    void displayDevice() {
        cout << "RS 232C Power Supply" << endl;
    }
};
//-----------------------------------------------------------
class TUSBPowerSupply : public IPowerSupply {
  private:
    void displayDevice() {
       cout << "USB Power Supply" << endl ;
    }
};
//-----------------------------------------------------------
class TUSBMultimeter : public IMultimeter {
  private:
    void displayDevice() {
        cout << "USB Multimeter" << endl ;
    }
};
//-----------------------------------------------------------
class IDeviceFactory {
  private:
    virtual IMultimeter* fMultimeter() = 0;
    virtual IPowerSupply* fPowerSupply() = 0;
  public:
    virtual ~ IDeviceFactory(){};
};
//-----------------------------------------------------------
class TRS232CDeviceFactory : public IDeviceFactory {
  public:
    IMultimeter* fMultimeter() {
        return new TRSMultimeter ;
```

```
    }
    IPowerSupply*  fPowerSupply() {
        return new TRSPowerSupply ;
    }
};
//-----------------------------------------------------------
class TUSBDeviceFactory : public IDeviceFactory {
  public:
    IMultimeter* fMultimeter() {
         return new TUSBMultimeter ;
    }
    IPowerSupply* fPowerSupply() {
         return new TUSBPowerSupply ;
    }
};
//-----------------------------------------------------------
int main()
{
//TUSBDeviceFactory *df;
  TRS232CDeviceFactory *df;
  IPowerSupply *id ;
  //df = new TUSBDeviceFactory;
  df = new TRS232CDeviceFactory;
  id = df->fPowerSupply() ;
  id->displayDevice() ;
  cin.get();
  return 0;
}
```

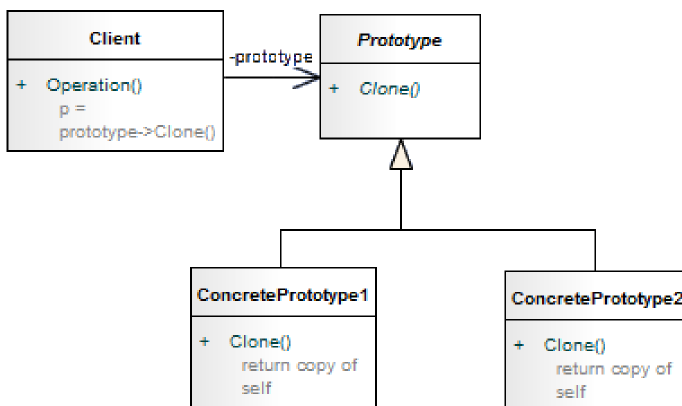
Wynik działania programu:

```
RS 232C Power Supply
```

5.4. Prototyp

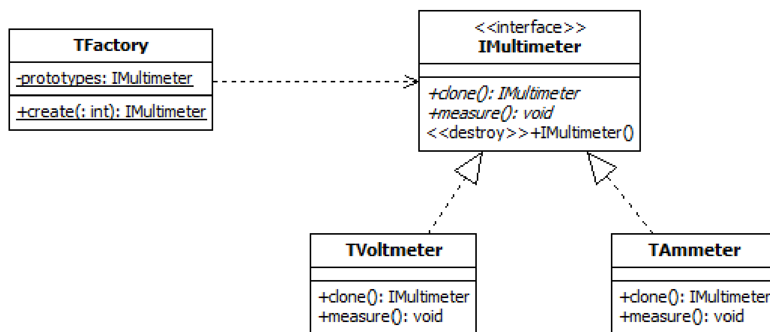
Prototyp jest wzorcem, opisującym mechanizm tworzenie nowych obiektów poprzez klonowanie jednego obiektu macierzystego. Zastosowanie wzorca ma miejsce w sytuacjach, w których należy utworzyć wiele podobnych obiektów tego samego typu znajdujących się w podobnym stanie. Prototyp tworzy klony wybranego obiektu zamiast tworzyć nowe egzemplarze tego samego obiektu. Mechanizm klonowania wykorzystywany jest wówczas, gdy należy wykreować dużą liczbę obiektów tego samego typu lub istnieje potrzeba tworzenia zbioru obiektów o bardzo podobnych właściwościach. Aby zaimplementować wzorzec deklaruje się klasę `Prototype` z abstrakcyjną operacją klonującą `Clone()`. Operacja ta jest implementowana w klasach realizujących abstrakcję `Prototyp`.

Klient, chcąc wykreować nowy obiekt wywołuje metodę `Clone()` pośrednio, za pomocą zdefiniowanej przez siebie operacji z parametrem określającym wymaganą docelową klasę realizującą abstrakcję `Prototype`. Na rysunku 5.4 pokazano diagram omawianego wzorca.



Rysunek 5.4. Podstawowa reprezentacja wzorca prototypu

Przykładem praktycznego wykorzystania wzorca prototypu jest sytuacja, w której w programie sterującym urządzeniami laboratoryjnymi należy uzyskać dostęp do wielu przyrządów tego samego typu. Nierzadko z bazy danych takich urządzeń pomiarowych pobieranych jest w jednym zapytaniu bardzo wiele konkretnych mierników. Standardowy diagram klas dla tego rodzaju przykładu oraz odpowiadający mu kod może wyglądać w sposób pokazany odpowiednio na rysunku 5.5 oraz listingu 5.3.



Rysunek 5.5. Klonowanie urządzeń laboratoryjnych

Listing 5.3. Implementacja diagramu z rysunku 5.5

```
#include <iostream>
#define number 5
using namespace std;
```

```
class IMultimeter {
  public:
   virtual IMultimeter* clone() = 0;//tworzy kopie
                                    //danego obiektu
   virtual void measure() = 0;
   virtual ~ IMultimeter(){};
};
//-----------------------------------------------------------
class TFactory {
  public:
    static IMultimeter* create(int);
  private:
    static IMultimeter* prototypes[number];
};
//-----------------------------------------------------------
class TVoltmeter: public IMultimeter {
  public:
    IMultimeter* clone() {return new TVoltmeter;}
    void measure()
         {cout << "woltomierz zmierzył napięcie = 30
                   V\n\n";}
};
//-----------------------------------------------------------
class TAmmeter: public IMultimeter {
  public:
    IMultimeter* clone() {return new TAmmeter;}
    void measure()
          {cout << "amperomierz zmierzył prąd = 0.2
                   mA\n\n";}
};
//-----------------------------------------------------------
IMultimeter* TFactory::prototypes[] =
                       {0, new TVoltmeter, new TAmmeter};
IMultimeter* TFactory::create(int i)
                       {return prototypes[i]->clone();}
//-----------------------------------------------------------
int main()
{
  IMultimeter* instance[number];
  int j, i = 0;
  cout << "Wybierz miernik: Woltomierz: 1,
           Amperomierz: 2\n";
  cin >> j;
  while (j) {
```

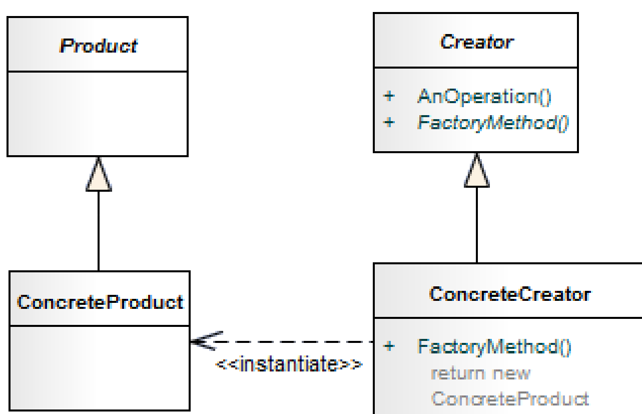
```
    instance[i++] = TFactory::create(j);
    cout << "Aby wyświetlić odczyt wprowadź kod: 0\n";
    cin >> j;
  }
  for (int j=0; j < i; j++)
    instance[j]->measure();
  for (int j=0; j < i; j++)
    delete instance[j];
  system("PAUSE");
  return 0;
}
```

Wynik działania programu:

```
wybierz miernik: Woltomierz: 1, Amperomierz: 2
1
Aby wyswietlic odczyt wprowadz kod: 0
0
woltomierz zmierzyl napiecie = 30 V

Aby kontynuować, naciśnij dowolny klawisz . . .
```

5.5. Metoda wytwórcza

Wzorzec metody wytwórczej dostarcza abstrakcji do tworzenia obiektów nieokreślonych, ale powiązanych typów, oraz umożliwia klasom dziedziczącym decydowanie jakiego typu ma to być obiekt. Dzięki metodzie wytwórczej klasy mogą zdać się na swoje podklasy w kwestii tworzenia obiektów klas zależnych. Wzorzec składa się z dwóch ról: produktu `Product` definiującego typ zasobów oraz kreatora `Creator` definiującego sposób ich tworzenia. Na rysunku 5.6 pokazano diagram klas omawianego wzorca.

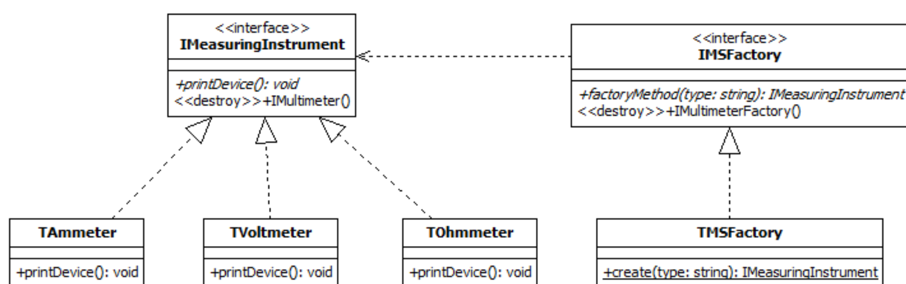


Rysunek. 5.6. Diagram wzorca metody wytwórczej

Kreator deklaruje operację wytwórczą `FactoryMethod()`, której typem

powrotnym jest typ konkretnego produktu. Produkt jest z reguły budowany w postaci interfejsu, który realizują klasy opisujące konkretne jego wystąpienia, które z kolei będą tworzone za pomocą metody wytwórczej.

Na rysunku 5.7 pokazano przykładowy diagram klas opisujący wzorzec metody wytwórczej z interfejsem `IMSFactory` deklarującym operację `factoryMethod()`, której implementacja służy do kreowania różnych typów urządzeń pomiarowych oraz interfejsem `IMeasuringInstrument` (przyrząd pomiarowy) określającym typ produktu. Klasy realizujące ten interfejs reprezentują bardziej szczegółowe postacie mierników laboratoryjnych. Pokazana na rysunku 5.7 odmiana wzorca metody wytwórczej wymaga implementacji podklas dla konkretnych obiektów opisujących konkretne przyrządy pomiarowe. Implementacja diagramu pokazana jest na listingu 5.4.



Rysunek 5.7. Metoda wytwórcza realizująca „fabrykę" przyrządów pomiarowych

Listing 5.4. Implementacja diagramu 5.7

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

class IMeasuringInstrument {
public:
  virtual void printDevice() = 0;
  virtual ~IMeasuringInstrument(){};
};
//-----------------------------------------------------------
class IMSFactory {
public:
  virtual IMeasuringInstrument*
          factoryMethod(const string& type) = 0;
  virtual ~IMSFactory(){};
};
//-----------------------------------------------------------
class TAmmeter: public IMeasuringInstrument {
public:
   void printDevice() {
```

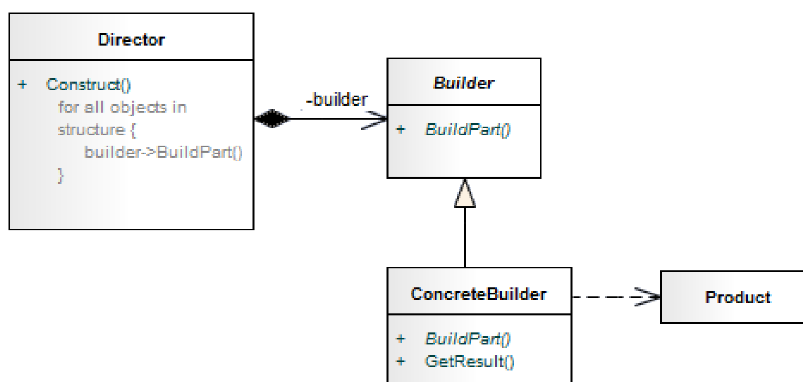
```
     cout << "Amperomierz w systemie" << endl;
   }
};
//----------------------------------------------------------
class TVoltmeter : public IMeasuringInstrument {
public:
   void printDevice() {
     cout << "Woltomierz w systemie "<< endl;
   }
};
//----------------------------------------------------------
class TOhmmeter : public IMeasuringInstrument {
public:
   void printDevice() {
     cout << "Omomierz w systemie" << endl;
   }
};
//----------------------------------------------------------
class TMSFactory: public IMSFactory {
public:
  IMeasuringInstrument* factoryMethod(const string& type) {
    if (type == "Amperomierz")
       return new TAmmeter();
      else if (type == "Omomierz")
         return new TOhmmeter();
        else
          return new TVoltmeter();
  }
};
//----------------------------------------------------------
int main() {
  IMSFactory *factory = new TMSFactory;
  auto_ptr<IMeasuringInstrument>
           ms(factory->factoryMethod("Amperomierz"));
  ms->printDevice();
  ms.reset(factory->factoryMethod("Omomierz"));
  ms->printDevice();
  ms.reset(factory->factoryMethod("Woltomierz"));
  ms->printDevice();
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Amperomierz w systemie
Omomierz w systemie
Woltomierz w systemie
```

5.6. Budowniczy

Wzorzec ten pozwala na odseparowanie konstrukcji złożonego obiektu od jego reprezentacji, umożliwiając powstanie w tym samym procesie konstrukcyjnym różnych reprezentacji złożonego obiektu. Budowniczy składa się z trzech ról: produktu `Product`, budowniczego `Builder` i dyrektora `Director`, tak jak pokazano to na rysunku 5.8.



Rysunek 5.8. Podstawowa reprezentacja wzorca budowniczego

Klasa abstrakcyjna `Builder` (często implementowana w postaci interfejsu) definiuje operację `buildPart()` służącą do tworzenia części składowych obiektu złożonego klasy `Product`. Operacje zaimplementowane w klasie `ConcereteBuilder` (konkretny budowniczy) konstruują i zestawią części produktu poprzez implementowanie funkcji klasy abstrakcyjnej `Builder`. Innymi słowy, konstruują wewnętrzną reprezentację produktu i definiuje proces jego składania. Operacja `Construct()` klasy `Director` ma za zadanie skonstruowanie konkretnego obiektu, używając interfejsu budowniczego. Listing 5.5 obrazuje jedną z możliwych implementacji diagramu z rysunku 5.8.

Listing 5.5. Implementacja diagramu 5.8

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

class Product {
  public:
    void addPart(string part) {
```

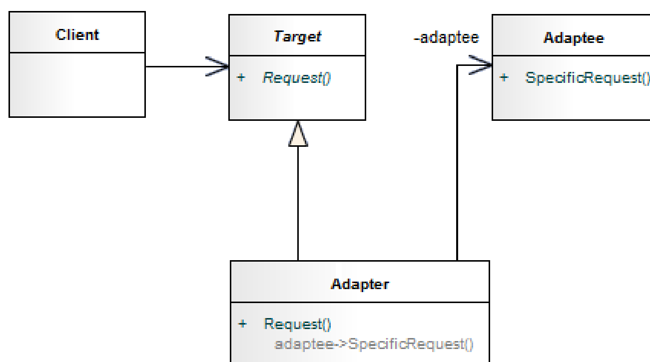
```
      cout << part << endl;
    }
};
//-----------------------------------------------------------
class Builder {
public:
  virtual void buildPartA(Product *product) = 0;
  virtual void buildPartB(Product *product) = 0;
  virtual ~Builder(){};
};
//-----------------------------------------------------------
class ConcreteBuilder : public Builder {
  public:
    void buildPartA(Product *product) {
       product->addPart("Dodana część pierwsza...");
    }
    void buildPartB(Product *product) {
       product->addPart("Dodana część druga...");
    }
};
//-----------------------------------------------------------
class Director {
  private:
    Builder *builder;
  public:
    Director() {
        builder = new ConcreteBuilder();
    }
    ~Director() {delete builder;};
    void construct() {
      builder->buildPartA(NULL);
      builder->buildPartB(NULL);
    }
};
//----------------------------------------------------------
int main()
{
  Director *director = new Director();
  director->construct();
  delete director;
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Dodana czesc pierwsza...
Dodana czesc druga...
```

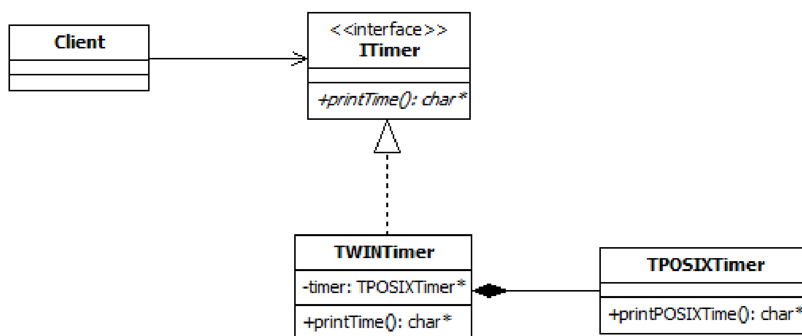
5.7. Adapter

Adapter służy do adaptacji interfejsów obiektowych, tak aby możliwa była współpraca obiektów o niezgodnych typach. Szczególnie istotną rolę odgrywa on w przypadku wykorzystania gotowych bibliotek o interfejsach niezgodnych ze stosowanymi w aplikacji. Struktura wzorca składa się z ról: `Target`, `Adaptee`, `Adapter` oraz klienta. `Target` jest abstrakcją z reguły implementowaną jako interfejs, którego oczekuje klient. Elementem dostarczającym żądanej przez klienta funkcjonalności jest `Adaptee`. Rolą adaptera, który implementuje typ `Target`, jest przetłumaczenie wywołania metod należących do typu `Target` poprzez wykonanie innych, specyficznych metod z klasy `Adaptee`. Dzięki temu klient współpracuje z obiektem klasy `Adapter` realizującym akceptowany przez siebie interfejs `Target`, jednocześnie wykorzystując funkcjonalność dostarczoną przez `Adaptee`, tak jak pokazuje to rysunek 5.9.



Rysunek 5.9. Podstawowa reprezentacja adaptera

Na rysunku 5.10 pokazano odmianę wzorca adaptera zastosowaną do problemu adaptowania interfejsu służącego do wywołania funkcji odczytującej aktualna datę i czas systemowy. Listing 5.6 obrazuje implementację diagramu 5.10. Konstrukcja oraz mechanizm wywołania funkcji odczytującej datę i czas w formie udostępnianej przez przenośny interfejs systemu operacyjnego POSIX różni się od jej odpowiednika, którym posługujemy się w systemach operacyjnych Windows. Aby ujednolicić sposób posługiwania się tego rodzaju funkcjami użytkownik zaadoptował wywołanie funkcji `printPOSIXTime()` na potrzeby wywołania funkcji `printTime()` interfejsu `ITimer`. Adaptacji tej dokonano w klasie `TWINTimer` będącej adapterem dla `TPOSIXTimer`.



Rysunek 5.10. Adapter wykorzystujący agregację całkowitą

Listing 5.6. Implementacja diagramu 5.10

```
#include <iostream>
#include <sysutils.hpp>//WIN
#include <ctime>//POSIX
using namespace std;

class ITimer {
  public:
    virtual char* printTime() = 0;
    virtual ~ITimer(){};
};
//----------------------------------------------------------
class TPOSIXTimer {
  public:
    char* printPOSIXTime() {
      static char buf[80];
      tm *timeNow;
      time_t secsNow;
      tzset();
      time(&secsNow);
      timeNow = localtime(&secsNow);
      strftime(buf, 80, "%M minut(y) po"
              " godzinie %I %A, %B %d 20%y\n\n", timeNow);
      return buf;
    }
};
//----------------------------------------------------------
class TWINTimer: public ITimer {
  public:
    char* printTime() {
      static char buf[80];
```

```
        char *ptr, nt[80];
        cout << "\Zegar WIN pokazał: " << DateTimeToStr(Now())
        << "\n\n";
        ptr = timer->printPOSIXTime();
        strcpy(nt, &ptr[0]);
        sprintf(buf, "%s", nt);
        return buf;
      }
      TWINTimer() {
          timer = new TPOSIXTimer();
      }
      ~TWINTimer() {delete timer;};
    private:
      TPOSIXTimer *timer;
};
//-----------------------------------------------------------
int main() //Klient
{
  ITimer *timer = new TWINTimer;
  char* ptr;
  ptr = timer->printTime();
  cout << "Zegar POSIX pokazał: " << ptr << endl;
  delete timer;
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Zegar WIN pokazal: 9/12/2010 10:58:30 AM

Zegar POSIX pokazal: 58 minut(y) po godzinie 10 Sunday, September 12 2010
```

Wzorzec adaptera może alternatywnie wykorzystywać dwa rodzaje relacji: agregację całkowitą oraz dziedziczenie po elementach prywatnych. Na rysunku 5.11 pokazano ten sam wzorzec, z tym że agregację klas `TPOSIXTimer` z `TWINTimer` zastąpiono dziedziczeniem po elementach prywatnych. Dziedziczenie po elementach prywatnych należy interpretować jako zawieranie się klasy `TWINTimer` w klasie `TPOSIXTimer`, co należy z kolei interpretować jako bezpośrednie dodanie nowej funkcjonalności do klasy `TWINTimer`. Na listingu 5.7 pokazano implementację omawianej odmiany wzorca adaptera.

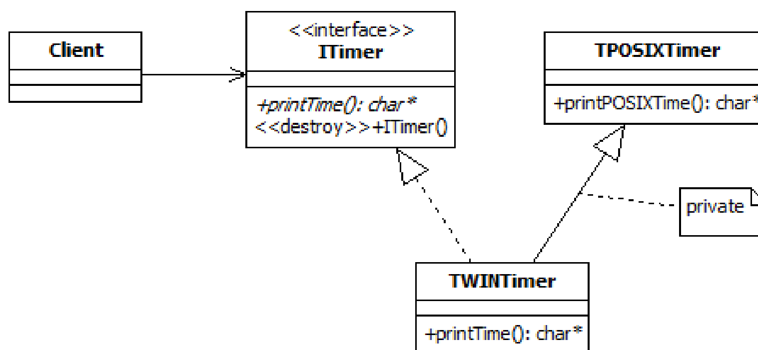
Listing 5.7. Implementacja wzorca adaptera z rysunku 5.11

```
#include <iostream>
#include <sysutils.hpp> //WIN
#include <ctime> //POSIX
```

```
using namespace std;

class ITimer {
  public:
    virtual char* printTime() = 0;
    virtual ~ITimer(){};
};
//-----------------------------------------------------------
class TPOSIXTimer { //POSIX
  public:
    char* printPOSIXTime() {
      static char buf[80];
      tm *timeNow;
      time_t secsNow;
      tzset();
      time(&secsNow);
      timeNow = localtime(&secsNow);
      strftime(buf, 80, "%M minut(y) po"
              " godzinie %I %A, %B %d 20%y\n\n", timeNow);
      return buf;
    }
};
//-----------------------------------------------------------
class TWINTimer: public ITimer, private TPOSIXTimer {
  public:
    char* printTime() {
      static char buf[80];
      char *ptr, nt[80];
      cout << "\Zegar WIN pokazał: " << DateTimeToStr(Now())
      << "\n\n";
      ptr = TPOSIXTimer::printPOSIXTime();
      strcpy(nt, &ptr[0]);
      sprintf(buf, "%s", nt);
      return buf;
    }
};
//-----------------------------------------------------------
int main()
{
  ITimer *timer = new TWINTimer;
  char* ptr;
  ptr = timer->printTime();
  cout << "Zegar POSIX pokazał: " << ptr << endl;
  delete timer;
  cin.get();
```

```
  return 0;
}
```

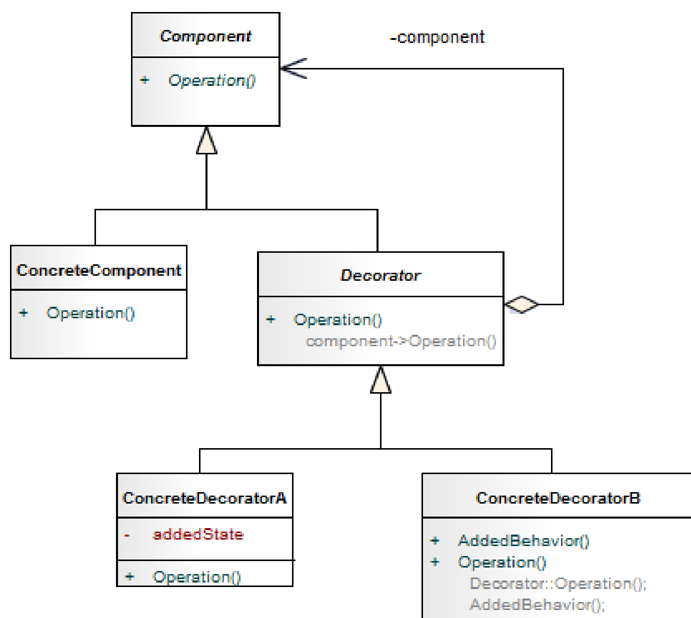
Wynik działania powyższego programu jest identyczny jak poprzednio.



Rysunek 5.11. Adapter wykorzystujący dziedziczenie prywatne

5.8. Dekorator

Dekorator jest wzorcem zbliżonym pod względem struktury do adaptera. Główny celem stosowania dekoratora jest stworzenie możliwości dodawania funkcjonalności do obiektu danej klasy w czasie wykonywania programu. Na rysunku 5.12 pokazano podstawową reprezentacje omawianego wzorca. Klasa abstrakcyjna `Component` jest z reguły implementowana w postaci interfejsu, co oznacza, że jest wspólnym interfejsem dla wszystkich klas, których funkcjonalność będzie poszerzana (klas dekorowanych). Interfejs ten realizuje klasa `ConcreteComponent`, która jest odpowiedzialna za podstawową funkcjonalność systemu. Klasa `Decorator` poprzez prywatny wskaźnik agreguje interfejs `Component`. Po otrzymaniu żądania wykonania określonej operacji `operation()`, dekorator deleguje żądanie do obiektu `Component`, a następnie opcjonalnie wykonuje dodatkową, specyficzną operację `AddedBehaviour()`. Dzięki temu dodanie do obiektu nowej funkcjonalności polega na utworzeniu dekoratora i przekazaniu mu obiektu dekorowanego jako wywołania. W sytuacji, gdy każdy element dekorujący (klasy `ConcreteDecoratorA` i `ConcreteDecoratorB`) dodaje do docelowego obiektu dekorowanego `ConcreteComponent` tylko jedną funkcję, wówczas dekorując ten sam docelowy obiekt wielokrotnie uzyskujemy efekt osiągnięcia żądanej sumarycznej funkcjonalności. Pod względem typu obiekt udekorowany nie różni się od obiektu wyjściowego (dla użytkownika dostępny jest on przez interfejs `Component`), dlatego wielokrotne stosowanie mechanizmu dekorującego wymaga w kodzie programu głównego wywoływania jedynie żądanych konstruktorów klas dekorujących. Na listingu 5.8 zilustrowano tę sytuację.



Rysunek 5.12. Podstawowa reprezentacja wzorca dekoratora

Listing. 5.8. Implementacja diagramu 5.12

```
#include <iostream>
using namespace std;
class Component {
  public:
    virtual ~Component(){ }
    virtual void operation() = 0;
};
//-----------------------------------------------------------
class ConcreteComponent : public Component {
  public:
    ~ConcreteComponent() {
       cout << "ConcreteComponent dtor\n";
    }
    void operation() {cout << "ConcreteComponent ";}
};
//-----------------------------------------------------------
class Decorator : public Component {
  private:
    Component* component;
  public:
    Decorator(Component* k) {component = k;}
   ~Decorator() {delete component;}
    void operation() {component->operation();}
```

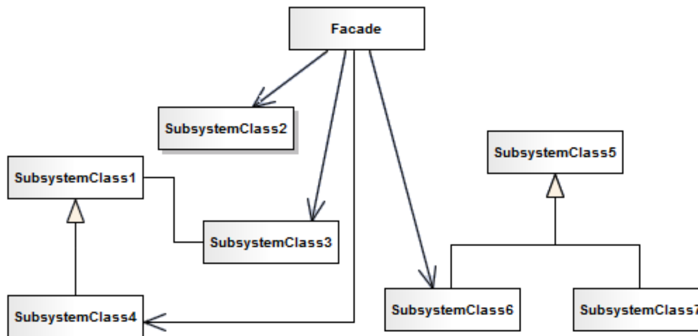
```
};
//------------------------------------------------------------
class ConcreteDecoratorA : public Decorator {
  public:
    ConcreteDecoratorA(Component* k):Decorator(k){}
    ~ConcreteDecoratorA() {
       cout << "ConcreteDecoratorA dtor ";
    }
    void operation() {
      Decorator::operation(); cout<<"ConcreteDecoratorA ";
    }
};
//------------------------------------------------------------
class ConcreteDecoratorB : public Decorator {
  public:
    ConcreteDecoratorB(Component* k):Decorator(k){}
   ~ConcreteDecoratorB() {
      cout<<"ConcreteDecoratorB dtor ";
    }
    void operation() {
      Decorator::operation(); cout<<"ConcreteDecoratorB ";
    }
};
//------------------------------------------------------------
int main() {
  Component* cA = new ConcreteDecoratorA(
                new ConcreteComponent);
  Component* cAB = new ConcreteDecoratorB(
                 new ConcreteDecoratorA(
                 new ConcreteComponent));
  cA->operation();
  cout << endl;
  cAB->operation();
  cout << endl;
  delete cA;
  delete cAB;
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
ConcreteComponent ConcreteDecoratorA
ConcreteComponent ConcreteDecoratorA ConcreteDecoratorB
ConcreteDecoratorA dtor ConcreteComponent dtor
ConcreteDecoratorB dtor ConcreteDecoratorA dtor ConcreteComponent dtor
```

5.9. Fasada

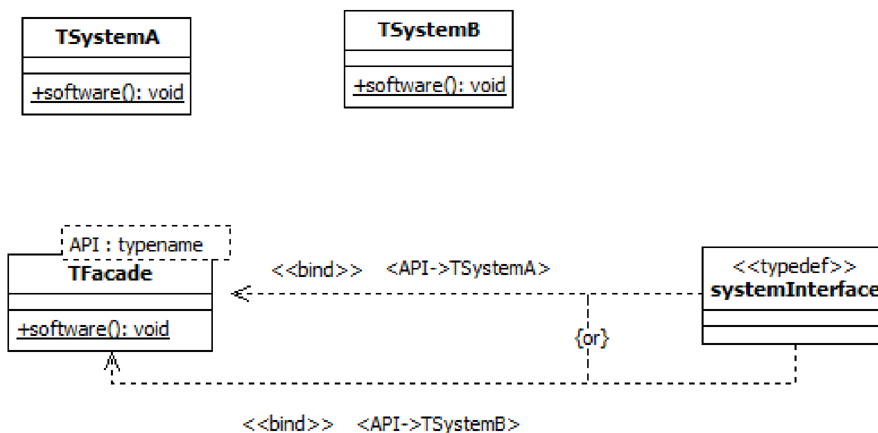
Fasada stanowi wygodny środek dostępu do złożonego systemu prezentując na zewnątrz uproszczony i uporządkowany interfejs programistyczny. W swojej podstawowej postaci fasada implementowana jest w sposób bardzo prosty, tzn. w postaci klasy powiązanej z klasami reprezentującymi system, do którego użytkownik pragnie uzyskać dostęp. Rysunek 5.13 prezentuje tego rodzaju interpretację wzorca fasady.



Rysunek 5.13. Ogólna reprezentacja wzorca fasady

Jednak zdarzają się sytuacje, w których istnieje potrzeba skorzystania z istniejącego już, rozbudowanego systemu w postaci bibliotek programistycznych, których funkcjonalność może być różnie określona w zależności od wyboru platformy systemowej. Wygodnie jest wówczas zaprojektować interfejs użytkownika jako element pośredniczący między elementami właściwymi dla różnych systemów operacyjnych, tak, aby ukryć złożoność wykorzystywanych systemów przez dostarczenie jednolitego, udokumentowanego publicznego API (interfejsu użytkownika). Skutkiem takiego podejścia jest zdefiniowanie dozwolonego dostępu do obiektów różnych systemów, dzięki czemu ilość możliwych przypadków ich błędnego użycia jest zredukowana do minimum. Kolejną korzyścią z zastosowania wzorca fasady jest to, że użytkownik korzystający z wybranego systemu musi zapoznać się jedynie z API (interfejsem programistycznym) fasady a nie funkcjami wszystkich obiektów systemu.

Na rysunku 5.14 pokazano sytuację, w której zdefiniowane są dwie klasy reprezentujące dwie różne biblioteki, które posiadają tę samą funkcjonalność, jednak przeznaczone są do wykonywanie w różnych systemach operacyjnych, np. Windows (`TSystemA`) oraz Linux (`TSystemB`). Zadaniem klasy parametryzowanej (szablonu) `TFacade` jest dostarczenie użytkownikowi publicznego API, którym będzie mógł się posłużyć niezależnie od tego w którym systemie operacyjnym kompiluje swój program. Fasada powinna automatycznie rozpoznać aktualny system operacyjny i załadować odpowiednią bibliotekę (w tym przykładzie reprezentowaną przez klasę odpowiednio `TSystemA` lub `TSystemB`).



Rysunek 5.14. Parametryzowana fasada

Listing 5.9. Implementacje diagramu 5.14

```
#ifndef __FACADE_H__
#define __FACADE_H__

#include <iostream>
using namespace std;

class TSystemA {
  public:
    inline static void software(/*...*/)
    {cout << "Oprogramowanie systemu dla Windows";}
};
//-----------------------------------------------------------
class TSystemB {
  public:
    inline static void software(/*...*/)
    {cout << "Oprogramowanie systemu dla Linux";}
};
//-----------------------------------------------------------
template<typename API>
class TFacade {
   public:
     inline static void software()
     {return API::software();}
};
//-----------------------------------------------------------
#if defined(__LINUX__) || defined(__linux__)
   typedef TFacade<TSystemB> systemInterface;
```

```
#elif defined(__WIN32__) || defined(__WIN64__)
   typedef TFacade<TSystemA> systemInterface;
#else
  #error Nieznana platforma systemowa
#endif
//-----------------------------------------------------------
#endif
```

Listing 5.10. Program użytkownika

```
#include "Facade.h"
int main()
{
  systemInterface::software(/*...*/);
  cout << endl;
  cin.get();
  return 0;
}
```

Wynik działania programu w systemie Windows:

```
Oprogramowanie systemu dla Windows
```

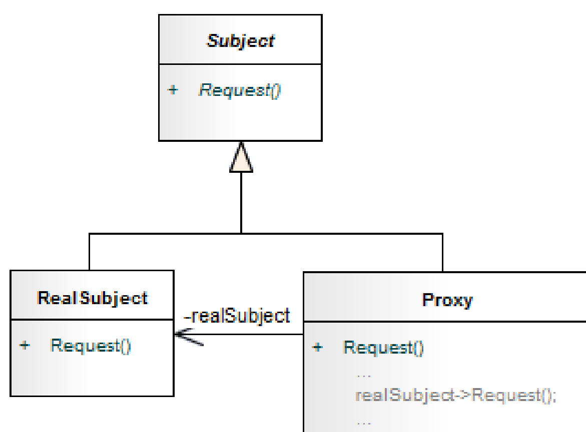
Wynik działania programu w systemie Linux:

```
Oprogramowanie systemu dla Linux
```

5.10. Pełnomocnik

Centralnym elementem wzorca jest powiązanie `Proxy – RealSubject`. Klasa `RealSubject` posiada funkcjonalność `Request()` wymaganą przez klienta. Klasa `Proxy` jest powiązana z `RealSubject` i kontroluje dostęp do niej. Celem takiego powiązania klas jest umożliwienie zastąpienia obiektu konstruowanego na bazie klasy `RealSubject` obiektem konstruowanym na bazie klasy `Proxy`. W takiej sytuacji, klient zamiast do obiektu docelowego konstruowanego na bazie klasy `RealSubject`, odwołuje się do obiektu pośredniczącego konstruowanego na bazie klasy `Proxy`, który deleguje żądania `Request()` kierowane do obiektu docelowego lub próbuje obsłużyć je samodzielnie. W szczególności obiekt klasy `Proxy` może utworzyć obiekt klasy `RealSubject` znacznie później niż klient zechce korzystać z niego, a tym samym opóźnić proces jego kreowania. Tak jak pokazano na rysunku 5.15 jednym z atrybutów klasy `Proxy` jest prywatna zmienna `realSubject` typu `RealSubject`. Dzięki występowaniu tego typu powiązania operacje klasy `Proxy` mogą delegować żądania kierowane do `RealSubject` kontrolując dostęp do nich. Klasa `RealSubject` jest źródłem definicji rzeczywistego obiektu wymagającego kontroli i ochrony. Obiekt klasy `Proxy` pełni główną rolę we wzorcu. Oznacza

to, iż zarządza on powiązanym obiektem klasy `RealSubject` i podejmuje decyzje dotyczące utworzenia go, przekazania mu sterowania, itp. W ten sposób obiekt klasy `Proxy` pełni funkcje ochronne (uniemożliwia nieautoryzowany dostęp) oraz kontrolne w stosunku do obiektu klasy `RealSubject`. Element abstrakcyjny `Subject` definiuje wspólny interfejs, poprzez który odbywa się wymiana komunikatów między klientem a powiązaniem `Proxy` – `RealSubject`. Należy wspomnieć, iż występowanie elementu abstrakcyjnego nie zawsze jest wymagane do poprawnej implementacji tego wzorca. Często można ograniczyć się do uwzględnienia jedynie powiązania kierunkowego `Proxy`–`RealSubject` będącego istotą wzorca pełnomocnika.



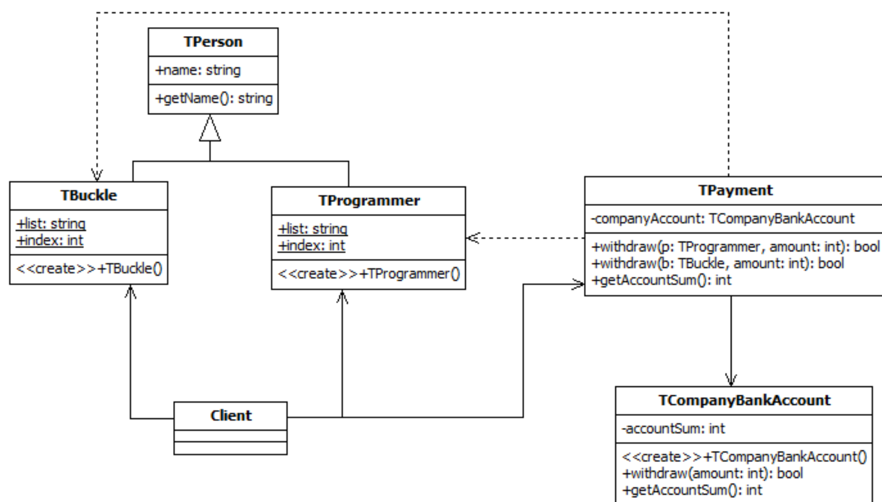
Rysunek 5.15. Podstawowa implementacja wzorca pośrednika

Istnieją trzy podstawowe rodzaje wzorca pośrednika:

- Pośrednik zdalny (ang. *remote proxy*) służący do reprezentacji obiektu znajdującego się w innej przestrzeni adresowej, np. na innym komputerze. Dzięki temu dla lokalnych klientów wszystkie odwołania są pozornie lokalne. Pośrednik przejmuje wówczas odpowiedzialność za zdalne wywołania metod oraz odbieranie wyników. Mechanizm ten jest stosowany w większości środowisk przetwarzania rozproszonego np. CORBA lub EJB.
- Pośrednik wirtualny (ang. *virtual proxy*) wykorzystywany w sytuacjach, w których należy zastąpić obiekt docelowy charakteryzujący się dużymi wymaganiami zużycia zasobów sprzętowych, np. alokujący duży obszar pamięci. Aby opóźnić (a niekiedy nawet zastąpić) proces tworzenia takiego obiektu, pośrednik obsługuje wszystkie zadania obiektu docelowego, które nie wymagają odwołań do obszaru pamięci zajmowanego przez obiekt docelowy.
- Pośrednik zabezpieczający (ang. *protected proxy*) zajmuje się zabezpieczeniem odwołań do obiektu docelowego przed nieautoryzowanym dostępem.

> Obiekt docelowy nigdy nie jest bezpośrednio dostępny dla klientów; w ich imieniu występuje pełnomocnik, który określa, którym z nich można udostępnić usługi oferowane przez obiekt docelowy, a którym nie.

Na rysunku 5.16 pokazano statyczny diagram klas opisujący najprostszy schemat funkcjonowania przedsiębiorstwa. Diagram ten składa się z trzech ról: roli opisującej grupy pracowników, roli opisującej konto bankowe przedsiębiorstwa i sposób dokonywania przelewów pieniężnych oraz roli klienta, reprezentującego osobę (np. księgową) dokonującą poleceń przelewów konkretnych kwot na konta pracownicze.



Rysunek 5.16. Jedna z możliwych odmian pośrednika zabezpieczającego

Grupy pracowników: sprzątaczki i programiści opisywane są odpowiednio przez dwie klasy `TBuckle` oraz `TProgrammer`. Klasa `TCompanyBankAccount` opisuje konto bankowe firmy. Klient nie może bezpośrednio pobrać kwot pieniężnych z konta przedsiębiorstwa. Operacje wypłaty `withdraw()` może wykonać jedynie pośrednio z wykorzystaniem klasy `TPayment` reprezentującej polecenie wypłaty. Polecenie wypłaty jest pełnomocnikiem (elementem pośredniczącym) pomiędzy pracownikami, księgową a kontem bankowym firmy. Każdorazowe zadysponowanie polecenia wypłaty dla konkretnych osób skutkuje zmniejszeniem wartości przechowywanej w atrybucie `accountSum` klasy `TCompanyBankAccount`. Listing 5.11 przedstawia implementację omawianego wzorca opisującego pośrednika zabezpieczającego.

Listing. 5.11. Szczegółowa implementacja diagramu 5.16

```
#include <iostream>
using namespace std;
class TPerson {
```

```
  public:
    string name;
    string getName() const {return name;}
};
//-----------------------------------------------------------
class TProgrammer: public TPerson {
  public:
    static string list[];
    static int index;
    TProgrammer() {name = list[index++];}
};
//-----------------------------------------------------------
string TProgrammer::list[] = {"Wacek", "Janek",
                              "Jola", "Kasia"};
int TProgrammer::index = 0;
//-----------------------------------------------------------
class TBuckle: public TPerson {
  public:
    static string list[];
    static int index;
    TBuckle() {name = list[index++];}
};
//-----------------------------------------------------------
string TBuckle::list[] = {"Waclawa", "Wieslawa",
                          "Weronika","Klementyna"};
int TBuckle::index = 0;
//-----------------------------------------------------------
class TCompanyBankAccount {  //konto firmy
  private:
    int accountSum;
  public:
   //Suma początkowa zgromadzona na koncie firmy = 1800 zł
    TCompanyBankAccount() {accountSum = 1800;}
    bool withdraw(int amount) {
      if (amount > accountSum)
        return false;
      accountSum -= amount;
     return true; }
    int getAccountSum() const {return accountSum;}
};
//-----------------------------------------------------------
class TPayment { //zapłata
  private:
    TCompanyBankAccount companyAccount;
  public:
```

```
    bool withdraw(TProgrammer& p, int amount) {
      if(p.getName()=="Wacek" || p.getName()=="Janek"
                              || p.getName()=="Jola")
        return companyAccount.withdraw(amount);
        else
          return false;
    }
    bool withdraw(TBuckle& b, int amount) {
      if(b.getName()=="Waclawa" || b.getName()=="Klementyna"
                                || b.getName()=="Wieslawa")
        return companyAccount.withdraw(amount);
        else
          return false;
    }
    int getAccountSum() {
      return companyAccount.getAccountSum();
    }
};
//-----------------------------------------------------------
int main()
{
  TPayment payment;
  TProgrammer programmers[4];
  TBuckle bucklies[4];
  for(int i=0, amount=100; i < 4; i++, amount += 100)
    if(! payment.withdraw(programmers[i], amount))
      cout<<programmers[i].getName()<<" nie otrzymał(a)
                                  wypłaty\n";
     else
       cout << amount << " zł otrzymał(a) "
            << programmers[i].getName() << '\n';
  cout << "Na koncie firmy pozostało "
       << payment.getAccountSum() << " zł\n\n";

  for(int i=0, amount=100; i < 4; i++, amount += 50)
    if(! payment.withdraw(bucklies[i], amount))
      cout<<bucklies[i].getName()<<" nie otrzymała
        wypłaty\n";
     else
       cout << amount << " zł otrzymała "
            << bucklies[i].getName() << '\n';
  cout << "Na koncie firmy pozostalo "
       << payment.getAccountSum() << " zł\n\n";

  cin.get();
```

```
  return 0;
}
```

Wynik działania programu:

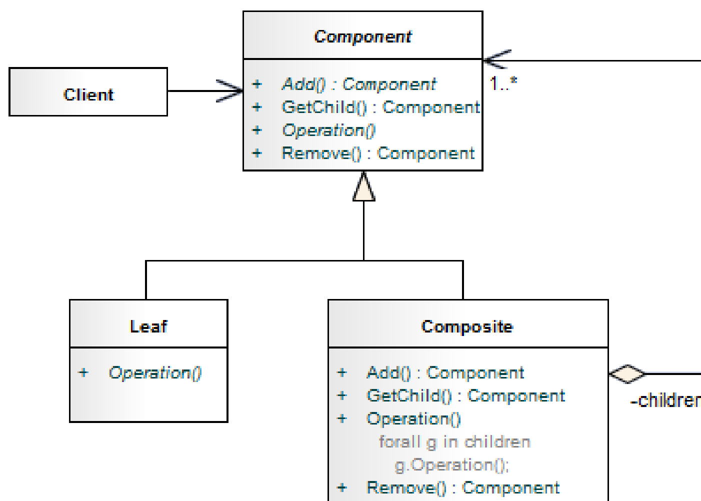
```
100 zl otrzymal(a) Wacek
200 zl otrzymal(a) Janek
300 zl otrzymal(a) Jola
Kasia nie otrzymal(a) wyplaty
Na koncie firmy pozostalo 1200 zl

100 zl otrzymala Waclawa
150 zl otrzymala Wieslawa
Weronika nie otrzymala wyplaty
250 zl otrzymala Klementyna
Na koncie firmy pozostalo 700 zl
```

5.11. Kompozyt

Celem wzorca kompozytu jest budowa hierarchii obiektów. Wzorzec opisuje, jak w oparciu o klasę finalną (klasę nie posiadającą potomków, klasę liść) zbudować drzewo obiektów połączonych mechanizmem dziedziczenia. Rysunek 5.17 przedstawia diagram generalizacji opisywany przez wzorzec kompozytu. Element abstrakcyjny `Component` deklaruje operacje umożliwiające dodawanie, usuwanie oraz odwoływanie się do każdego elementu drzewa obiektów. W klasie `Component` zdefiniowana jest także operacja `operation()`, która powinna być zaimplementowana przez każdy węzeł struktury obiektów. Klasa `Component` (często implementowana w postaci interfejsu) jest realizowana przez dwie klasy: `Leaf` oraz `Composite`. Klasa `Leaf` reprezentuje obiekty, które nie posiadają potomków (czyli liście w strukturze), natomiast `Composite` jest dowolnym węzłem pośrednim. Każdy węzeł pośredni może zarządzać poddrzewem, którego jest korzeniem, dlatego metoda `operation()`, poza wykonaniem operacji specyficznych dla każdego węzła, wywołuje swoje odpowiedniki w obiektach potomnych, w ten sposób propagując żądane wywołanie. Z punktu widzenia programu klienta taka struktura umożliwia zarządzanie całością za pomocą jednego obiektu – korzenia drzewa. Niepotrzebna jest także wiedza o rozmiarze drzewa, ponieważ wywołanie zostanie przekazane automatycznie do wszystkich jego elementów.

Standardowym mechanizmem implementacji wzorca kompozytu jest opakowanie finalnej klasy `Leaf` poprzez odpowiednie zdefiniowanie klas `Composite` i `Component`. W niektórych językach programowania istnieje możliwość zdefiniowania funkcji pobierającej jako argument klasę i zwracającą inną klasę. Jednak większość popularnych języków, takich jak C++ i Java nie zezwala na zdefiniowanie takiej funkcji. Wynika to z faktu, iż funkcja tego typu powinna definiować nową klasę bazową. W większości języków możliwa jest definicja nowej klasy dziedziczącej, ale nie nowej klasy bazowej. Ograniczenia te nie dyskwalifikują jednak wzorca kompozytu, gdyż pisząc np. w C++, Javie, C# lub Object Pascalu zawsze można wykorzystać omawianą już w tej książce

koncepcję delegowania.



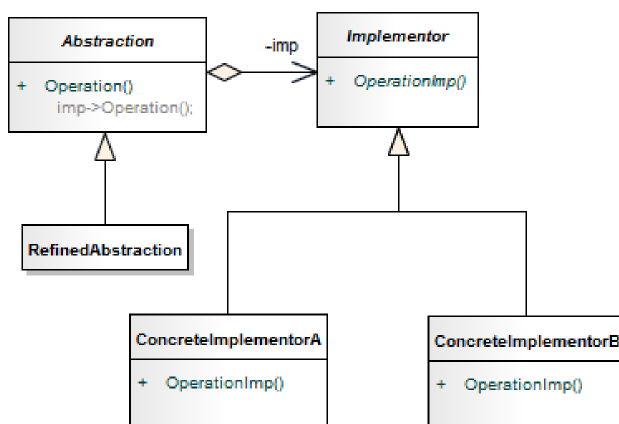
Rysunek 5.17. Statyczny diagram klas opisujący wzorzec kompozytu

5.12. Most

Istnieją sytuacje, w których klasy pochodne klasy abstrakcyjnej używają wielu różnych implementacji, co w konsekwencji prowadzi do nadmiernego wzrostu liczy implementacji w kodzie. Jednym z rozwiązań tego problemu jest zdefiniowanie wspólnego interfejsu dla wszystkich implementacji i wykorzystywanie go jako elementu wspólnego przez klasy pochodne klasy abstrakcyjnej. Zastosowanie pokazanego na rysunku 5.18 wzorca mostu umożliwia usunięcie nadmiernej liczby powiązań pomiędzy zbiorem implementacji a zbiorem korzystających z nich obiektów. Innymi słowy, wzorzec mostu odseparowuje interfejs od implementacji, w taki sposób, aby oba elementy mogły istnieć niezależnie, a co za tym idzie by powstała możliwość wprowadzania zmian do implementacji bez konieczności zmian w kodzie, który korzysta z klasy. Tak jak przedstawia to rysunek 5.18, element abstrakcyjny `Abstraction` deklaruje interfejs implementowanych obiektów. Element abstrakcyjny `Implementor` deklaruje interfejs dla klas stanowiących implementację. Obiekty klas realizujących interfejs `Abstraction` wykorzystują obiekty klas realizujących interfejs `Implementor`, nie posiadając wiedzy, którą z klas realizujących (`ConcreteImplementorA` lub `ConcreteImplementorB`) aktualnie reprezentują. Poprzez usunięcie powiązań pomiędzy różnymi implementacjami a obiektami, które je wykorzystują w znaczącym stopniu łatwiejsze staje się ewentualne późniejsze rozbudowanie systemu, z tego względu iż obiekty tracą wiedzę na temat swojej implementacji.

Praktyczne wykorzystanie wzorca mostu w takich językach jak Java i C# nie stanowi większego problemu, gdyż języki te posiadają wbudowane

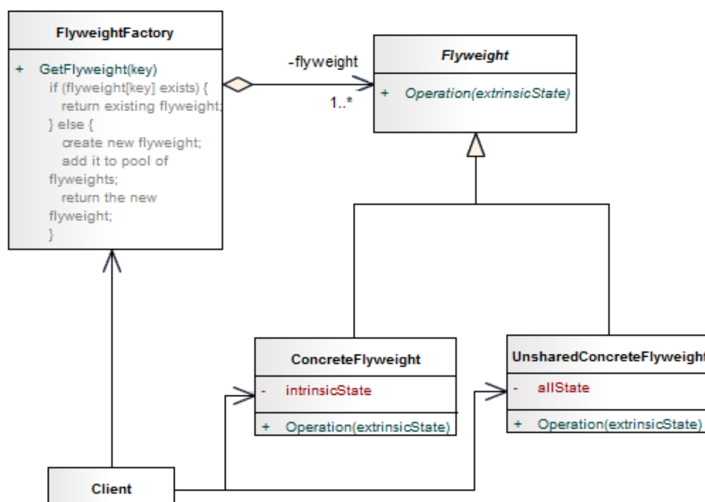
mechanizmy zautomatyzowanego zarządzania pamięcią. System zarządzający pamięcią w stosownym momencie jest w stanie przywrócić pamięć zajmowaną przez nieużywane obiekty implementacji. Z zupełnie inną sytuacją spotkamy się programując w C++ lub Object Pascalu. W językach tych należy samodzielnie zarządzać pamięcią przydzieloną obiektom implementacji. W tym celu należy przede wszystkim kontrolować liczbę obiektów implementacji, np. poprzez wykorzystanie zmiennej wskaźnikowej wskazującej na aktualnie używane obiekty (zmienna taka powinna spełniać rolę licznika obiektów). W celu kontrolowania ilości kreowanych obiektów można też z powodzeniem wykorzystać wzorzec singletonu.



Rysunek 5.18. Statyczny diagram klas opisujący wzorzec mostu

5.13. Pyłek

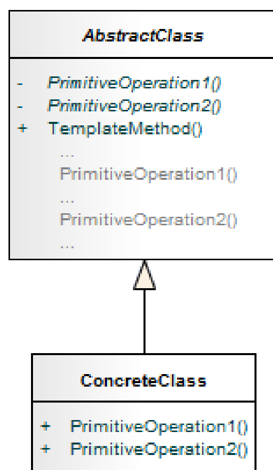
Istotą omawianego wzorca jest zaprezentowanie prostego mechanizmu współdzielenia wielu obiektów o niewielkim rozmiarze w celu zwiększenia wydajności systemu pod względem zużycia pamięci. Pyłek zajmuje się udostępnianiem pojedynczego obiektu wielu klientom. Wspólny dostęp znajduje zastosowanie wówczas, gdy występuje potrzeba zarządzania dużą ilością obiektów, które aktualnie mogą znajdować się w takim samym stanie. Aby zrealizować ten mechanizm zaproponowano podział danych przechowywanych w atrybutach obiektów na współdzielone dane wewnętrzne `intristicState` oraz niewspółdzielone, unikatowe dla każdego obiektu dane zewnętrzne `allState`, tak jak pokazano to na rysunku 5.19. Wzorzec składa się z czterech głównych elementów. Element abstrakcyjny `Flyweight` definiuje operacje służące do przyjmowania i odtwarzania stanu zewnętrznego obiektu opisywanego przez klasę `UnsharedConcreteFlyweight`. Obiekt tworzony na bazie klasy `ConcreteFlyweight` przechowuje stan wewnętrzny (współdzielony) obiektu i jest niezależny od kontekstu wywołania. Zadaniem fabryki `FlyweightFactory` jest kreowanie i składowanie obiektów realizujących interfejs `Flyweight`.



Rysunek 5.19. Statyczny diagram klas opisujący wzorzec pyłku

5.14. Metoda szablonu

Wzorzec metody szablonu pozwala zdefiniować szkielet algorytmu, który jest implementowany poprzez funkcję szablonową `TemplateMethod()` w klasie bazowej `AbstractClass`. Uszczegółowieniem sposobu działania ogólnego algorytmu implementowanego przez funkcję szablonową zajmują się tzw. operacje prymitywne, które powinny być definiowane jako operacje abstrakcyjne. Operacje te są implementowane w klasie dziedziczącej `ConcreteClass`.

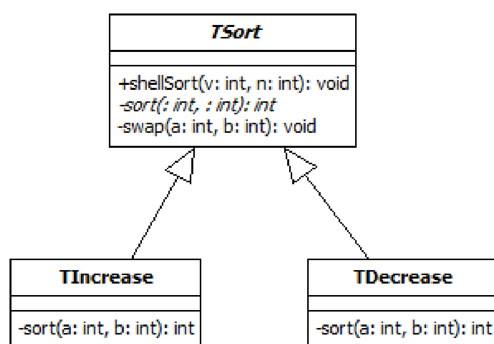


Rysunek 5.20. Podstawowa reprezentacja wzorca metody szablonu

W momencie utworzenia obiektu na bazie klasy `ConcreteClass` i wywołania publicznej funkcji szablonowej tworzony jest egzemplarz algorytmu opisywanego przez funkcję szablonową z uszczegółowieniem sposobu jego wykonania. Uszczegółowienie wykonywania algorytmu następuje poprzez wywołanie operacji prymitywnych na rzecz funkcji szablonowej, tak jak pokazano to na rysunku 5.20. Metoda szablonu umożliwia ponowne zdefiniowania operacji prymitywnych bez konieczności zmieniania struktury podstawowego algorytmu.

Metoda szablonu leży u podstaw wielu metod implementacji wykorzystywanych w językach ogólnego przeznaczenia (językach wieloparadygmatowych) takich jak C++ i Object Pascal. W specjalizowanych językach czysto obiektowych, takich jak Java i C# wzorzec ten jest rzadko implementowany ze względu na niską efektywność.

Na rysunku 5.21 zaprezentowano statyczny diagram klas opisujący metodę szablonu zastosowaną do problemu sortowania ciągu liczb. Szkielet ogólnego algorytmu sortowania metodą Shella został zaimplementowany w publicznej metodzie `shellSort()` klasy abstrakcyjnej `TSort`. Uszczegółowieniem wykonania algorytmu, tzn. określeniem „kierunku" sortowania zajmuje się prywatna czysto wirtualna (abstrakcyjna) operacja prymitywna `sort()`, która każdorazowo wywoływana jest na rzecz metody szablonowej. Operacja ta jest odpowiednio implementowana w klasach dziedziczących `TIncrease` oraz `TDecrease`. W zależności od tego, w jaki sposób użytkownik chce posortować generowane w sposób losowy dane wejściowe – tworzy anonimowy obiekt na bazie klasy `TDecrease` lub/i `TIncrease`, a następnie wywołuje publiczną metodę szablonową `shellSort()`, tak jak pokazano to na listingu 5.12.



Rysunek 5.21. Metoda szablonu zastosowana do problemu sortowania liczb

Listing 5.12. Implementacja diagramu 5.21

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#define size 10
using namespace std;
```

```
class TSort {
  public:
    void shellSort(int *v, int n) {//sortowanie metodą
                                   // Shella
      for (int g = n/2; g > 0; g /= 2)
        for (int i = g; i < n; i++)
          for (int j = i-g; j >= 0;j -= g)
             if (sort(v[j],v[j+g])) //wywołanie funkcji
                                    //prymitywnej sort()
               swap(v[j], v[j+g]);
    }
  private:
    virtual int sort(int, int) = 0;
    void swap(int& a, int& b) {int t = a; a = b; b = t; }
};
//-----------------------------------------------------------
class TIncrease: public TSort {
  private:
    int sort(int a, int b) {return (a > b); }
};
//-----------------------------------------------------------
class TDecrease : public TSort {
  private:
    int sort(int a, int b) {return (a < b); }
};
//-----------------------------------------------------------
int main()
{
  srand((unsigned int)time((time_t *)NULL));
  int array[size];
  cout << "wektor wyjściowy:\n";
  for (int i=0; i < size; i++) {
    array[i] = rand() % 100;
    cout << array[i] << ' ';
  }
  cout << endl;
  TIncrease *increase = new TIncrease;
  increase->shellSort(array, size);
  cout << "posortowany rosnąco:\n";
  for (int i=0; i < size; i++)
    cout << array[i] << ' ';
  cout << endl;
  TDecrease *decrease = new TDecrease;
  decrease->shellSort(array, size);
```

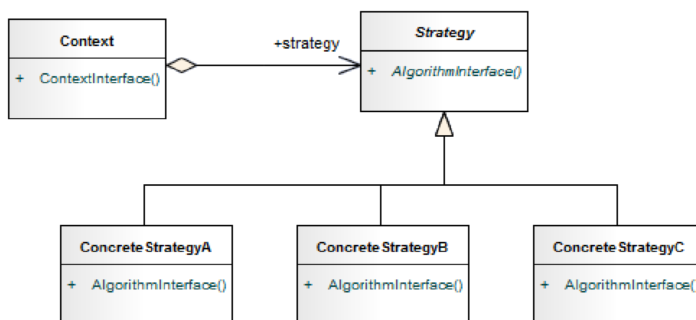
```
    cout << "posortowany malejąco:\n";
    for (int i=0; i < size; i++)
        cout << array[i] << ' ';
    cout << endl;
    delete increase;
    delete decrease;
    cin.get();
    return 0;
}
```

Wynik działania programu:

```
wektor wyjsciowy:
74 81 72 53 63 57 49 44 12 12
posortowany rosnaco:
12 12 44 49 53 57 63 72 74 81
posortowany malejaco:
81 74 72 63 57 53 49 44 12 12
```

5.15. Strategia

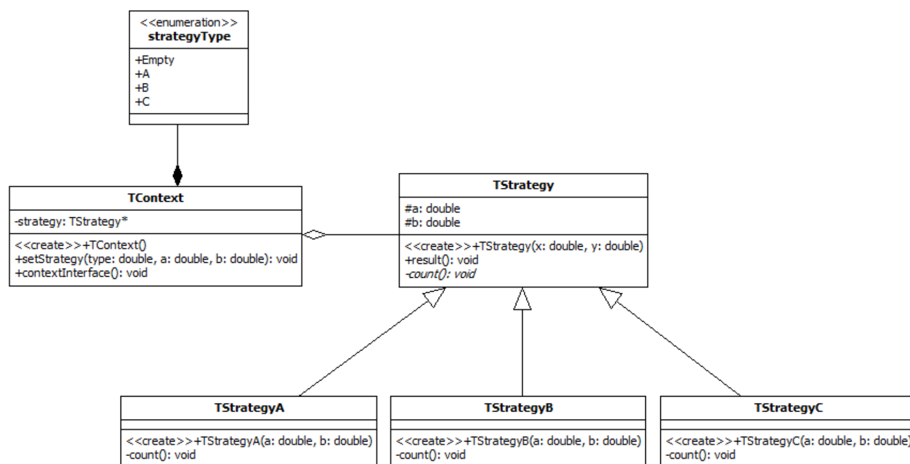
Wzorzec strategii wykorzystywany jest w sytuacjach, w których należy ukryć szczegóły algorytmu przed użytkownikiem. Ponadto, umożliwia on wykorzystanie różnych wersji algorytmu w zależności od kontekstu ich dalszego użycia. Na rysunku 5.22 pokazano diagram omawianego wzorca.



Rysunek. 5.22. Podstawowa reprezentacja wzorca strategii

Centralnym elementem wzorca jest klasa polimorficzna (lub abstrakcyjna) `Strategy`, która jest klasą bazową dla klas `ConcreteStrategyA`, `ConcreteStrategyB` oraz `ConcreteStrategyC` implementujących różne algorytmy lub wersje tego samego algorytmu. W klasie `Strategy` definiowana jest funkcja określająca sposób wywoływania algorytmu. Każda z klas pochodnych implementuje dany algorytm we własnym zakresie. Obiekt klasy `Context` agreguje obiekty klas `ConcreteStrategyA`, `ConcreteStrategyB` oraz `ConcreteStrategyC` odwołując się do nich poprzez klasę `Strategy`.

Na rysunku 5.23 zaprezentowano rozbudowany wzorzec strategii, za pomocą którego można odseparować wybór wersji algorytmu (dodawanie, odejmowanie, mnożenie liczb) od jego implementacji. Wybór algorytmu dokonywany jest na podstawie kontekstu jego wywołania w funkcji `setStrategy()`. Taka konstrukcja powoduje, iż standardowe instrukcje warunkowe lub instrukcje wyboru mogą zostać wyeliminowane z kodu. Wszystkie algorytmy są wywoływane w identyczny sposób, tak jak pokazano to na listingu 5.13.



Rysunek 5.23. Strategia wyboru podstawowych działań arytmetycznych

Listing 5.13. Implementacja diagramu 5.23

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

class TStrategy;

class TContext { //Kontekst - działanie
  public:
    enum strategyType {Empty, A, C, B};
    TContext() {strategy = NULL;}
    void setStrategy(double type, double a, double b);
    void contextInterface();//interfejs wewnętrzny kontekstu
  private:
    TStrategy* strategy;
};
//-----------------------------------------------------------
class TStrategy { // Strategia
  public:
```

```
    TStrategy(double x, double y){ a = x; b=y; }
    virtual ~TStrategy(){}
    void result(){count();}
  protected:
    double a, b;
  private:
    virtual void count() = 0; // interfejs algorytmu
};
//-------------------------------------------------------
class TStrategyA: public TStrategy {//Konkretna strategia A
  public:
    TStrategyA(double a, double b): TStrategy(a, b) { }
  private:
    void count() {
       cout << "wynik dodawania = " << a+b << endl;}
};
//-------------------------------------------------------
class TStrategyC: public TStrategy {//Konkretna strategia C
  public:
    TStrategyC(double a, double b): TStrategy(a, b) { }
  private:
    void count() {
      cout << "wynik odejmowania = " << a-b << endl; }
};
//-------------------------------------------------------
class TStrategyB: public TStrategy {//Konkretna strategia B
  public:
    TStrategyB(double a, double b ): TStrategy(a, b) { }
  private:
    void count() {
      cout << "wynik mnozenia= " << a*b << endl;}
};
//-------------------------------------------------------
void TContext::setStrategy(double type, double a, double b){
  delete strategy;
  if (type == A) strategy = new TStrategyA(a, b);
    else if (type == C) strategy = new TStrategyC(a, b);
      else if (type == B) strategy = new TStrategyB(a, b);
}
//-------------------------------------------------------
void TContext::contextInterface() {
  strategy->result();
}
//-------------------------------------------------------
int main() {
```

```
  TContext context;
  unsigned int option;
  long double a,b;
  cout<<"wyjście(0) dodawanie(1) odejmowanie(2)\
         możenie(3): ";
  cin>>option;
  while (option) {
    cout << "podaj liczbę a= ";
    cin >> a;
    cout << "podaj liczbę b= ";
    cin >> b;
    context.setStrategy(option, a, b);
    context.contextInterface();
    cout<<"wyjście(0) dodawanie(1) odejmowanie(2)\
           mnożenie(3): ";
    cin>>option;
  }
  cin.get();
  return 0;
}
```

Wynik działania programu:

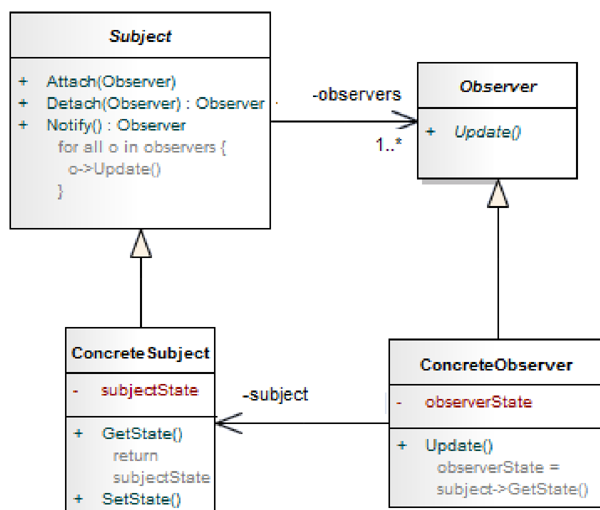
```
wyjscie(0) dodawanie(1) odejmowanie(2) mozenie(3): 1
podaj liczbe a= 3
podaj liczbe b= 3
wynik dodawania = 6
wyjscie(0) dodawanie(1) odejmowanie(2) mozenie(3): 2
podaj liczbe a= 3
podaj liczbe b= 3
wynik odejmowania = 0
wyjscie(0) dodawanie(1) odejmowanie(2) mozenie(3): 3
podaj liczbe a= 3
podaj liczbe b= 3
wynik mnozenia= 9
wyjscie(0) dodawanie(1) odejmowanie(2) mozenie(3):
```

5.16. Obserwator

Głównym obszarem stosowalności wzorca Obserwator jest stworzenie relacji typu jeden-do-wielu łączącej grupę obiektów. Dzięki zastosowaniu wzorca zmiana stanu obiektu po stronie „jeden" umożliwi automatyczne powiadomienie o niej wszystkich innych dołączanych elementów (tzw. obserwatorów). Wzorzec składa się z dwóch ról: obiektu obserwowanego reprezentowanego przez klasę dziedziczącą po abstrakcji `Subject` oraz obserwatorów realizujących interfejs `Observer`. Klasa abstrakcyjna `Subject` definiuje operacje `attach()` i `detach()` pozwalające odpowiednio na dołączanie i odłączanie obserwatorów: każdy zainteresowany obiekt poprzez interfejs `Observer` może zarejestrować się jako obserwator w klasie `Subject`. W klasie tej zdefiniowana jest też

operacja `notify()`, służąca do powiadamiania wszystkich zarejestrowanych obserwatorów o zmianie stanu obiektu obserwowanego poprzez wywołanie w pętli na ich rzecz metody `update()`, która jest zadeklarowana w interfejsie `Observer`. Operacja ta jest implementowana w klasie realizującej interfejs i służy do powiadamiania konkretnego obserwatora `ConcreteObserver` o zmianie stanu obiektu obserwowanego `ConcreteSubject`.

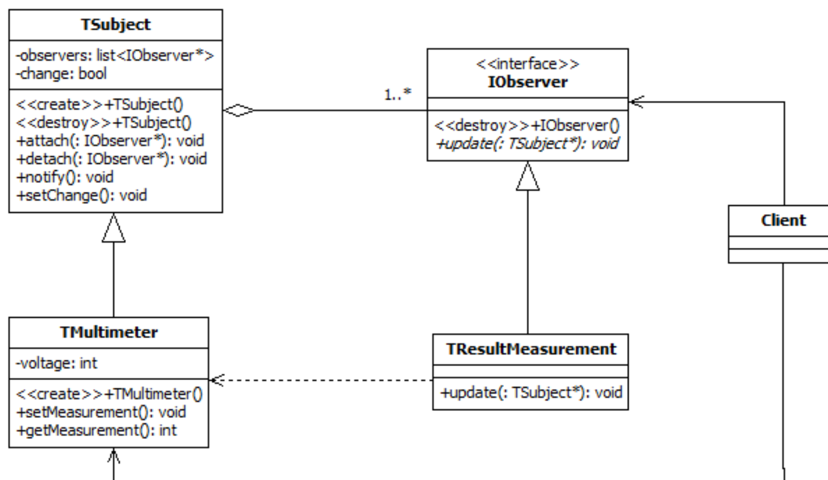
Na rysunku 5.24 zaprezentowana statyczny diagram klas opisujący wzorzec obserwatora. Konstrukcja obserwatora umożliwia ograniczenie liczby powiązań i zależności występujących pomiędzy obiektami obserwującymi (dołączanymi) a obiektami obserwowanymi (podstawowymi).



Rysunek 5.24. Podstawowa reprezentacja wzorca obserwatora

Na rysunku 5.25 pokazano odmianę wzorca obserwatora, którą zastosowano do problemu odczytywania aktualnych wskazań miernika, w chwili, gdy miernik dokona odczytu wartości mierzonej przechowywanej w atrybucie `voltage` klasy `TMultimeter`. Odczyt wartości mierzonej powoduje zmianę stanu miernika. W chwili wykrycia zmiany stanu miernika dołączany jest obserwator. Wynik pomiaru staje się konkretnym obserwatorem. Obserwowany obiekt (miernik uniwersalny) posiada referencje do obserwatorów, jednak jego „wiedza” na temat elementu, który będzie dołączany ogranicza się tylko i wyłącznie do znajomości interfejsu `IObserver`. Również obserwatorzy nie posiadają wiedzy na temat obserwowanego przyrządu, gdyż metoda `update()` wywoływana jest w sposób asynchroniczny. Ponieważ ilość informacji przekazywanych obiektom realizującym interfejs `IObserver` może istotnie wpływać na wydajność systemu, dlatego istnieją dwa podejścia do implementacji omawianego wzorca. W modelu *push* każdy obserwator otrzymuje w postaci parametru metody `update()` pełną informację o stanie obiektu klasy `TSubject`. W

modelu *pull* obserwatorzy otrzymują tylko wskaźnik do obserwowanego obiektu klasy `TSubject`, dzięki czemu mogą następnie uzyskać informację o szczegółach dotyczących zmiany jego stanu. Na listingu 5.14 zaprezentowano implementacje omawianego przykładu w wersji *pull*.



Rysunek 5.25. Jedna z możliwych reprezentacji odmiany *pull* wzorca obserwatora

Listing 5.14. Implementacja diagramu 5.25

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

class TSubject;

class IObserver { //Interfejs obserwatora
  public:
    virtual ~IObserver(){};
    virtual void update(TSubject* ) = 0;
};
//-----------------------------------------------------------
class TSubject { //Obserwowany przedmiot
  public:
    TSubject();
    virtual ~TSubject(){};
    virtual void attach(IObserver*);//dołącza(rejestruje)
                   //anonimowe obiekty klasy obserwator
    virtual void detach(IObserver*); //usuwa anonimowe
                  //obiekty klasy obserwator
    virtual void notify();   //powiadamia o wystąpieniu
```

```
                              //zdarzenia
    virtual void setChange(); //rejestruje zmianę stanu
                               //pojedynczego obiektu
  private:
    list<IObserver*> observers;//dwukierunkowa lista
                                //obiektów klasy obserwator
                                //list jest kontenerem
                                //dwukierunkowym
    bool change; //reprezentuje stan pojedynczego
                 //obiektu
};
//-----------------------------------------------------------
void TSubject::setChange() {
  change = true ;
}
//-----------------------------------------------------------
//początkowo przedmiot jest w stanie nieaktywnym
TSubject::TSubject(): change(false) {}
//-----------------------------------------------------------
//dodaje obserwatora na koniec kontenera
void TSubject::attach (IObserver *o) {
  observers.push_back(o);
}
//-----------------------------------------------------------
void TSubject::detach (IObserver* o) {
  observers.remove(o);
}
//-----------------------------------------------------------
void TSubject::notify () {
  //powiadamia obserwatora o zmianie stanu
  if (change) {
    list<IObserver*>::iterator i = observers.begin();
    for ( ; i!= observers.end(); i++ ) {
      (*i)->update(this);
    }
    change = false;
  }
}
//-----------------------------------------------------------
//konkretnym przedmiotem jest miernik uniwersalny
class TMultimeter: public TSubject {
  private:
    int voltage;
  public:
    TMultimeter();
```

```
    void setMeasurement(); //dokonuje pomiaru
    int getMeasurement() const;  //zwraca wynik pomiaru
};
//-------------------------------------------------------
TMultimeter::TMultimeter() : voltage(225) {}
//-------------------------------------------------------
void TMultimeter::setMeasurement() {
  voltage++;
  setChange();
}
//-------------------------------------------------------
int TMultimeter::getMeasurement() const {
  return voltage; //miernik "odczytuje" napięcie prądu
}
//-------------------------------------------------------
//wynik pomiaru jako konkretny obserwator
class TResultMeasurement: public IObserver {
public:
  virtual void update(TSubject*);
};
//-------------------------------------------------------
void TResultMeasurement::update(TSubject *o) {
  //polimorficzne rzutowanie czasu wykonania
  TMultimeter *m = dynamic_cast<TMultimeter *>(o);
  if (m) {
    cout << "V = " << m->getMeasurement();
  }
}
//-------------------------------------------------------
int main()
{
  //konktetny przedmiot
  TMultimeter *multimeter = new TMultimeter();
  //konkretny obserwator
  IObserver *resultMeasurement=new TResultMeasurement();
  //dołączenie (zarejestrowanie) konkretnego obserwatora
  multimeter->attach(resultMeasurement);
  for(int i = 0; i < 5; i++) {
    multimeter->setMeasurement();
    //powiadamia o pomiarze
    multimeter->notify();
    cout << endl;
  }
  //usunięcie (wyrejestrowanie) konkretnego obserwatora
  multimeter->detach(resultMeasurement);
```

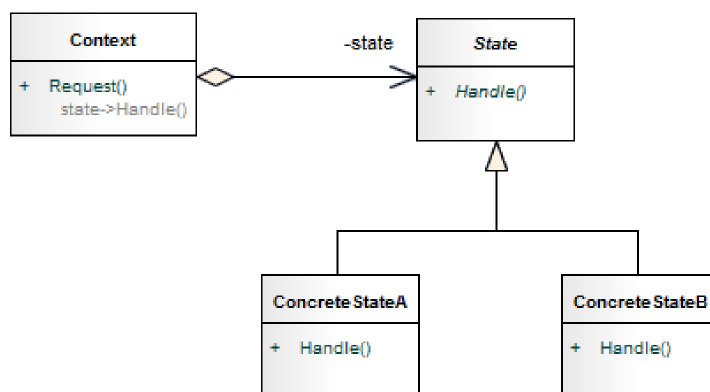
```
  delete multimeter;
  delete resultMeasurement;
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
U = 226
U = 227
U = 228
U = 229
U = 230
```

5.17. Stan

W trakcie pisania programów często pojawiającym się problemem jest konieczność zaprojektowania klasy, której obiekt potencjalnie będzie znajdować się w jednym z kilku możliwych stanów. Najprostszy sposób rozwiązania tego typu problemu polega na odpowiednim zmodyfikowaniu kodu wybranych operacji. Należy jednak zwrócić uwagę, iż podejście takie w praktyce jest mało wydajne. Innym rozwiązaniem jest wykorzystanie wzorca stanu. Wzorzec ten pozwala na wyodrębnienie w osobnych klasach każdego ze stanów w jakim potencjalnie może znaleźć się obiekt danej klasy. W tej konwencji stan obiektu nie będzie reprezentowany poprzez proste przypisanie wartości do odpowiednich atrybutów klasy. Oznacza to, iż konstruowany obiekt będzie obiektem stanowym, zaś zmiana jego stanu oznacza zmianę reprezentującego obiektu. Tak jak pokazano na rysunku 5.26 centralnym elementem wzorca stanu jest obiekt klasy `Context`.

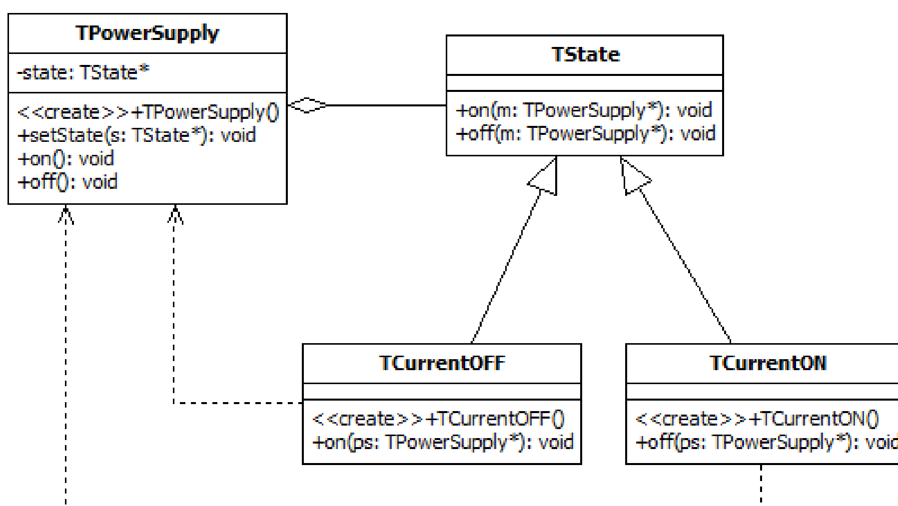


Rysunek. 5.26. Podstawowa reprezentacja wzorca stanu

Metody zaimplementowane w tej klasie wywoływane są w programie głównym. Ich zadaniem jest przekazywanie żądań do zagregowanego z obiektem klasy `Context` interfejsu `State` realizowanego przez klasy

`ConcreteStateA` i `ConcreteStateB` reprezentujące stany, w jakich aktualnie może pozostawać obiekt klasy `Context`. Ponieważ operacje interfejsu `State` są polimorficzne, zatem wraz ze zmianą implementacji automatycznie zmienia się też ich funkcjonalność. W konsekwencji prowadzi to do sytuacji, w której zmiana stanu obiektów realizujących interfejs `State` powoduje zmianę zachowania metod obiektu klasy `Context`. Oznacza to, że obiekt klasy `Context` pozornie zmienia klasę, do której należy.

Na rysunku 5.27 zaadoptowano wzorzec Stanu do problemu śledzenia zamian stanu urządzenia laboratoryjnego w postaci zasilacza wysokiego napięcia. Urządzenie takie może znajdować się dwóch głównych dyskretnych stanach: może być włączone (stan ON) lub wyłączone (stan OFF). Przejścia międzystanowe realizowane są poprzez operacje `on()` i `off()` odpowiednio włączające i wyłączające zasilacz. Dodatkowo, przejścia takie kontrolowane są poprzez spełnienie odpowiednich warunków dozoru zaimplementowanych w klasach `TCurrentON` i `TCurrentOFF`, tak jak pokazano to na listingu 5.15. Rysunek 5.28 obrazuje diagram zamian stanu urządzenia symulowanego przez kod z listingu 5.15. Urządzenie może sekwencyjnie pracować pomiędzy stanami ON i OFF.



Rysunek 5.27. Symulacja działania zasilacza wysokiego napięcia

Listing 5.15. Implementacja diagramu 5.27

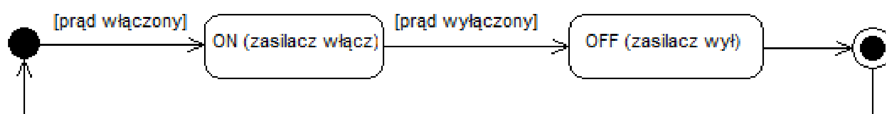
```
#include <iostream>
using namespace std;
//dwa stany, dwa zdarzenia
class TState;
class TPowerSupply {
    private:
      TState* state;
```

```
    public:
      TPowerSupply();
      void setState(TState* s) {state = s;}
      void on();
      void off();
};
//-----------------------------------------------------------
class TState {
  public:
     virtual void on(TPowerSupply* m) {
       cout << " Zasilacz włączony (ON)\n"; }
     virtual void off(TPowerSupply* m) {
       cout << " Zasilacz wyłąaczony (OFF)\n"; }
};
//-----------------------------------------------------------
void TPowerSupply::on() {
  state->on(this);
}
//-----------------------------------------------------------
void TPowerSupply::off() {
  state->off(this);
}
//-----------------------------------------------------------
class TCurrentON: public TState {
  public:
    TCurrentON() {cout << " prąd włączony\n";};
    void off(TPowerSupply* ps);
};
//-----------------------------------------------------------
class TCurrentOFF: public TState {
  public:
    TCurrentOFF() {cout << " prąd wyłączony\n";};
    void on(TPowerSupply* ps) {
      ps->setState(new TCurrentON());
      delete this;
    }
};
//-----------------------------------------------------------
void TCurrentON::off(TPowerSupply* ps) {
  ps->setState(new TCurrentOFF());
  delete this;
}
//-----------------------------------------------------------
TPowerSupply::TPowerSupply() {
  state = new TCurrentOFF(); cout << endl;
```

```
}
//----------------------------------------------------------
int main()
{
  void (TPowerSupply::*ptr[])() =
                  {TPowerSupply::off, TPowerSupply::on};
  TPowerSupply powerSupply;
  int option;
  while(true) {
    cout << "Enter 0-OFF, 1-ON, 2-exit:\n\n";
    cin >> option;
    if (option == 2)
      exit(EXIT_SUCCESS);
      else
       (powerSupply.*ptr[option])();
  }
  return 0;
}
```

Wynik działania programu:

```
 prad wylaczony

Enter 0-OFF, 1-ON, 2-exit:

1
 prad wlaczony
Enter 0-OFF, 1-ON, 2-exit:

0
 prad wylaczony
Enter 0-OFF, 1-ON, 2-exit:
```

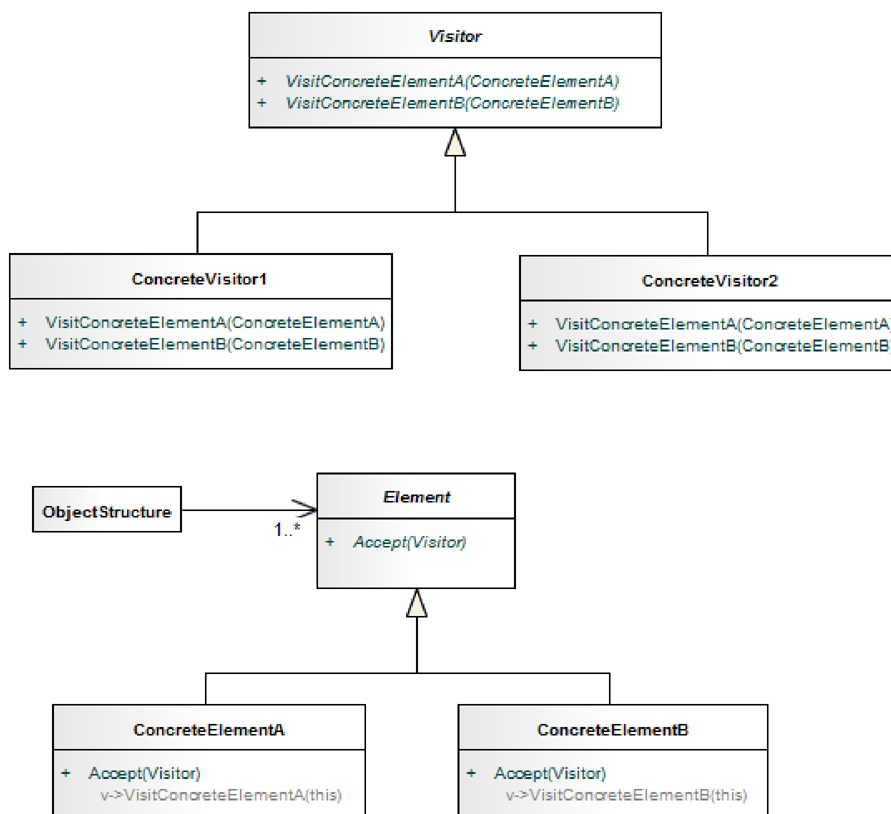


Rysunek. 5.28. Diagram stanów sekwencyjnych zasilacza wysokiego napięcia

5.18. Wizytator

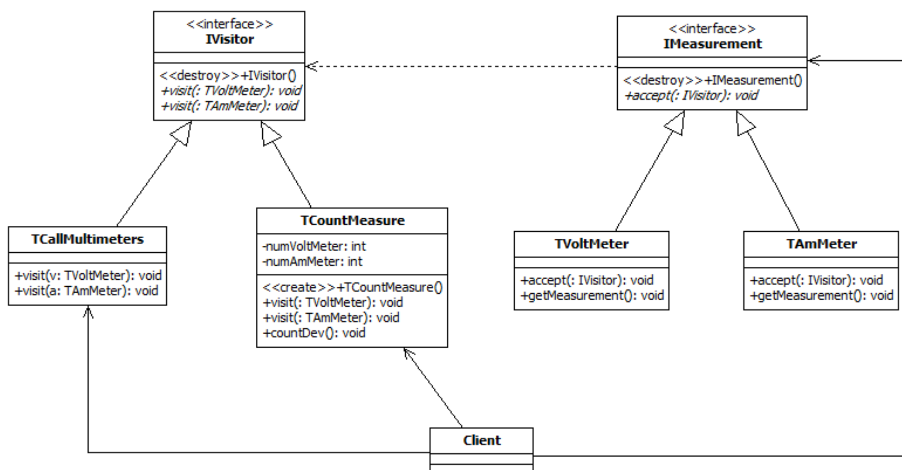
Stosowalność wzorca wizytatora ujawnia się w sytuacjach, w których należy wykonać tą samą operację na każdym obiekcie występującym w systemie. Obiekty będące częścią systemu powinny być do siebie funkcjonalnie podobne, choć niekoniecznie posiadać taki sam interfejs lub należeć do tej samej hierarchii. Tak jak zaprezentowano na rysunku 5.29, obiekty klas realizujących abstrakcję `Visitor` odwiedzają (wizytują) obiekty klas reprezentujących konkretne elementy systemu. Konkretny wizytator prosi konkretny odwiedzany obiekt o

akceptację wizyty. Akceptacja polega na wywołaniu przez odwiedzany obiekt metody `VisitConcreteElementA()` lub `VisitConcreteElementB()`.W ten sposób następuje podwójne wywołanie polimorficzne. W pierwszym wiązaniu ustalany jest konkretny wizytator, który został przekazany do obiektu w metodzie `Accept()`. W drugim ustalana jest odpowiednia metoda `VisitXXX()` dla danego typu obiektu.



Rysunek 5.29. Podstawowa reprezentacja wzorca wizytatora

Na rysunku 5.30 zaprezentowano odmianę wzorca wizytatora zastosowaną do problemu odczytywania aktualnych wskazań urządzeń pomiarowych. Konkretne, występujące w systemie urządzenia pomiarowe są obiektami klas `TAmMeter` (amperomierz) i `TVoltMeter` (woltomierz). Interfejs opisujący wizytatora realizowany jest przez dwa konkretne elementy wizytujące: klasę `TCallMultimeters`, której metody `visit()` umożliwiają odwiedzanie (zlokalizowanie) konkretnego urządzenia oraz klasę `TCountMeasure`, której operacje pozwalają na kontrolę urządzeń, z których odebrano odczyty. Listing 5.16 obrazuje implementację omawianego modelu.



Rysunek 5.30. Odczytywanie wskazań urządzeń pomiarowych

Listing 5.16. Implementacja diagramu 5.30

```
#include <iostream>
using namespace std;
class IVisitor;
class IMeasurement {
  public:
    virtual ~IMeasurement(){};
    virtual void accept(IVisitor&) = 0;
};
//-----------------------------------------------------------
class TVoltMeter: public IMeasurement{
  public:
    void accept(IVisitor&);
    void getMeasurement() {cout<<"odczyt z woltomierza\n";}
};
//-----------------------------------------------------------
class TAmMeter: public IMeasurement{
  public:
    void accept(IVisitor&);
    void getMeasurement() {cout<<"odczyt z amperomierza\n";}
};
//-----------------------------------------------------------
class IVisitor{
  public:
    virtual ~IVisitor(){};
    virtual void visit(TVoltMeter&) = 0;
    virtual void visit(TAmMeter&) = 0;
};
```

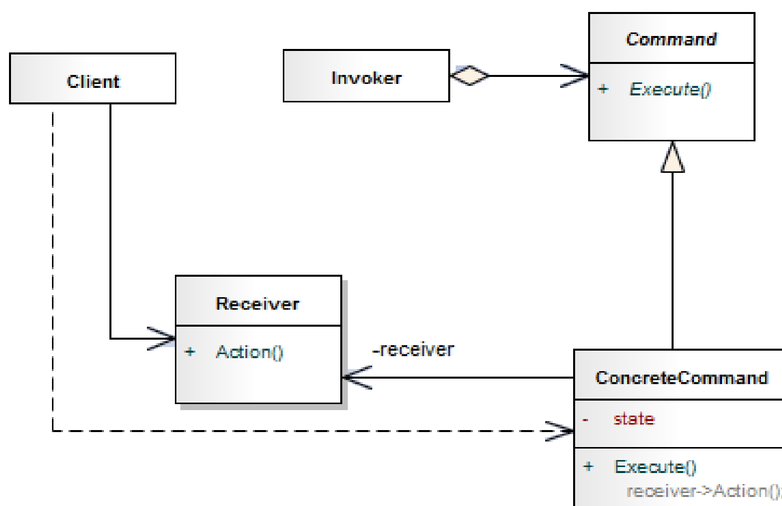
```
//---------------------------------------------------------
class TCountMeasure: public IVisitor{
  public:
    TCountMeasure() {numVoltMeter = numAmMeter = 0;}
    void visit(TVoltMeter&) {numVoltMeter++;}
    void visit(TAmMeter&) {numAmMeter++;}
    void countDev() {cout<<"liczba woltomierzy w systemie:"
                     << numVoltMeter <<
                     ", liczba amperomierzy w systemie: "
                     << numAmMeter << endl;}
  private:
    int numVoltMeter;
    int numAmMeter;
};
//---------------------------------------------------------
class TCallMultimeters: public IVisitor{
  public:
    void visit(TVoltMeter& v) {v.getMeasurement();}
    void visit(TAmMeter& a) {a.getMeasurement();}
};
//---------------------------------------------------------
void TVoltMeter::accept(IVisitor& v) {v.visit(*this);}
void TAmMeter::accept(IVisitor& v) {v.visit(*this);}
//---------------------------------------------------------
int main()
{
  IMeasurement* setOfMeasurements[] = {new TVoltMeter,
                             new TAmMeter,new TAmMeter,
                             new TVoltMeter, 0};
  TCountMeasure countMeasure;
  TCallMultimeters callMultimeters;
  for(int i=0; setOfMeasurements[i]; i++) {
    setOfMeasurements[i]->accept(countMeasure);
    setOfMeasurements[i]->accept(callMultimeters);
  }
  countMeasure.countDev();
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
odczyt z woltomierza
odczyt z amperomierza
odczyt z amperomierza
odczyt z woltomierza
liczba woltomierzy w systemie: 2, liczba amperomierzy w systemie: 2
```

5.19. Polecenie

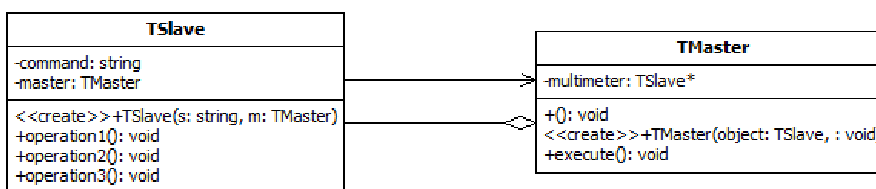
Zaprezentowany na rysunku 5.31 wzorzec polecenia stosowany jest w celu hermetyzowania poleceń do wykonania w postaci obiektów, w ten sposób, aby można było traktować je w sposób abstrakcyjny i przekazywać innym częściom kodu jako parametry wywołania.



Rysunek 5.31. Podstawowa reprezentacja wzorca polecenia

Centralnym elementem wzorca jest interfejs `Command`, deklarujący operację `Execute()` reprezentującą polecenie do wykonania. Operacja ta jest implementowana w klasie `ConcreteCommand` w postaci polecenia wykonania określonej operacji (akcji) `Action()` na obiekcie klasy `Receiver`, która jest powiązana z klasą `ConcreteCommand`. Obiekt klasy `Receiver` jest przedmiotem akcji deklarowanej w interfejsie `Command`. Klasa `Invoker` reprezentuje obiekt inicjujący wywołanie operacji interfejsu. Warto zauważyć, iż klient nie uzyskuje bezpośredniego dostępu zarówno do interfejsu `Command`, jak i obiektu klasy `Invoker`. Komunikuje się jedynie z obiektami klas `Receiver` i `ConcreteCommand`. Inicjatorem przetwarzania komunikatów jest obiekt klasy `Invoker`, który zarządza interfejsem `Command`. W momencie nadejścia żądania wykonania określonej operacji, `Invoker` parametryzuje skojarzony z nią interfejs `Command` właściwym odbiorcą działań, czyli obiektem `Receiver`. Następnie wywołuje metodę `Execute()` powodując określone skutki w obiekcie `Receiver`, widoczne dla programu klienta.

W języku C++ istnieje możliwość posłużenia się wskaźnikiem do funkcji. Oznacza to, iż w C++ implementacja wzorca polecenia znacznie się upraszcza. Na rysunku 5.32 pokazano diagram klas, którego implementacja przedstawiona na listingu 5.17 daje te same efekty co pełna implementacja wzorca polecenia.



Rysunek 5.32. Uproszczony wariant wzorca polecenia stosowany w C++

Listing. 5.17. Implementacja diagramu 5.32

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

class TSlave;

class TMaster { // Klasa TMaster agreguje TSlave
  private:
    TSlave* multimeter;
  public:
    void (TSlave::*operation)(); // wskaźnik do funkcji

    TMaster(TSlave* object = NULL,
            void(TSlave::*op)() = NULL) {
            multimeter = object;
            operation = op;
    }
    void execute() {(multimeter->*operation)();}
};
//-----------------------------------------------------------
class TSlave {
  private:
    string command;
    TMaster master;
  public:
    TSlave(string s, TMaster m) : master(m) {
           command = s;}
    void operation1() {
      cout << command<<" pomiar natężenia prądu" << endl;
      master.execute();}
    void operation2() {
      cout << command<<" pomiar oporu w obwodzie" << endl;
      master.execute();}
    void operation3() {
      cout << command<<" pomiar napięcia prądu" << endl;}
```

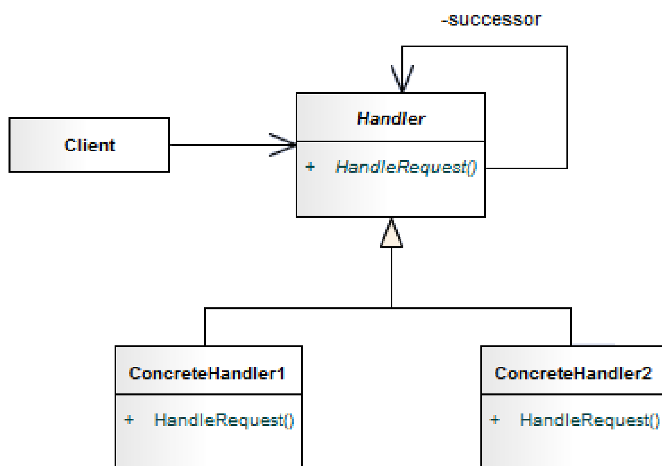
```
};
//-----------------------------------------------------------
int main() {
  TSlave slave1("Amperomierz: ", TMaster());
  TSlave slave2("Woltomierz: ",
                TMaster(&slave1, &TSlave::operation1));
  TSlave slave3("Omomierz: ", TMaster(&slave2,
                &TSlave::operation3));
  slave3.operation2();
  cin.get();
  return 0;
}
```

Wynik działania programu:

```
Omomierz:  pomiar oporu w obwodzie
Woltomierz:  pomiar napiecia pradu
```

5.20. Łańcuch odpowiedzialności

Centralnym elementem wzorca jest interfejs `Handler` realizowany przez sekwencję (łańcuch) klas `ConcreteHandlerN`. Każda z klas realizujących interfejs obsługuje jeden rodzaj żądania, pozostałe zaś przekazuje do następnego elementu w łańcuchu klas. Wynika stąd, iż każda z klas `ConcreteHandler` powinna posiadać referencję do następnej klasy w łańcuchu. `Client` inicjuje przetwarzanie danych, przekazując żądanie `HandleRequest()` do pierwszego w łańcuchu obiektu realizującego interfejs.



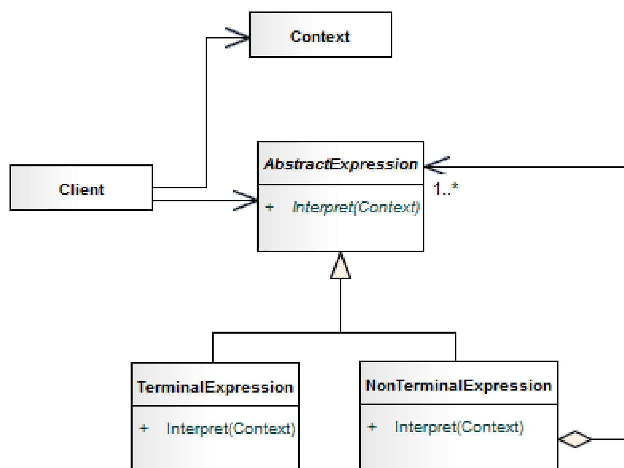
Rysunek 5.33. Podstawowa reprezentacja łańcucha odpowiedzialności

Żądania zwykle są implementowane w postaci jednej metody realizującej jakiś algorytm: jeżeli pierwszy napotkany obiekt typu `ConcreteHandler` jest w

stanie obsłużyć żądanie, to obsługuje je; w przeciwnym wypadku przekazuje je do następnego elementu w łańcuchu. Charakterystyczną cechą wzorca jest dowolna konfigurowalność łańcucha obiektów. Oznacza to, iż żaden jego element nie musi posiadać wiedzy o rodzaju żądań obsługiwanych przez kolejne elementy, dlatego zmiany w jego strukturze nie mają wpływu na zachowanie całości. Na rysunku 5.33 pokazano standardową reprezentacje omawianego wzorca.

5.21. Interpreter

Interpreter jest wzorcem projektowym, którego głównym zadaniem jest interpretacja poleceń innego języka. Dany język rozkładany jest na części gramatyczne, które reprezentowane są przez odpowiednią hierarchię klas. Interpreter w istocie jest odmianą wzorca Kompozyt. Różnica polega na tym, iż w odróżnieniu od kompozytu reprezentuje pewne reguły gramatyczne. Na rysunku 5.34 pokazano reprezentację interpretera. Klasa abstrakcyjna (często implementowana w postaci klasy polimorficznej) `AbstractExpression` definiuje operację (lub grupę operacji) służących do interpretowania określonych reguł gramatycznych.

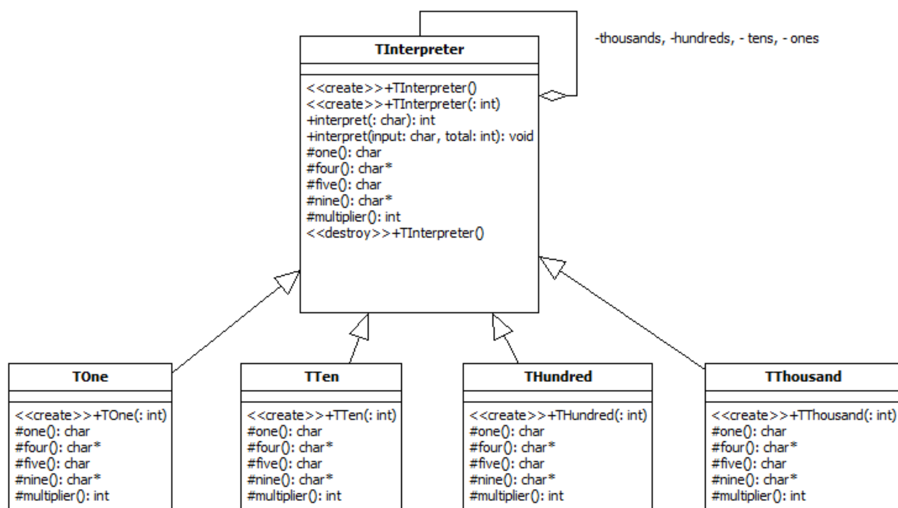


Rysunek 5.34. Podstawowa reprezentacja wzorca interpretera

Klasa `Context` przechowuje informacje globalne na temat wartości interpretowanych zmiennych. Klasa `TerminalExpression` realizuje abstrakcję `AbstractExpression` poprzez implementuję operacji dla symbolu terminalnego (symbolu, który może wystąpić w generowanym słowie języka) występującego w przetwarzanych danych wewnętrznych. Klasa `NonTerminalExpression` zawiera operację interpretującą regułę gramatyczną dla symbolu nieterminalnego (symbolu pomocniczego, który nie może wystąpić

w generowanym słowie języka). Zadaniem klienta jest stworzenie abstrakcyjnego drzewa składni przedstawiającego zadanie do wykonania oraz uruchomienie operacji interpretacji.

Na rysunku 5.35 pokazano diagram klas przedstawiający uproszczoną odmianę wzorca interpretera opartą na klasie polimorficznej `TInterpreter`. Na listingu 5.18 zaprezentowano implementację diagramu 5.35. Program służy do interpretacji cyfr rzymskich.



Rysunek 5.35. Odmiana interpretera z klasą polimorficzną

Listing 5.18. Implementacja diagramu 5.35

```
#include <iostream>
using namespace std;
class TThousand; class THundred;
class TTen; class TOne;
class TInterpreter {
  public:
    // konstruktor dla klienta
    TInterpreter();
    // konstruktor dla klas pochodnych
    TInterpreter(int) {}
    // interpret() dla klienta
    int interpret(char*);
    virtual void interpret(char* input, int& total) {
       // zmienne do uzytku wewnetrznego
       int index;
       index = 0;
       if(!strncmp(input, nine(), 2)) {
```

```
          total += 9 * multiplier();
          index += 2;
        }
        else
          if(!strncmp(input, four(), 2)) {
            total += 4 * multiplier();
            index += 2;
          }
          else {
            if(input[0] == five()) {
              total += 5 * multiplier();
              index = 1;
            }
            else
              index = 0;
            for(int end = index + 3 ; index < end; index++)
              if(input[index] == one())
                total += 1 * multiplier();
                else
                  break;
          }
        strcpy(input, &(input[index]));
     }
     virtual ~TInterpreter(){};
  protected:
   // funkcje nie mogą być czysto wirtualne,
   //ponieważ klient tworzy egzemplarze klasy
     virtual char one() {}
     virtual char* four() {}
     virtual char five() {}
     virtual char* nine() {}
     virtual int multiplier() {}
  private:
     TInterpreter* thousands;
     TInterpreter* hundreds;
     TInterpreter* tens;
     TInterpreter* ones;
};
//-----------------------------------------------------------
class TThousand : public TInterpreter {
  public:
     TThousand(int): TInterpreter(1) {}
  protected:
     char one() {return 'M';}
     char* four() {return "";}
```

```
    char five() {return '\0';}
    char* nine() {return "";}
    int multiplier() {return 1000;}
};
//-----------------------------------------------------------
class THundred: public TInterpreter {
  public:
    THundred(int): TInterpreter(1) {}
  protected:
    char one() {return 'C';}
    char* four() {return "CD";}
    char five() {return 'D';}
    char* nine() {return "CM";}
    int multiplier() {return 100;}
};
//-----------------------------------------------------------
class TTen: public TInterpreter {
  public:
    TTen(int): TInterpreter(1) {}
  protected:
    char one() {return 'X';}
    char* four() {return "XL";}
    char five() {return 'L';}
    char* nine() {return "XC";}
    int multiplier() {return 10;}
};
//-----------------------------------------------------------
class TOne: public TInterpreter {
  public:
    TOne(int): TInterpreter(1) {}
  protected:
    char one() {return 'I';}
    char* four() {return "IV";}
    char five() {return 'V';}
    char* nine() {return "IX";}
    int multiplier() {return 1;}
};
//-----------------------------------------------------------
TInterpreter::TInterpreter() {
  thousands = new TThousand(1);
  hundreds = new THundred(1);
  tens = new TTen(1);
  ones = new TOne(1);
}
//-----------------------------------------------------------
```

```
int TInterpreter::interpret(char* input) {
  int total;
  total = 0;
  thousands->interpret(input, total);
  hundreds->interpret(input, total);
  tens->interpret(input, total);
  ones->interpret(input, total);
  // jezeli na wejsciu pojawi sie niewlasciwa dana
  if(strcmp(input, ""))
    return 0;
  return total;
}
//---------------------------------------------------------
int main() {
  TInterpreter interpreter;
  char input[20];
  cout << "Wprowadz cyfre rzymska, q - wyjscie.\n";
  while (cin >> input) {
    if(input[0] == 'q') break;
    cout << "arabski odpowiednik wynosi: "
         << interpreter.interpret(input) << endl;
    cout << "Wprowadz cyfre rzymska:\n ";
  }
  cin.get();
  return 0;
}
```

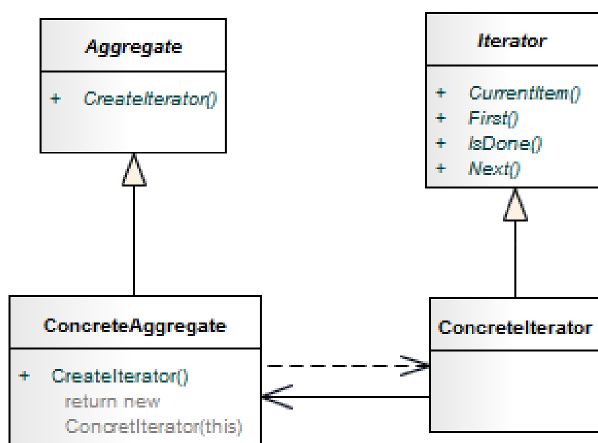
Wynik działania programu:

```
Wprowadz cyfre rzymska, q - wyjscie.
XLI
arabski odpowiednik wynosi: 41
Wprowadz cyfre rzymska:
 LII
arabski odpowiednik wynosi: 52
Wprowadz cyfre rzymska:
```

5.22. Iterator

W standardowym ujęciu, wiele kolekcji obiektowych, takich jak listy, mapy, kolejki, stosy, itp. wymaga specyficznej obsługi i stosowania różnorodnych metod dostępu do ich składowych elementów. Wzorzec Iterator podaje przepis na zunifikowanie uzyskiwania dostępu do elementów kolekcji pozwalając zaniedbać różnice w ich implementacji. W podstawowej reprezentacji wzorzec składa się z dwóch klas abstrakcyjnych (często implementowanych w postaci interfejsów): `Aggregate` i `Iterator`, oraz dwóch klas dziedziczących `ConcreteAggregate` i `ConcreteIterator`. Klasa `Iterator` deklaruje operacje `First()`, `Next()`, `CurrentItem()` oraz `IsDone()` umożliwiające

uzyskiwanie sekwencyjnego dostępu do kolekcji obiektów. W niektórych implementacjach składowe iteratora pozwalają również na dokonywanie modyfikacji kolekcji, np. dodawanie i usuwanie jej elementów. Klasa `Aggregate` deklaruje iterator `CtrateIterator()`, który jest implementowany w klasie dziedziczącej `ConcreteAggregate`, tak jak pokazano to na rysunku 5.36. Wszystkie kolekcje posiadają metodę tworzącą iterator i są implementacją abstrakcji `Aggregate`. `Iterator`, podobnie jak `Aggregate`, jest jedynie specyfikacją interfejsu, jaki każdy iterator powinien posiadać. Poprzez wywołanie metody `CreateIterator()` dla każdego elementu w kolekcji, klient otrzymuje klasę implementującą interfejs `Iterator`. Dzięki temu klient nie zna konkretnej klasy implementacyjnej, a jedynie interfejs, do którego powinien się odwołać.



Rysunek 5.36. Podstawowa reprezentacja wzorca Iterator

Stosowalność omawianego wzorca w C++ jest mało efektywna. Już od początku lat 90. ubiegłego wieku w skład języka C++ wchodzi biblioteka standardowa STL, której podstawowymi elementami są: kontenery, obiekty funkcyjne, adaptery, alokatory oraz algorytmy [32]. Kontenery STL (będąc uogólnieniem pojęcia tablicy w C++) są wzorcami klas, które po dowiązaniu aktualnych parametrów umożliwiają składowanie innych obiektów (umożliwiają tworzenie kolekcji obiektów). Iteratory, uogólniając pojęcie wskaźnika, wskazują na składowane w kontenerach obiekty. Pozwalają tym samym na odseparowanie kontenerów od algorytmów. Wszystkie kontenery udostępniają funkcje `begin()` i `end()`, umożliwiające uzyskanie iteratorów wskazujących na pierwszy i ostatni element przechowywany w kontenerze. Iteratory STL dzielą się na pięć podstawowych typów [33]:

- Iteratory wejściowe (ang. *input iterator*) — pozwalające pobrać wskazywany obiekt oraz dokonać jego inkrementacji, tak aby uzyskać wskaźnik do kolejnego elementu przechowywanego w kontenerze. Przykładem iteratora

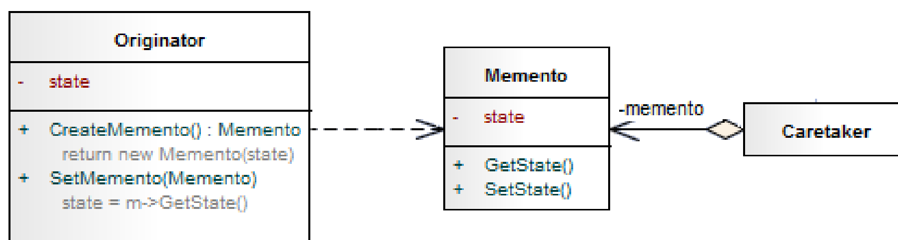
wejściowego jest istream_iterator.

- Iteratory wyjściowe (ang. *output iterator*) — pozwalające na zapis wartości we wskazywanym obiekcie, co jednak nie jest tożsame z uzyskiwaniem dostępu do obiektu. Przykładem iteratora wyjściowego jest ostream_iterator.
- Iteratory postępujące (ang. *forward iterator*) — umożliwiające wykonanie przejścia do następnego elementu składowanego w kontenerze, nie unieważniając poprzednio wskazywanej wartości. Przykładem iteratora postępującego jest slist<T>::iterator.
- Iteratory dwukierunkowe (ang. *bidirectional iterator*) — pozwalające zarówno na inkrementację, jak i dekrementację elementów składowanych w kontenerze. Przykładem iteratora dwukierunkowego jest list<T>::iterator.
- Iteratory dostępu swobodnego (ang. *random access iterator*) — umożliwiające inkrementację, dekrementację oraz poruszanie się zarówno do przodu, jak i do tyłu po elementach składowanych w kontenerze. Dla tego typu iteratorów zdefiniowane są ponadto operacje dodawania i odejmowania wartości do iteratora i od niego, jak również operacje odejmowania iteratorów. Przykładem iteratora o dostępie swobodnym jest vector<T>::iterator.

5.23. Memento

Memento jest wzorcem, którego wykorzystanie umożliwia zapamiętanie stanu obiektu w celu jego późniejszego wykorzystania. Niektóre programy wykorzystują wzorzec Memento w celu implementacji funkcji odtwarzającej ostanie działanie (ang. *undo*). Pokazany na rysunku 5.37 wzorzec składa się z trzech głównych klas: Originator, Memento i Caretaker. Obiekt klasy Originator posiada możliwość utworzenia obiektu klasy Memento (tzw. obiektu migawki stanu) ze swoim aktualnym stanem i odtworzenia tego stanu na podstawie stanu obiektu klasy Memento. W tym celu w klasie Originator deklarowany jest prywatny atrybut state przechowujący wartość opisującą aktualny stan obiektu oraz publiczne operacje createMemento() i setMemento() służące odpowiednio do utworzenia migawki stanu oraz jej ewentualnego późniejszego odczytania w celu przywrócenia wcześniejszej wartości. Klasa Memento definiuje dwie publiczne metody getState() i setState() służące odpowiednio do odczytania i zapisania wartości prywatnego atrybutu state. Obiekt klasy Memento przechowuje zapisany w atrybucie state stan obiektu Originator uniemożliwiając obiektowi klasy Caretaker uzyskanie bezpośredniego dostępu do wartości tego atrybutu. Głównymi funkcjonalnościami obiektu klasy Caretaker są: możliwość przechowywania kolekcji obiektów klasy Memento oraz udostępnianie metod do zapisu i odczytu stanu obiektu, nie mając jednak możliwości modyfikacji stanu

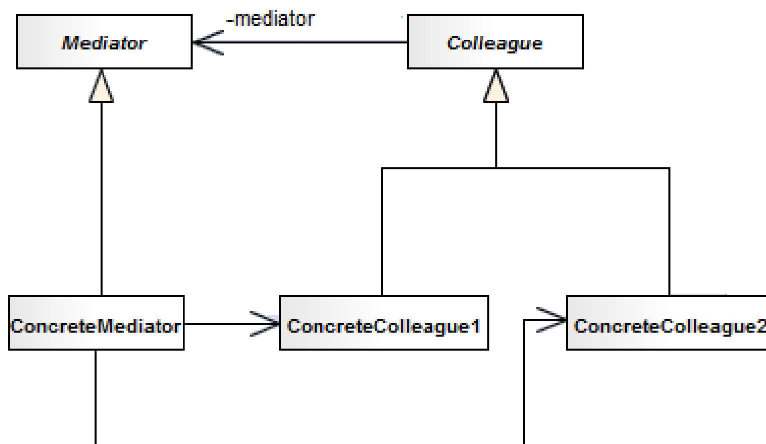
przechowywanego obiektu.



Rysunek 5. 37. Podstawowa reprezentacja wzorca Memento

5.24. Mediator

Często spotyka się sytuacje, w których obiekty realizujące wspólny interfejs komunikują się celu wykonania określonego zadania. Najprostszym i najmniej wydajnym rozwiązaniem tego problemu jest powiązanie wszystkich obiektów w topologii grafu pełnego. Zaprezentowany na rysunku 5.38 wzorzec Mediator proponuje topologię gwiazdy, której centralnym elementem jest obiekt klasy `ConcreteMediator` realizującej interfejs `Mediator`.



Rysunek 5.38. Podstawowa reprezentacja wzorca mediator

Interfejs `Mediator` powiązany jest z interfejsem `Colleague`. `Colleague` definiuje wspólny interfejs dla obiektów klas `ConcreteColleague`. Komunikacja pomiędzy obiektami realizującymi interfejs `Colleague` wymaga pośrednictwa obiektu realizującego interfejs `Mediator`, który to obiekt potrafi przekazać komunikat do właściwego odbiorcy. Mediator definiuje operacje służące do dołączania i odłączania obiektów realizujących interfejs `Colleague`.

Ponadto jego zadaniem jest implementacja mechanizmów komunikacji polegających na podejmowanie decyzji który z obiektów klas `ConcreteColleague` powinien wykonać określone żądanie. Obiekty klas `ConcreteColleague` nie są obciążone zadaniem komunikacji z pozostałymi obiektami. Ich wiedza jest ograniczona do znajomości klasy `Mediator`. Także dołączenie i odłączenie obiektów `ConcreteColleague` wymaga jedynie powiadomienia konkretnego mediatora, a nie wszystkich pozostałych obiektów.

Podsumowanie

Wzorce projektowe programowania obiektowego są modelami rozwiązań bardzo wielu zagadnień programistycznych wykorzystujących paradygmat obiektowości. Praktyczne zastosowanie ich w trakcie pracy nad różnego rodzaju projektami informatycznymi znacznie zwiększa efektywność zarówno w fazie projektowania jak i na etapie implementacji. Sprawne korzystanie ze wzorców projektowych programowania obiektowego wiąże się jednak z koniecznością praktycznego poznania języka oraz metod modelowania obiektowego. Obecny rozdział miał za zadanie zapoznać Czytelnika z podstawowymi mechanizmami posługiwania się wzorcami projektowymi. Zakres przedstawionego materiału z oczywistych względów nie wyczerpuje całkowicie obszernej tematyki związanej ze wzorcami projektowymi programowania obiektowego. Osobom zainteresowanym poszerzeniem przedstawionych tu wiadomości należy polecić opracowania specjalnie dedykowane temu tematowi [31,34,35].

W niniejszym rozdziale omówiono jedynie wzorce projektowe programowania obiektowego. Pozostałe kategorie wzorców powinny być się tematem wykładów z takich przedmiotów jak: Inżynieria oprogramowania, Programowanie komponentowe, Programowanie współbieżne oraz wykładów traktujących o podstawach programowania w wybranych językach (C++, Java, C#)

ROZDZIAŁ 6

BIBLIOTEKI ŁĄCZONE DYNAMICZNIE

6.1. Biblioteki współdzielone 200
6.2. API Windows...201
6.2.1. Funkcja DllEntryPoint() 205
6.3. Klasy jako elementy bibliotek dołączanych dynamicznie 208
6.4. GNU/Linux 212
Podsumowanie 216

6.1. Biblioteki współdzielone

Większość programów tworzonych w popularnych systemach operacyjnych, takich jak Windows czy GNU/Linux posiada tzw. nierozwiązane odniesienia do funkcji — często nazywa się je importowanymi wywołaniami bibliotek. Poszczególne biblioteki dołączane dynamicznie zawierają programy potrzebne do wykonania wywoływanej funkcji i w momencie instalacji z reguły stają się częścią API systemu operacyjnego. Biblioteki współdzielone (ang. *shared library*) przechowują implementacje różnych funkcji (podprogramów). Funkcje i zasoby zawarte w bibliotece współdzielonej mogą być wykorzystane bezpośrednio lub pośrednio (za pośrednictwem innej biblioteki współdzielonej) przez dowolny plik wykonywalny. Sama biblioteka współdzielona nie jest programem przeznaczonym do samodzielnego wykonania. Funkcje biblioteki współdzielonej mogą być jednocześnie importowane przez wiele działających programów, dzięki czemu zarządzanie zasobami pamięci operacyjnej przebiega w sposób bardziej wydajny. Podstawowym aspektem projektowania aplikacji wykorzystującej zewnętrzne biblioteki współdzielone jest sposób łączenia programu głównego z biblioteką. Połączenie takie może mieć charakter statyczny lub dynamiczny [22].

- *Charakter statyczny*. Biblioteki łączone są z programem wykonywalnym w czasie jego konsolidowania. Aby uruchomić program, należy mieć dostęp do wszystkich skompilowanych bibliotek, od których program jest zależny pośrednio lub bezpośrednio. W trakcie uruchamiania wszystkie składniki są ładowane i łączone. W tym stanie program pozostaje aż do zakończenia działania.
- *Charakter dynamiczny*. W celu uruchomienia programu nie jest wymagany dostęp do wszystkich skompilowanych bibliotek, od których program jest zależny pośrednio lub bezpośrednio. Dołączanie biblioteki (wczytanie jej do pamięci operacyjnej) następuje „na żądanie” użytkownika. Również w ten sposób bibliotekę można odłączyć. Łączenie dynamiczne zapewnia szybszą kompilację i uruchamianie programu oraz bardziej racjonalne wykorzystanie zasobów pamięci.

W systemach operacyjnych Windows biblioteki łączone dynamicznie (ang. *dynamic-link library*) mają rozszerzenia *.dll*, *.ocx* (gdy biblioteka jest komponentem ActiveX) lub *.cpl* (gdy biblioteka jest rozszerzeniem Panelu Sterowania) i mogą być wykorzystane w programach napisanych w różnych językach programowania wykorzystywanych na platformie Windows, m.in. w Visual Basicu, C, C++ ,C#, Object Pascalu (Delphi) czy asemblerze. Biblioteki łączone statycznie mają rozszerzenie *.lib* (ang. *library*). W systemach operacyjnych GNU/Linux, Solaris, System V, BSD biblioteki łączone dynamicznie mają rozszerzenie *.so* (ang. *shared object*), zaś biblioteki łączone statycznie mają rozszerzenie *.a* (ang. *archive*, *static library*).

6.2. API Windows

Zawarte w jądrze systemu Windows pliki *.lib* i *.dll* są wykonywalnymi modułami zawierającymi definicje eksportowanych zmiennych, obiektów i funkcji, a w szczególności funkcji sterowników urządzeń. Biblioteki dynamicznie dołączane w czasie wykonywania programu (ang. *run-time dynamic linking*) wymagają wywołań funkcji `LoadLibrary()`, `GetProcAddress()` oraz `FreeLibrary()`.

Funkcja `LoadLibrary()` odwzorowuje wskazany moduł wykonawczy (plik *.dll* lub *.exe*) w przestrzeń adresową macierzystego procesu.

```
HINSTANCE LoadLibrary(IN LPCTSTR lpLibFileName);
```

Parametr `lpLibFileName` jest wskaźnikiem do ciągu znaków zakończonych zerowym ogranicznikiem, zawierającym nazwę modułu wykonawczego (plik *.dll* lub *.exe*). Podana nazwa dotyczy pliku modułu i nie jest związana z nazwą własną przechowywaną w module bibliotecznym. Funkcja zgłosi błąd w postaci wskaźnika pustego, jeśli podana nazwa zawiera ścieżkę dostępu do pliku, w którym żądany moduł nie występuje. Jeżeli nie podano ścieżki dostępu do pliku i pominięto rozszerzenie nazwy pliku, to domyślnie dołączanym rozszerzeniem jest *.dll*. W przypadku powodzenia wartością zwracaną przez funkcję jest identyfikator modułu `HINSTANCE` (ang. *handle of instance*).

Funkcja `LoadLibrary()` używana jest głównie w celu dynamicznego odwzorowywania modułów DLL i pobierania ich identyfikatorów, które z kolei używane są przez funkcję:

```
FARPROC GetProcAddress(IN HMODULE hModule,
                       IN LPCSTR lpProcName);
```

do pobierania adresów funkcji składowych bibliotek DLL. Funkcji `LoadLibrary()` można użyć do odwzorowywania innych modułów wykonawczych. W przypadku, gdy funkcja odnosi się do pliku *.exe*, zwrócony identyfikator można wykorzystać jako argument funkcji Windows API `FindResource()` oraz `LoadResource()`. Warto jednak zdawać sobie sprawę z faktu, iż identyfikatory modułów nie są zmiennymi globalnymi. Oznacza to, że wywołanie funkcji `LoadLibrary()` w jednym procesie nie może być wykorzystane do określania identyfikatora, używanego w procesach potomnych.

Do zwalniania identyfikatora przydzielonego za pomocą funkcji `LoadLibrary()` służy funkcja `FreeLibrary()`, której argumentem jest identyfikator zwrócony przez `LoadLibrary()`.

Na listingu 6.1 zaprezentowano przykład programu konsolowego C++, który dynamicznie dołącza bibliotekę *hid.dll* zdefiniowaną w jądrze systemu Windows i udostępniającą funkcję `HidD_GetHidGuid()`, za pomocą której można

dokonać enumeracji urządzeń aktualnie podłączonych do magistrali USB [24].

Listing 6.1. Przykład zaprogramowania wstępnej enumeracji urządzeń USB na podstawie identyfikatora GUID klasy urządzeń

```
#include <windows>
#pragma option push -a1
   #include <setupapi>
#pragma option pop
#include <assert>
#include <iostream>

using namespace std;

void displayError(const char* msg){
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//-----------------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete x;
  x = NULL;
}
//-----------------------------------------------------------
GUID classGuid;
HMODULE hHidLib;
DWORD /*unsigned long*/ memberIndex = 0;
DWORD deviceInterfaceDetailDataSize;
DWORD requiredSize;

HDEVINFO deviceInfoSet;
SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA
           deviceInterfaceDetailData = NULL;
//-----------------------------------------------------------
int main(){

  void (__stdcall *HidD_GetHidGuid)(OUT LPGUID HidGuid);

  hHidLib = LoadLibrary("C:\\Windows\\system32\\HID.DLL");
  if (!hHidLib)
    displayError("Błąd dołączenia biblioteki HID.DLL.");
```

```
(FARPROC&) HidD_GetHidGuid = GetProcAddress(hHidLib,
                             "HidD_GetHidGuid");
 if (!HidD_GetHidGuid){
    FreeLibrary(hHidLib);
    displayError("Nie znaleziono identyfikatora GUID");
 }

 HidD_GetHidGuid(&classGuid);

 deviceInfoSet = SetupDiGetClassDevs(&classGuid, NULL,
         NULL, DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
 if (deviceInfoSet == INVALID_HANDLE_VALUE)
    displayError("Nie zidentyfikowano podłączonych
                 urządzeń.\n");

 deviceInterfaceData.cbSize =
                     sizeof(SP_DEVICE_INTERFACE_DATA);

 while(SetupDiEnumDeviceInterfaces(deviceInfoSet, NULL,
      &classGuid, memberIndex, &deviceInterfaceData)){
     memberIndex++; //inkrementacja numeru interfejsu

     SetupDiGetDeviceInterfaceDetail(deviceInfoSet,
           &deviceInterfaceData, NULL, 0,
           &deviceInterfaceDetailDataSize, NULL);

     deviceInterfaceDetailData =
            (PSP_DEVICE_INTERFACE_DETAIL_DATA) new
            DWORD[deviceInterfaceDetailDataSize];

     deviceInterfaceDetailData->
      cbSize=sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

     if (!SetupDiGetDeviceInterfaceDetail(deviceInfoSet,
         &deviceInterfaceData, deviceInterfaceDetailData,
         deviceInterfaceDetailDataSize, &requiredSize,
                                            NULL)){
         releaseMemory(deviceInterfaceDetailData);
     }
     cout << deviceInterfaceDetailData->DevicePath
          << endl;
     releaseMemory(deviceInterfaceDetailData);
  };//koniec while
```

```
   SetupDiDestroyDeviceInfoList(deviceInfoSet);
   FreeLibrary(hHidLib);
   cin.get();
   return 0;
}
```

Biblioteki łączone statycznie wymagają dołączenia ich w trakcie konsolidowania programu. Operację tę można wykonać albo poprzez odpowiednie opcje kompilatora, albo (co jest dużo prostsze) wykorzystując dyrektywę:

```
#pragma comment(lib, "nazwaBiblioteki.lib")
```

Na listingu 6.2 pokazano przykładową implementacją funkcji API Windows wykorzystywanych do programowej kontroli urządzeń Bluetooth. Definicje tego typu funkcji znajdują się w modułach *Ws2bth.h*, *Bthsdpdef.h* oraz *BluetoothAPIs.h*. Moduł *Ws2bth.h* powinien być włączany do kodu po obowiązkowo używanym module *Winsock2.h*. Dodatkowo program główny powinien być statycznie łączony z biblioteką *Bthprops.lib*. Biblioteka ta jest standardowo dostępna w zasobach systemów Windows XP z Service Pack 3, Vista oraz 7. Program używa wygodnych procedur za pomocą których można szybko zidentyfikować urządzenie z włączoną opcją Bluetooth oraz sparować je z komputerem za pomocą wybranej opcji oraz kodu parowania.

Listing 6.2. Programowa obsługa wykrywania i parowania urządzeń bezprzewodowych Bluetooth

```
#include <iostream>
#pragma option push -a1
  #include <winsock2>
  #include "Ws2bth.h"
  #include "BluetoothAPIs.h"
#pragma option pop

#pragma comment(lib, "Bthprops.lib")

using namespace std;

BOOL __stdcall devInfoCallback(LPVOID pvParam,
                               PBLUETOOTH_DEVICE_INFO
pDevice)
{
  //...
  return TRUE;
}

//----------------------------------------------------------
```

```
int main()
{
   BLUETOOTH_SELECT_DEVICE_PARAMS bthsdp={sizeof(bthsdp)};

   bthsdp.hwndParent = NULL;
   bthsdp.fShowUnknown = TRUE;
   bthsdp.fAddNewDeviceWizard = TRUE;
   bthsdp.fSkipServicesPage = FALSE;
   bthsdp.fShowRemembered = TRUE;
   bthsdp.fShowAuthenticated = TRUE;
   bthsdp.pfnDeviceCallback = devInfoCallback;
   bthsdp.cNumDevices = 0;
   //...
   BOOL bthSelectDevices = BluetoothSelectDevices(&bthsdp);
   if (bthSelectDevices) {
      BLUETOOTH_DEVICE_INFO * pbtdi = bthsdp.pDevices;
      pbtdi->dwSize = sizeof(pbtdi);
      for (ULONG cDevice=0; cDevice<bthsdp.cNumDevices;
           cDevice++){
         if (pbtdi->fAuthenticated || pbtdi->fRemembered ) {
            wprintf(L"Device: %s\n",pbtdi->szName);
            wprintf(L"Class of Device: %d\n",pbtdi->
                    ulClassofDevice);
            //...
         }
         pbtdi=(BLUETOOTH_DEVICE_INFO *)
               ((LPBYTE)pbtdi+pbtdi->dwSize);
      }
      BluetoothSelectDevicesFree(&bthsdp);
   }//koniec if
   cin.get();
   return 0;
}
```

6.2.1. Funkcja DllEntryPoint()

Kod źródłowy C++ kompilowany jest do postaci biblioteki *.dll* w systemach operacyjnych Windows wygląda podobnie jak kod pliku źródłowego standardowego programu. Różnica polega na tym, iż zamiast funkcji `WinMain()` typu `WINAPI` biblioteka *.dll* używa innej funkcji `DllEntryPoint()`. Funkcja ta jest wykonywana przy każdym załadowaniu biblioteki do aplikacji. Kiedy ładowana lub usuwana jest z pamięci biblioteka *.dll* lub uruchamiany (zatrzymywany) jest wątek, Windows wywołuje tę funkcję, przekazując jej odpowiednio argument `reason`. Funkcja `DllEntryPoint()` jest wywoływana, kiedy biblioteka *.dll* jest

po raz pierwszy ładowana do pamięci w trakcie wykonania odnośnego procesu [22]. Parametr `reason` zawiera jedną z następujących stałych:

`DLL_PROCESS_DETACH = 0`

Proces kończy swoją pracę lub usuwa z pamięci bibliotekę *.dll*.

`DLL_PROCESS_ATTACH = 1`

Kod inicjujący DLL jest wywoływany i specyfikuje `DLL_PROCESS_ATTACH` jako parametr.

`DLL_THREAD_ATTACH = 2`

Proces, który załadował bibliotekę *.dll*, utworzył nowy wątek.

`DLL_THREAD_DETACH = 3`

Wątek został zakończony.

Programista nie musi pisać oddzielnego kodu funkcji przeznaczonej do obsługi zdarzenia `DLL_PROCESS_ATTACH`. Często niezbędne jest jednak stworzenie funkcji obsługi dla odłączenia procesu lub odłączenia (dołączania) zdarzeń wątków. Określone zdarzenie przypisuje się do zmiennej globalnej:

```
extern PACKAGE void *DllProc;
```

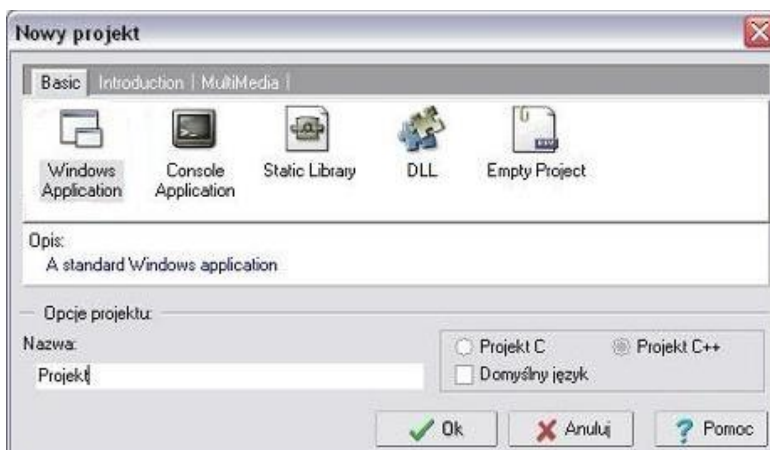
Kompilator C++ odpowiada za zarejestrowanie funkcji w Windows. Funkcja ta jest następnie wywoływana przez system operacyjny w chwili odłączania procesu, ewentualnie dołączania (odłączania) wątku (jeżeli, oczywiście, takowy istnieje). Listing 6.3 przedstawia przykład wykorzystania funkcji `DllEntryPoint()`.

Listing 6.3. Przykład wykorzystania funkcji `DllEntryPoint()`

```
#include <windows.h>

void LibExit(int reason)
{
  if (reason == DLL_PROCESS_DETACH) {
    // kod powrotny z biblioteki
  }
};
//------------------------------------------------------
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long
```

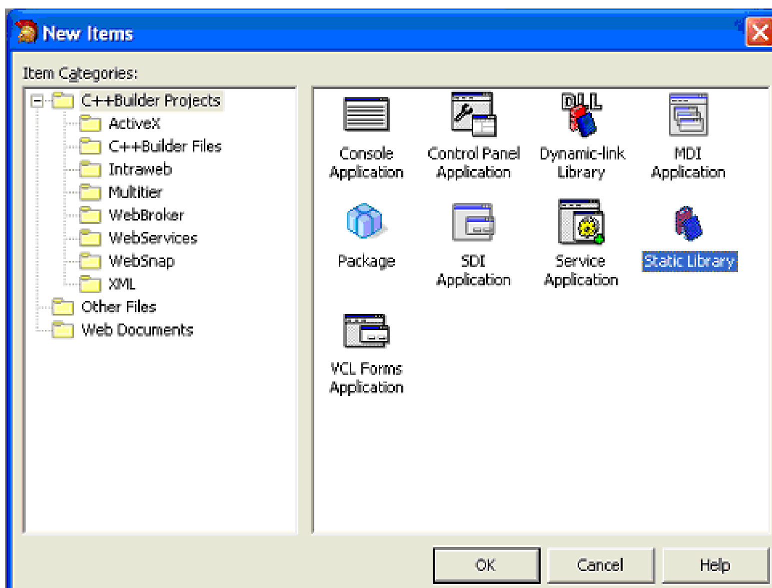
```
                                reason, void* lpReserved)
{
 //hinst = LoadLibrary("nazwa");
  DllProc = LibExit;
  return 1;
}
```

Stosowanie bibliotek *.dll* wymaga zachowania ostrożności w trakcie wykorzystywania wskaźników oraz pamięci alokowanej dynamicznie. Pamięć zajęta przez DLL jest zwalniana w chwili usunięcia z niej biblioteki. Aplikacja może jednak zachować wskaźniki do takiej pamięci. Brak ostrożności, polegający na użyciu wskaźnika do zwolnionego obszaru, niezmiennie prowadzi do powstania błędu naruszenia pamięci *Access Violation*. W sytuacji takiej najprostszym rozwiązaniem jest użycie biblioteki *Memmgr.lib* menedżera pamięci jako pierwszego modułu w aplikacji oraz każdej bibliotece ładowanej przez aplikację. Moduł *Memmgr.lib* przekierowuje wszystkie żądania pamięci do odpowiednich modułów *.dll*, które pozostają załadowane tak długo, jak długo wykonuje się aplikacja. Rozwiązanie takie pozwala na swobodne ładowanie i usuwanie bibliotek z pamięci bez obawy o puste wskaźniki. System umożliwia też natychmiastowe odłączenie modułu *.dll* od procesu, który go używał, bez konieczności anulowania odwzorowania modułu w przestrzeń adresową procesu. Można tego dokonać przy użyciu funkcji `DllEntryPoint()` z ustawionym znacznikiem `DLL_PROCESS_DETACH`, co daje możliwość natychmiastowego zwolnienia zasobów zajmowanych przez bieżący proces. Wartość `DLL_PROCESS_DETACH` jest najbardziej użyteczna, gdy `DLLProc` znajduje się w pliku projektu biblioteki. Po wykonaniu funkcji `DllEntryPoint()` moduł *.dll* zostaje usunięty z przestrzeni adresowej procesu. W funkcji `DllEntryPoint()` parametr `hinst` stanowi identyfikator biblioteki *.dll*. Wartość tego parametru jest jej adresem bazowym.



Rysunek 6.1. Opcje projektu Dev C++

Sposób rozpoczęcia pracy z biblioteką *.dll* w wybranym kompilatorze C++ zależy od wyboru opcji projektu. W większości kompilatorów C++ działających w systemach operacyjnych Windows, w celu rozpoczęcia konstruowania biblioteki *.dll*, w opcjach nowego projektu należy wybrać odpowiednią ikonę: *DLL* – dla bibliotek łączonych dynamicznie lub *Static Library* – dla bibliotek łączonych statycznie, tak jak pokazano to na rysunkach 6.1 i 6.2.



Rysunek 6.2. Opcje projektu C++ Builder

Po stworzeniu kodu biblioteki, moduł należy skompilować do postaci *.dll* (dla bibliotek łączonych dynamicznie) lub *.lib* (dla bibliotek łączonych statycznie) korzystając również z odpowiednich opcji wykorzystywanego kompilatora.

6.3. Klasy jako elementy bibliotek dołączanych dynamicznie

Obiekty mogą być elementami eksportowanymi z bibliotek dołączanych dynamicznie. Sytuację taką rozpatrzmy na przykładzie kodu pokazanego w Rozdziale 5. (patrz listing 5.12). Przykład ten zmodyfikujemy w ten sposób, aby definicja klasy bazowej znajdowała się w oddzielnym pliku nagłówkowym. Biblioteka *.dll* zawiera jedynie definicje klas pochodnych (klas – liści) oraz ciała funkcji eksportowych. Głównym zadaniem funkcji eksportowych jest eksportowanie obiektów stworzonych na bazie odpowiednich klas. W momencie dynamicznego połączenia biblioteki z programem głównym można na podstawie eksportowanych obiektów odzyskać wszystkie operacje w nich zdefiniowane. Ciąg czynności niezbędnych do wykorzystania biblioteki *.dll* w charakterze bytu

eksportującego obiekty pokazano na listingach 6.4 – 6.6.

Listing 6.4. Kod źródłowy pliku nagłówkowego *sort.h* zawierającego definicję bazowej klasy abstrakcyjnej

```
#ifndef __SORT_H__
#define __SORT_H__
class TSort {
  public:
    void shellSort(int *v, int n) {
      for (int g = n/2; g > 0; g /= 2)
        for (int i = g; i < n; i++)
          for (int j = i-g; j >= 0;j -= g)
             if (sort(v[j],v[j+g]))
               swap(v[j], v[j+g]);
    }
  private:
    virtual int sort(int, int) = 0;
    void swap(int& a, int& b) {int t = a; a = b; b = t; }
};
#endif
```

Listing 6.4. Kod źródłowy modułu *sort.cpp* biblioteki DLL

```
#include <windows.h>
#include "sort.h"
class TIncrease: public TSort {
  private:
    int sort(int a, int b) {return (a > b); }
};
//-----------------------------------------------------------
class TDecrease : public TSort {
  private:
    int sort(int a, int b) {return (a < b); }
};
//-----------------------------------------------------------
extern "C" void* __declspec(dllexport) shellSortDec() {
  return new TDecrease;
}
//----------------------------
extern "C" void* __declspec(dllexport) shellSortInc(){
  return new TIncrease;
}
```

Można zauważyć, iż ciała funkcji eksportowych w bibliotece piszemy w sposób identyczny, jak w standardowym programie. Niemniej jednak istnieją pewne różnice w deklaracjach tych funkcji. Każda biblioteka dołączana do

macierzystej aplikacji powinna w pewien sposób udostępniać jej swoje funkcje lub zmienne. Oznacza to, że funkcje takie należy odpowiednio eksportować. Do tego celu służy dyrektywa `__declspec(dllexport)`:

```
__declspec(dllexport) typ_powrotny
                      nazwa_funkcji(lista_parametrów);
```

Aby wskazać, że odpowiednia funkcja będzie udostępniana do innych, zewnętrznych modułów aplikacji, należy też użyć dyrektywy `extern "C"`. Oznacza to, iż funkcje będzie wywoływana zgodnie ze standardem stosowanym w C, czyli poprzez podanie jej nazwy (C nie umożliwia przedefiniowywania funkcji).

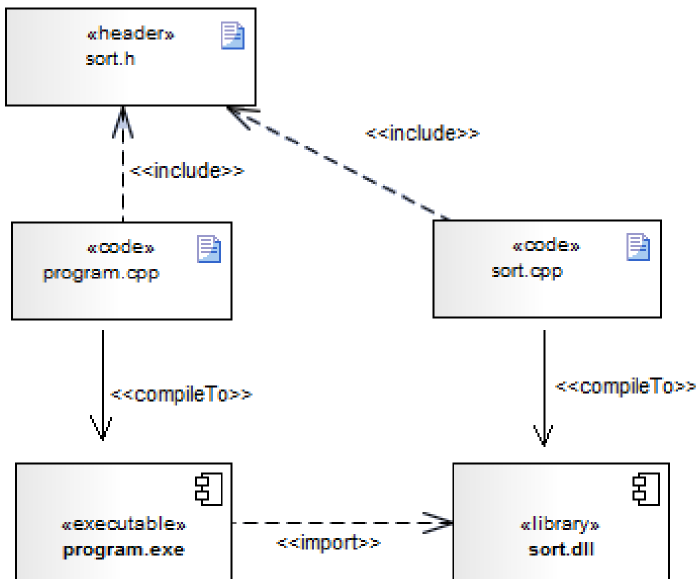
Aby biblioteka mogła być dołączana do innych aplikacji, musi zostać odpowiednio skompilowana. Używając właściwego dla danego kompilatora polecenia kompilacji pliku biblioteki spowodujemy automatyczne wygenerowanie pliku *sort.dll*. Ostatnim etapem jest stworzenie programu importującego funkcje eksportowane z biblioteki. Na listingu 6.6 zaprezentowano jego konstrukcję.

Listing 6.6. Kod źródłowy głównego modułu *program.cpp* importującego wybrane funkcje

```
#include <windows>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include "sort.h"
#define size 10
using namespace std;
//-----------------------------------------------------------
int main()
{
  srand((unsigned int)time((time_t *)NULL));
  int array[size];
  cout << "wektor wyjsciowy:\n";
  for (int i=0; i < size; i++) {
    array[i] = rand() % 100;
    cout << array[i] << ' ';
  }
  cout << endl;
  //domyślnie plik sort.dll znajduje się w bierzącym
  //katalogu
  HINSTANCE hDLL = LoadLibrary("sort.dll");
  TSort* myDLL;
  if(hDLL == NULL) {
     cout << "nie odnaleziono biblioteki" << endl;
     cin.get();
```

```
    return 0;
  }
  typedef TSort* (*funType)();
  funType shellSortInc = (funType)GetProcAddress(hDLL,
                                  "_shellSortInc");

  myDLL = shellSortInc();
  myDLL->shellSort(array, size);
  cout << "posortowany rosnąco:\n";
  for (int i=0; i < size; i++)
    cout << array[i] << ' ';
  cout << endl;

  funType shellSortDec = (funType)GetProcAddress(hDLL,
                                  "_shellSortDec");
  myDLL = shellSortDec();
  myDLL->shellSort(array, size);
  cout << "posortowany malejąco:\n";
  for (int i=0; i < size; i++)
     cout << array[i] << ' ';
  cout << endl;
  FreeLibrary(hDLL);
  cin.get();
  return 0;
}
```

W celu zaimportowania z biblioteki żądanych funkcji, w pierwszej kolejności należy skompilowaną bibliotekę dołączyć do programu głównego (innymi słowy należy – odwzorować identyfikator biblioteki w przestrzeń adresową macierzystego procesu). Czynność tę wykonujemy za pomocą funkcji API Windows `LoadLibrary()`. Po dołączeniu biblioteki, należy odczytać adresy funkcji, które powinny być z biblioteki importowane. Program główny nie może importować innych funkcji niż te, które biblioteka eksportuje (patrz listing 6.5). Żądane adresy odczytujemy za pomocą funkcji API `GetProcAddress()`. Po tej czynności można korzystać z funkcji eksportowych biblioteki w identyczny sposób, jak w przypadku funkcji udostępnianych przez standardowe obiekty. Od tej pory poprawne działanie głównego programu wykonywalnego *program.exe* będzie wymagało zaimportowania w jego przestrzeń adresową pliku biblioteki *sort.dll*. Na rysunku 6.3 pokazano uproszczony diagram wdrożenia dla poszczególnych elementów projektu, które zostały zilustrowane na listingach 6.4-6.6. W momencie, gdy biblioteka nie będzie już potrzebna – można ją odłączyć od głównego procesu za pomocą funkcji `FreeLibrary()`.



Rysunek 6.3. Diagram wdrożenia dla przykładów 6.4-6.6

6.4. GNU/Linux

Mechanizm konstruowania i implementacji współdzielonych bibliotek dołączanych dynamicznie w systemach GNU/Linux pod względem stosowanej metodologii nie odbiega zasadniczo od tego stosowanego w systemach operacyjnych Windows. Główna różnica polega na wykorzystaniu API właściwego GNU/Linux. Metody posługiwania się funkcją odwzorowującą bibliotekę współdzieloną w przestrzeni adresowej głównego procesu oraz funkcją pobierającą adresy funkcji eksportowych są tożsame z tymi stosowanymi w Windows. Poniżej przedstawiono podstawowe funkcje API GNU/Linux umożliwiające zarządzanie bibliotekami współdzielonymi.

Funkcja `dlopen()` otwiera i ładuje bibliotekę do przestrzeni adresowej procesu. Przyjmuje ona dwa argumenty. Poczynając od lewej są to: nazwa pliku biblioteki współdzielonej (razem ze ścieżką dostępu, jeżeli jest w niestandardowej lokalizacji) oraz jedna ze stałych symbolicznych określająca sposób uzyskiwania dostępu do funkcji eksportowanych z biblioteki:

- `RTLD_LAZY` (wartość 1) – określenie adresu funkcji eksportowej w chwili jej wywołania.
- `RTLD_NOW` (wartość 2) – określenie adresu funkcji eksportowej w trakcie dołączania biblioteki,
- `RTLD_GLOBAL` (wartość 100h) – funkcje eksportowe będą widoczne w programie głównym w ten sposób, jakby biblioteka była łączona statycznie (na stałe) z programem głównym.

Wartością powrotną funkcji `dlopen()` jest adres dołączonej biblioteki.

Dwuargumentowa funkcja `dlsym()` odczytuje adres eksportowanej z biblioteki funkcji. Pierwszym jej argumentem jest adres biblioteki, który zwraca funkcja `dlopen()`, drugim argumentem jest nazwa funkcji eksportowej pisana w formie łańcucha znaków (identycznie jak w przypadku funkcji `GetProcAddress()` API Windows). Funkcja `dlsym()` zwraca adres żądanej funkcji eksportowej.

Funkcja `dlclose()` odłącza bibliotekę, zwalniając jej identyfikator. Jedynym jej argumentem jest adres biblioteki otrzymany uprzednio za pomocą funkcji `dlopen()`.

Na listingach 6.7-6.9 zaprezentowano praktyczną metodę eksportowania obiektu pewnej klasy polimorficznej. Definicja klasy polimorficznej `TMyClass` znajduje się w pliku *myclass.h*. Funkcje zdefiniowane w klasie, tj. konstruktor i przykładowa funkcja składowa `doSomething()` oraz funkcje eksportowe biblioteki `createObject()` oraz `destroyObject()` są implementowane w pliku *myclass.cpp*. Plik ten jest następnie kompilowany do postaci biblioteki współdzielonej *myclass.so*. Implementacja programu głównego zawarta jest w pliku *classUser.cpp*. Na rysunku 6.4 pokazano zestaw artefaktów (plików implementacyjnych) oraz komponentów powstałych w wyniku kompilacji (transformacji) tychże artefaktów odpowiednio do postaci wykonywalnej i biblioteki współdzielonej.

Listing 6.7. Kod źródłowy pliku nagłówkowego *myclass.h* zawierającego definicję klasy polimorficznej

```
#ifndef __MYCLASS_H__
#define __MYCLASS_H__

class TMyClass
{
  public:
    TMyClass();
    virtual void doSomething();
  private:
    int x;
};
#endif
```

Listing 6.8. Kod źródłowy modułu *myclass.cpp* biblioteki współdzielonej

```
#include "myclass.h"
#include <iostream>
using namespace std;
//-----------------------------------------------------
extern "C" TMyClass* createObject()
{
```

```
   return new TMyClass;
}
//----------------------------------------------------------
extern "C" void destroyObject(TMyClass* object)
{
   delete object;
}
//----------------------------------------------------------
TMyClass::TMyClass()
{
   x = 10;
}
//----------------------------------------------------------
void TMyClass::doSomething()
{
   cout << x << endl;
}
```

Listing 6.9. Kod źródłowy głównego modułu *classUser.cpp* importującego wybrane operacje

```
#include <dlfcn.h>
#include <iostream>
#include "myclass.h"
using namespace std;
//----------------------------------------------------------
int main()
{
  void* handle = dlopen("myclass.so", RTLD_LAZY);

  TMyClass* (*create)();
  void (*destroy)(TMyClass*);

  create=(TMyClass* (*)())dlsym(handle,"createObject");
  destroy=(void (*) (TMyClass*))
           dlsym(handle,"destroyObject");
  TMyClass* myClass=(TMyClass*)create();
  myClass->doSomething();
  destroy( myClass );
}
```

Biblioteki współdzielone są pisane jako tzw. kod PIC – kod niezależny od pozycji (ang. *Position-Independent Code*). Oznacza to, iż skompilowany plik biblioteki współdzielonej nie posiada wiedzy, pod jakim adresem zostanie dołączony (załadowany) do macierzystego procesu. W trakcie kompilacji należy zatem poinformować kompilator o tym fakcie wykorzystując znacznik `-fPIC`.

Poniżej przedstawiono parametry kompilacji modułów z listingów 6.7-6.9 właściwych dla kompilatora g++ uruchomionego odpowiednio w systemach

GNU/Linux:

kompilacja biblioteki:

```
g++ -fPIC -shared myclass.cpp -o myclass.so
```

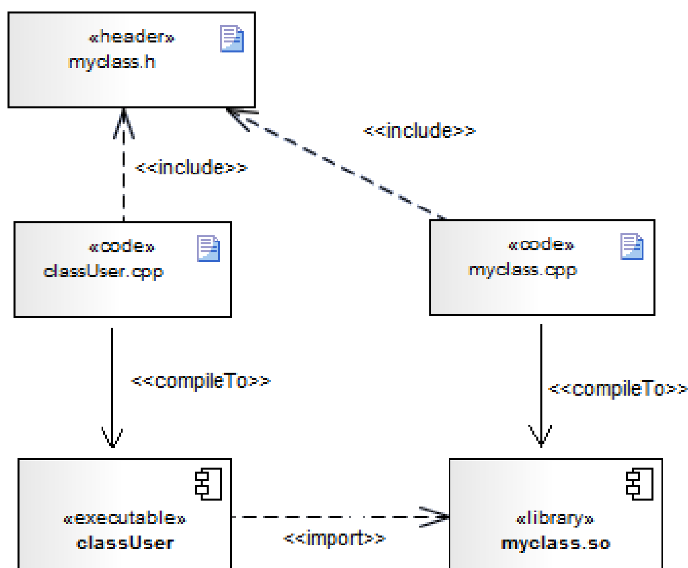
kompilacja programu głównego:

```
g++ classUser.cpp -ldl -o classUser
```

oraz Mac OS:

```
g++ -dynamiclib -flat_namespace myclass.cpp -o myclass.so
g++ classUser.cpp -o classUser
```

Rysunek 6.4 obrazuje uproszczony diagram wdrożenia dla artefaktów źródłowych z listingów 6.7-6.9 oraz komponentów powstałych w wyniku ich kompilacji do postaci wykonywalnej i współdzielonej.



Rysunek 6.4. Diagram wdrożenia dla przykładów 6.7-6.9

Podsumowanie

W rozdziale tym przedstawiono podstawowe zasady rządzące mechanizmem włączania do programu samodzielnie tworzonych bibliotek współdzielonych. Zostały przedstawione statyczne oraz dynamiczne sposoby łączenia biblioteki z programem. Omówiono też sposoby samodzielnego konstruowania bibliotek współdzielonych eksportujących obiekty klas oraz wyszukiwania adresów funkcji eksportowanych.

Rozdział 7

Obsługa wyjątków

7.1. Standardowa obsługa wyjątków 218
7.2. Klasy wyjątków 225
Podsumowanie 227

7.1. Standardowa obsługa wyjątków

Możliwość programowej obsługi wyjątków (tzw. sytuacji wyjątkowych) jest bardzo ważnym mechanizmem współczesnych języków obiektowych, pozwalającym w odpowiedni sposób kontrolować błędy powstające w trakcie działania programu. Wyjątki umożliwiają w sposób automatyczny tworzenie procedur obsługi błędów. We współczesnych kompilatorach, technika obsługi wyjątków została znacznie rozbudowana o mechanizmy, które postaramy się przedstawić w treści tego rozdziału.

Standardowa obsługa wyjątków w C++, Javie i C# opiera się na trzech słowach kluczowych: `try`, `catch` oraz `throw`. Ogólna postać bloku instrukcji `try...catch()` ma postać:

```
try // "próbuj" wykonać operację
  {
     // ciąg wykonywanych operacji
  }
catch(lista argumentów)  // jeżeli operacja nie powiodła
                         // się, „przechwyć wyjątek"
  {
      // przetwarzanie wyjątku
      // jeżeli nastąpił wyjątek, wyświetl komunikat
  }
```

Każdy pojawiający się wyjątek zostanie przechwycony i odpowiednio przetworzony przez instrukcję `catch()`, występującą bezpośrednio po `try`.

Wyjątki generowane są instrukcją `throw`. Po słowie kluczowym `throw` występuje dowolne wyrażenie określonego typu. Wartość tego wyrażenia jest zgłaszana jako wyjątek:

```
throw wyjątek;
```

Każda taka instrukcja powinna znajdować się w bloku `try...catch()` lub w jednej z wywoływanych tam funkcji.

Gdy wyjątek zostanie wygenerowany za pomocą instrukcji `throw`, biblioteka czasu wykonania RTL (ang. *run time library*):

- Pobiera wyjątek określając jego typ.
- Przeszukuje stos wywołań funkcji w poszukiwaniu takiej, która zawiera mechanizmy obsługujące wyjątek danego typu (czyli odpowiednią instrukcję `catch`).
- Jeżeli odpowiednia obsługa wyjątku zostanie przechwycona, stos wywołań funkcji jest rozwijany w celu obsługi wyjątku.

- Jeżeli w trakcie poszukiwania stosu wywołań funkcji nie został znaleziony pasujący blok obsługi wyjątku, następuje powrót do programu głównego (w C++ funkcji `main()`).
- Jeżeli i w tym miejscu procedura obsługi zgłoszonego wyjątku nie zostanie przechwycona, program jest przerywany w trybie awaryjnym.
- W trakcie przeszukiwania kolejnych bloków w stosie wywołań funkcji, usuwane są wszystkie obiekty automatyczne, co oznacza wywoływanie ich destruktorów.

Na listingu 7.1 zaprezentowano zmodyfikowany kod z listingu 1.11 (patrz Rozdział 1), w którym użyto instrukcji `throw` do zgłaszania wyjątków sygnalizujących przepełnienie stosu danych lub próbę usunięcia ze stosu nieistniejącej danej.

Listing 7.1.Generowanie wyjątków dla stosu danych

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

template <class T, size_t size = 50>
class TStack {
  private:
    T st[size];
    int put;
  public:
    TStack();
    void push(T i);
    T pop();
};
//-----------------------------------------------------------
//konstruktor
template <class T, size_t size = 50> TStack<T>::TStack()
{
  put = 0;
  cout << "Utworzono stos danych\n\n";
};
//-----------------------------------------------------------
template <class T, size_t size=50> void TStack<T>::push(T i)
{
  if (put >= size) {
    throw "Zapełniono stos\n";
  }
  st[put] = i;
  put++;
};
```

```
//-----------------------------------------------------------
template <class T,size_t size = 50> T TStack<T>::pop()
{
  if (put == 0) {
    throw "\n\nBrak elementow na stosie\n";
  }
  put--;
  return st[put];
};
//-----------------------------------------------------------
int main()
{
  TStack<int,50> x;
  for(int i=0;i<10;i++)
    x.push(i); //wygenerowanie wyjątku
  for(int i=0;i<10;i++)
    cout << x.pop();
  cout << x.pop();//ew. wygenerowanie wyjątku
  cin.get();
  return 0;
}
```

C++ nie ma funkcji bibliotecznej obliczającej średnią harmoniczną wyrazów tablicy otwartej. Średnią harmoniczną stosuje się wtedy, gdy wartości cechy mierzalnej są podane w przeliczeniu na stałą jednostkę innej zmiennej, czyli w postaci wskaźników natężenia.

Istnieją eksperymenty, w których określa się np. średni czas x_i życia zwierząt poddawanych działaniu różnego rodzaju nowych środków farmakologicznych. W analizie takiego rodzaju eksperymentów nie wylicza się średniej arytmetycznej czasu życia zwierząt biorących udział w doświadczeniu, gdyż czas taki jest trudny do określenia. Zamiast wyliczania średniej arytmetycznej, do obliczeń stosuje się średnią harmoniczną, będącą ilorazem liczby obserwacji i sumy odwrotności danych liczbowych:

$$\langle x_h \rangle = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Jako przykład użycia bloku instrukcji `try` i bezparametrowej `catch()` rozpatrzmy sytuację, w której chcemy obliczyć średnią harmoniczną z elementów pewnej tablicy otwartej. Jak wiemy, operacja dzielenia przez zero nie jest możliwa do wykonania. Pokazana na listingu 7.2 funkcja `harmonicMean()` oblicza

średnią harmoniczną niezerowych i nieujemnych elementów tablicy otwartej `data`.

Listing 7.2. Zapis funkcji `harmonicMean()` z wykorzystaniem bloku instrukcji `try...catch()`

```
long double harmonicMean(const double *data,
                         const int dataSize)
{
  long double sum = 0, HM;
  for (int i=0; i<= dataSize; i++)
    {
      try {
            sum += 1.0/data[i];
      }
      catch(...){
        cout << "Niedozwolone parametry funkcji HM()"
             << endl;
      }

    };
    HM = (dataSize+1.0)/sum;
  return HM;
}
```

Jeżeli jednym z elementów tablicy otwartej będzie liczba zero:

```
double data[] = {1, 2, 3.9, 4.9, 5, 9, 0};
```

wówczas w trakcie działania programu kompilator niezwłocznie poinformuje nas o przykrym fakcie wystąpienia próby dzielenia przez zero. Argumentem funkcji obliczającej średnią harmoniczną nie powinna być też tablica otwarta, zawierająca elementy w postaci kombinacji liczb ujemnych i dodatnich, gdyż w takim przypadku mianownik we wzorze na średnią harmoniczną może się zerować. Jeżeli funkcję `harmonicMean()`, pokazaną na listingu 7.2, wywołamy z argumentem w postaci tablicy:

```
double data[] = {1, 2, 3.9, 4.9, 5, 9, -7, 0 };
```

wynik działania programu jako całości będzie identyczny jak poprzednio. Po przetestowaniu omawianego wcześniej wariantu funkcji bez trudu przekonamy się, iż nie otrzymamy dostatecznej informacji o błędach w deklaracji elementów tablicy otwartej `data`.

W celu otrzymania pełniejszej informacji o występujących nieprawidłowościach można zastosować instrukcję `throw`, tak jak pokazuje to listing 7.3.

Listing 7.3. Zapis funkcji `harmonicMean()` z wykorzystaniem bloku `try...catch()` oraz instrukcji `throw`

```
long double harmonicMean(const double *data,
                         const int dataSize)
{
  long double sum = 0, hM;
  for (int i=0; i<= dataSize; i++)
    {
      try {
        if ((! data[i]) || (data[i]<0)) throw data[i];
        sum += 1.0/data[i];
      }
      catch(...){
        cout << "Niedozwolone parametry funkcji HM()"
             << endl;
      }

    };
    hM = (dataSize+1.0)/sum;
  return hM;
}
```

W tym przypadku w trakcie działania programu otrzymamy tyle informacji o błędach, ile jest błędnie zadeklarowanych elementów tablicy będącej argumentem funkcji `harmonicMean()`.

Dla bardziej wymagającego użytkownika pokazany wcześniej sposób przechwytywania błędów może okazać się niewystarczający. Pójdźmy krok dalej. Zażądamy mianowicie, aby uruchomienie programu zawierającego np. tablicę otwartą z błędnie określonymi elementami okazało się niemożliwe do zrealizowania. W tym celu skorzystajmy z bloku `try` oraz instrukcji `catch()` z parametrem będącym zarazem pierwszym argumentem funkcji `harmonicMean()`. Zapis:

```
//...
catch(const double *data)
//...
```

spowoduje jawne wskazanie na przechwytywany obiekt wyjątku (tzn. wskaźnik do typu `double`). W ten oto sposób, w przypadku błędnej deklaracji jakiegokolwiek elementu tablicy otwartej wskazywanej przez `data`, program

wykonywalny powinien zostać przerwany w trybie awaryjnym. Sytuację tę ilustruje listing 7.4.

Listing 7.4. Zapis funkcji `harmonicMean()` z wykorzystaniem parametryzowanego bloku `try...catch()` oraz instrukcji `throw`

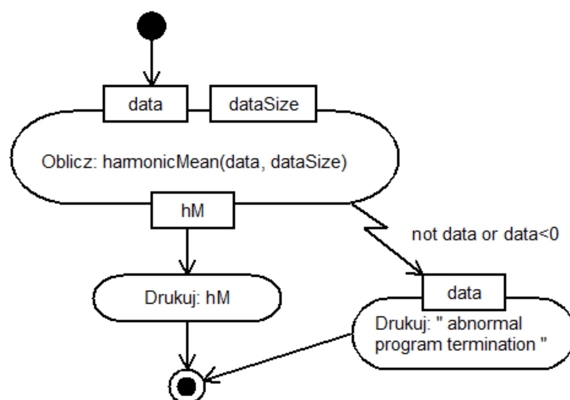
```
#include <iostream>
using namespace std;
// prototyp funkcji harmonicMean()
long double harmonicMean(const double *data,
                         const int dataSize);
int main()
{
  double data[] = {1, 2, 3.9, 4.9, 5, 9, 7, 0};
  long double hM = harmonicMean(data,
                   (sizeof(data)/sizeof(data[0]))- 1);
  cout << "Średnia harmoniczna= " << hM << endl;
  cin.get();
  return 0;
}
//-----------------------------------------------------------
long double harmonicMean(const double *data,
                         const int dataSize)
{
  long double sum = 0, hM;
  for (int i=0; i<= dataSize; i++)
    {
      try {
        if ((! data[i]) || (data[i]<0)) throw data[i];
        sum += 1.0/data[i];
      }
      catch(const double *data){}

    };
    hM = (dataSize+1.0)/sum;
  return hM;
}
```

Propagacja wyjątku może zakończyć się dwoma wariantami:

- Jego obsługą w bloku `try...catch`, po czym wykonywane są kolejne instrukcje następujące po tym bloku.
- Przerwaniem wykonania programu, gdy wyjątek nie zostanie prawidłowo przechwycony i obsłużony.

W przypadku, gdy wyjątek nie zostanie prawidłowo przechwycony przez instrukcję `catch()`, to znaczy, gdy nie istnieje instrukcja `catch()` odpowiadająca wygenerowanemu wyjątkowi, automatycznie zostanie wywołana funkcja `terminate()` z modułu *except.h*. Funkcja `terminate()` wywołuje z kolei inną funkcję `abort()`, kończącą w sposób natychmiastowy działanie programu (ang. *abnormal program termination*).

Na rysunku 7.1 pokazano diagram czynności dla przykładu z listingu 7.4. Na diagramie zaznaczono sytuację, w której program jest kończony w przypadku dokonania próby przetworzenia wyjątku dla niewłaściwych danych wejściowych.



Rysunek 7.1. Diagram czynności dla programu nieprawidłowo przechwytującego wyjątki

Podczas konstruowania bardziej zaawansowanych programów można wykorzystać funkcje `set_terminate()` lub `set_unexpected()`:

```
set_terminate(terminate_handler pnew) throw();
```

```
set_unexpected(unexpected_handler pnew) throw();
```

umożliwiające zdefiniowane własnych, niestandardowych procedur obsługi wyjątków, tak jak przedstawiono to listingu 7.5.

Listing 7.5. Definiowanie własnych procedur obsługi wyjątków

```
#include <except>
#include <iostream>

using namespace std;

void terminateFunc();

int main()
```

```
{
  int i = 10, j = 0;
  set_terminate(terminateFunc);
  try {
    if(j == 0)
      throw "Dzielenie przez zero!";
        else
          cout << i/j;
  }
  catch(int) {
     cout << "Komunikat...\n";
  }
  cin.get();
  return 0;
}
//-----------------------------------------------------------
void terminateFunc()
{
  cout << "Wywołano funkcję terminateFunc()\n";
  cin.get();
  exit(-1);
}
```

Wynik działania programu:

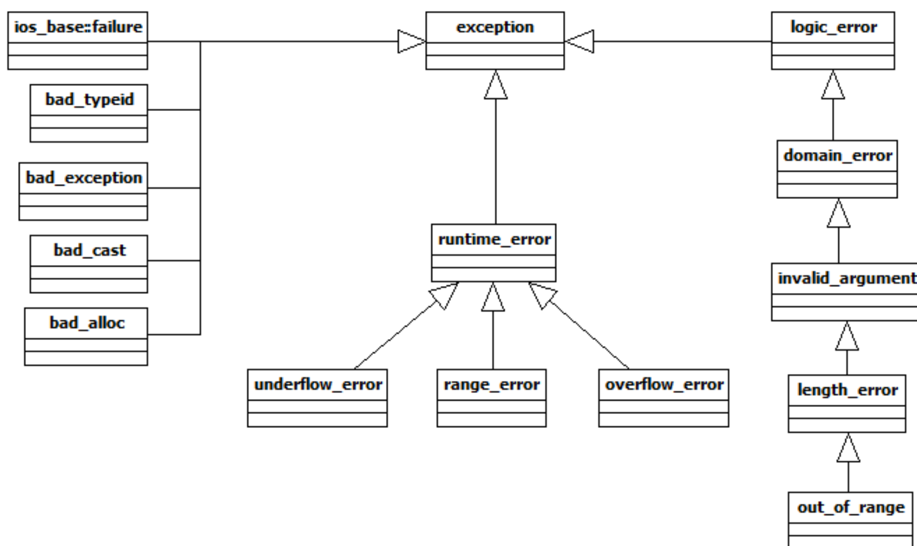
```
Wywolano funkcje terminateFunc()
```

7.2. Klasy wyjątków

Wyjątki pozwalają osobie piszącej kod na uwzględnienie sytuacji, w której program w jakimś momencie wykonywania może ulec niekontrolowanemu zakończeniu. Wyjątek może być umiejscowiony w dowolnej metodzie. W standardzie ANSI/ISO C++ podstawową klasą, zajmującą się obsługą wyjątków jest `exception` (wyjątek) zdefiniowana w pliku nagłówkowym *except.h*. Dziedzicząc po bazowej klasie wyjątku mamy pewność, że kod, który przechwytuje standardowe typy wyjątków, poprawnie obsłuży też samodzielnie zdefiniowane przez programistę. W większości wypadków dziedziczenie po klasie `exception` powinno być wystarczające. Hierarchia standardowych klas wyjątków C++ została przedstawiona na rysunku 7.2.

Korzystanie z bibliotecznych klas wyjątków, zwłaszcza w trakcie ich strukturalnego przechwytywania, może w wielu przypadkach być czynnością mało efektywną, chociażby z tego względu, że należy dokładnie orientować się w nazwach odpowiednich klas, ich zasobach oraz hierarchii. W sytuacji, gdy programista nie ma wystarczającej wiedzy na temat liczby i rodzaju interesujących go klas wyjątków, bez większych problemów może zaprojektować swoją własną

klasę wyjątku [22]. Pokazana na listingu 7.6 klasa `TMyException` reprezentuje uniwersalną klasę wyjątków, w której nie precyzujemy dokładnie, jaki rodzaj błędów w przyszłości będzie diagnozowany. Zadaniem przeładowanego konstruktora `TMyException()` jest skopiowanie do obszaru pamięci wskazywanego przez odpowiedni wskaźnik za pomocą funkcji `_mbsdup()` łańcucha znaków, reprezentującego określony wyjątek. Pierwsza postać konstruktora użyta jest w instrukcji `throw`, druga zaś jego postać —w instrukcji `catch()` strukturalnego bloku obsługi wyjątków.



Rysunek 7.2. Hierarchia klas wyjątków C++. Należy pamiętać, iż niektóre typy wyjątków wymagają dołączenia odpowiednich plików nagłówkowych biblioteki standardowej

Listing 7.6. Kod modułu wykorzystującego uniwersalną klasę wyjątków

```
#include <iostream>
#include <excpt>
#include <mbstring> // _mbsdup()
using namespace std;

class TMyException: public exception {
  public:
    TMyException(const char *s)
                {ss = _mbsdup(s);}
    TMyException(const TMyException &e )
                {ss = _mbsdup(e.ss); }
      // destruktor, za pomocą funkcji
      // free() zwalnia obszar pamięci wskazywany
```

```
      // przez wskaźnik ss
    virtual ~TMyException() {free(ss);}
    const char *message() const {return ss;}
  private:
    unsigned char *ss;
    unsigned char *s;
};
//-----------------------------------------------------------
int main()
{
  double data[] = {2, 2, 2, 0};
  double sum = 0;
  try {
     for (int i=0; i< sizeof(data)/sizeof(data[0]); i++){
        if(data[i] == 0)
          throw(TMyException("Błąd dzielenia przez 0"));
        sum += 1.0/data[i];
     }
  }
  catch(const TMyException &e) {
      cout << e.message() << endl;
  }
  cout << sum;
  cin.get();
  return 0;
}
```

Podsumowanie

Możliwość obsługi wyjątków daje do dyspozycji programistom niezwykle wydajne narzędzie, pozwalające kontrolować błędy czasu wykonywania programu w jego najbardziej newralgicznych punktach. Niemniej jednak należy zdawać sobie sprawę z faktu, iż wyjątki w żadnym wypadku nie mogą być używane jako prosta metoda zwracania komunikatów przez program w zupełnie niegroźnej sytuacji. Wynika to z faktu, iż wygenerowanie i obsłużenie wyjątku pociąga za sobą konieczność przydzielenia określonych zasobów procesora, który w tym czasie może być zajęty przez zupełnie inne zadania. Wyjątki należy traktować jako zło konieczne i w żadnym wypadku nie można ich używać jako normalnej drogi zwracania informacji.

ROZDZIAŁ 8

MODELOWANIE INTERFEJSU GUI

8.1. Elementy GUI 230
8.2. Konstruowanie GUI. Podejście deklaratywno-proceduralne 231
8.3. Konstruowanie GUI. Podejście RAD 236
8.4. Przydatne techniki modelowania GUI 238
Podsumowanie 240

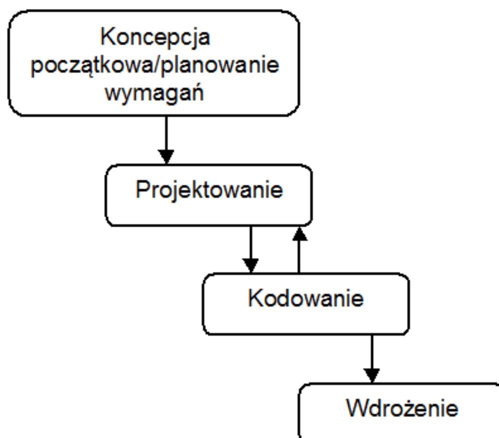
8.1. Elementy GUI

Obecny rozdział omawia podstawowe zasady obiektowo zorientowanego projektowania graficznych interfejsów użytkownika GUI (ang. *Graphical User Interface*). Istnieją trzy główne podejścia do projektowania i programowania GUI [26].

Pierwszym z nich jest koncepcja oparta na formalizmie czysto proceduralnym. Interfejs użytkownika tworzony jest jako wynik wykonywania przez programistę sekwencji poleceń graficznych. Polecenia te mogą mieć charakter imperatywny, jak w przypadku tcl/tk lub obiektowy (Java AWT – Abstract Window Tolkit i rozszerzenia dla komponentów Swing). Za pomocą odpowiednio konstruowanych poleceń graficznych w praktyce można stworzyć dowolny rodzaj GUI. Wadą tej metody jest stosunkowo duża pracochłonność i znaczny stopień komplikacji w porównaniu z pozostałymi metodami tworzenia interfejsu użytkownika.

Kolejnym, jest podejście deklaratywne polegające na wyborze odpowiednich opisowych deklaracji (konstrukcji) spośród predefiniowanego ich zbioru. Klasycznym przykładem stosowania tego rodzaju podejścia jest język znaczników HTML, przeznaczony do tworzenia stron internetowych. Podejście deklaratywne jest bardzo proste w praktycznej realizacji, jednak jego wadą jest niemal całkowity brak ekspresywności.

Technologicznie najbardziej zaawansowanym podejściem jest wykorzystanie narzędzi projektowania wizualnego, tzw. „builderów”. W tym przypadku GUI projektowane i tworzone jest bezpośrednio przez programistę w zintegrowanych środowiskach programistycznych typu RAD (ang. *Rapid Application Development*) [22,18,19]. Przykładem tego typu środowisk są: Delphi, Lazarus, C++ Builder, Visual Studio, Visual Basic, Qt Creator. Niewątpliwą zaletą stosowania wizualnych narzędzi jest wytworzenie sprzężenia zwrotnego projektant - programista.



Rysunek 8.1. Proces RAD SDLC

Programista na bieżąco śledzi i weryfikuje efekty swoich działań, zaś tak tworzone GUI jest od razu integrowane z kodem opisującym dziedzinę problemu. Kolejną zaletą podejścia wizualnego jest możliwość dynamicznego kreowania (i usuwania) elementów GUI, które w zasadzie są elastycznymi komponentami z dobrze zdefiniowanymi przez producenta interfejsami. Istnieje metodologia znana jako RAD SDLC (ang. *Rapid Application Development Systems Development Live Cycle*) dość dobrze opisująca poszczególne etapy prowadzące do skonstruowania programu z graficznym interfejsem użytkownika z wykorzystaniem narzędzi programistycznych typu RAD. Na rysunku 8.1 pokazano główne założenia leżące u podstaw RAD SDLC [8].

Zgodnie z ogólnie akceptowanym modelem [26], graficzny interfejs użytkownika składa się z pięciu głównych elementów:

- *Okna i widgety*. Okno (zwane inaczej formularzem) jest prostokątnym fragmentem ekranu zawierającym zbiór widgetów (kontrolek), czyli elementów pierwotnych interfejsu graficznego wchodzących w skład okna i zorganizowanych zgodnie z wymogami projektu (nazwa *widget* jest skrótem od *window gadget*). Zachowanie każdego widgetu jest zdeterminowane przez protokół szczegółowo definiujący reguły interakcji z użytkownikiem. W językach programowania, widget jest wyrażany przez klasę, strukturę lub rekord określający jego typ, stan początkowy, odwołanie do elementu właściciela oraz zbiór oferowanych metod. W zintegrowanych narzędziach programistycznych typu RAD, widget jest reprezentowany przez graficzny komponent czasu projektowania będący implementacją predefiniowanego zbioru klas. Komponent taki realizuje jednoznacznie zdefiniowany interfejs.
- *Zdarzenia i akcje*. Zdarzeniem (z punktu widzenia GUI) nazywamy dobrze określoną interakcję ze światem zewnętrznym, wywołaną przez określone warunki lub określone działanie użytkownika. Podstawowymi elementami każdego zdarzenia są jego typ i czas wystąpienia. Zdarzenia wyrażane są (manifestowane są) za pośrednictwem akcji, których skutkiem jest wywołanie pewnej funkcji lub procedury obsługi wymienionej akcji.
- *Zarządcy*. Zarządca (ang. *handler*) jest obiektem, za pośrednictwem którego program kontroluje funkcjonowanie widgetu. Każdy widget powiązany jest ze swoim zarządcą.

8.2. Konstruowanie GUI. Podejście deklaratywno-proceduralne

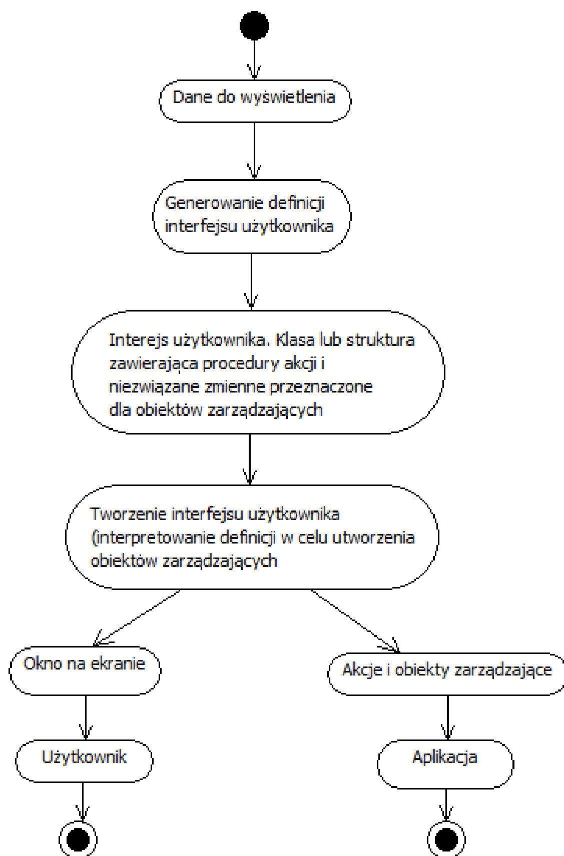
W ramach podejścia deklaratywnego programista posługuje się zbiorem dostępnych struktur danych opisujących interfejs użytkownika. Tego rodzaju struktury danych definiowane są poprzez przypisywanie określonych wartości ich atrybutom. Podejście czysto deklaratywne bardzo ułatwia manipulowanie definicją interfejsu. W wyniku stosowania podejścia proceduralnego, interfejs użytkownika powstaje w wyniku realizowania przez program niepodzielnych, predefiniowanych operacji konstruujących interfejs graficzny. W praktycznych

zastosowaniach łączy się dwa wymienione wcześniej podejścia zgodnie z opisanym poniżej schematem.
Dla każdego okna stanowiącego zasadniczą część GUI w sposób deklaratywny definiowane są cztery główne elementy:

- Statyczna struktura okna w postaci zbioru zagnieżdżonych widgetów, z których każdy jest niepodzielnym, podstawowym elementem GUI.
- Typy poszczególnych widgetów.
- Początkowe stany każdego z używanych widgetów.
- Sposoby reakcji widgetów na ewentualne zmiany rozmiaru głównego okna aplikacji, czyli zmiany ich rozmiarów i względnego położenia.

W sposób proceduralny definiowane są:

- Procedury lub funkcje wykonywane jako rezultat wystąpienia określonych zdarzeń zewnętrznych generowanych przez użytkownika lub sam program.
- Obiekty zarządzające (handlery) wywoływane w celu dokonania zmian w interfejsie użytkownika.



Rysunek 8.2. Deklaratywno-proceduralne budowanie interfejsu GUI

W podejściu deklaratywno-proceduralnym definicja GUI konstruowana jest w postaci zagnieżdżonych struktur danych zawierających wbudowane funkcje, procedury oraz obiekty stanowe. Deklaratywna część każdej ze struktur danych może być z łatwością modyfikowana, zaś za pomocą odpowiednich funkcji i procedur obsługi zdarzeń można wykonywać dowolne obliczenia. Sieć działań prowadząca do budowania interfejsu GUI w oparciu o tradycyjne podejście deklaratywno-proceduralne została przedstawiona na rysunku 8.2.

Na listingu 8.1 zaprezentowano przykład nieskomplikowanego programu skonstruowanego zgodnie z podejściem deklaratywno-proceduralnym, który tworzy główne okno GUI, jeden widget typu okienka edycyjnego (ang. *edit*) i dwa widgety typu przycisków (ang. *button*), według konwencji stosowanej w API Windows. Analizując poniższy kod należy być świadomym faktu, iż każdy program reprezentowany jest przez własny proces w systemie. Oznacza to, że proces, taki może stworzyć własne okno oraz odpowiednio zarządzać i sterować nim. Okno (formularz) jako główny element GUI nie jest programem, lecz jedynie elementem procesu. Każdy proces może stworzyć wiele różnych okien i wszystkie te okna, będą sterowane za pomocą odpowiednio konstruowanych pętli komunikatów przetwarzanych przez jeden program (proces). Rysunek 8.3 obrazuje wynik działania programu. Konstruując program pokazany na listingu 8.1 wykorzystano standardowe typy danych, funkcje oraz pętle komunikatowe udostępniane przez interfejs programisty API Windows.

Listing 8.1. Kod programu tworzący okno jako główny element GUI

```
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <tchar.h>

enum {ID_EDIT,ID_BUTTON1,ID_BUTTON0};

// Nazwa klasy głównego okna
static TCHAR szWindowClass[] = _T("Klasa Okna");
//Tytuł okna
static TCHAR szTitle[] = _T("Tytuł okna");
HINSTANCE hInst;
HWND button0;
HWND button1;
HWND edit;
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//-----------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
                   hPrevInstance,LPSTR lpCmdLine,
                   int nCmdShow)
{
   //Deklarcja struktury klasy okna o nazwie wcex
```

```
    WNDCLASSEX wcex;
    //Rozmiar klasy okna
    wcex.cbSize = sizeof(WNDCLASSEX);
    //Styl klasy okna
    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    //Procedura obsługi dla klasy okna
    wcex.lpfnWndProc    = WndProc;
    wcex.cbClsExtra     = 0;
    wcex.cbWndExtra     = 0;
    //Zarządca procesu programu
    wcex.hInstance      = hInstance;
    //Wygląd okna (ikony, kursory, tło, typ menu
    wcex.hIcon          = LoadIcon(hInstance,
                          MAKEINTRESOURCE(IDI_APPLICATION));
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = NULL;
    //Nazwa klasy okna
    wcex.lpszClassName  = szWindowClass;
    wcex.hIconSm        = LoadIcon(wcex.hInstance,
                          MAKEINTRESOURCE(IDI_APPLICATION));
    //Rejestracja klasy okna w systemie
    if (!RegisterClassEx(&wcex)) {
        MessageBox(NULL,
            _T("Błędna rejestracja! "),
            _T("Klasa okna"),
            NULL);
         return 1;
    }

    hInst = hInstance;
    //Tworzenie okna o nazwie wskazywanej przez szTitle przy
    //użyciu funkcji API CreateWindow()
    HWND hWnd = CreateWindow(
        szWindowClass,
        szTitle,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT,
        500, 300,  //Rozmiar okna (szerokosc i wysokosc)
        NULL,
        NULL,
        hInstance,
        NULL
    );
    if (!hWnd) {
```

```
        MessageBox(NULL,
            _T("Błędne wywołanie funkcji CreateWindow!"),
            _T("Tytuł okna"),
            NULL);
        return 1;
    }
    //Wyświetlanie okna i jego odświeżanie
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    MSG msg; //Struktura komunikatów
    // Główna pętla obsługująca wymianę komunikatów
    while (GetMessage(&msg, NULL, 0, 0)) {
        //Funkcja tłumacząca sygnały z klawiatury na
        //odpowiednie komunikaty systemowe
        TranslateMessage(&msg);
        //Funkcja przetwarzająca komunikaty systemowe
        //przez procedury obsługi
        DispatchMessage(&msg);
    }
    return (int) msg.wParam;
}
//-----------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;
    TCHAR greeting[] = _T("Komunikat");

  switch (message){
    case WM_CREATE:
       button0 = CreateWindow("Button","Cancel Button",
       BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE ,
       60,220,350,35,hWnd,(HMENU)ID_BUTTON0,hInst,0);
       button1 = CreateWindow("Button","Ok Button",
       BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE
       ,60,170,350,35,hWnd,(HMENU)ID_BUTTON1,hInst,0);
       edit = CreateWindow("Edit", NULL,WS_BORDER | NULL |
       WS_CHILD | WS_VISIBLE | NULL | NULL ,
       35,45,400,20,hWnd,(HMENU)ID_EDIT,hInst,0);
       break;
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        //Współrzędne wyświetlanego tekstu w oknie
```

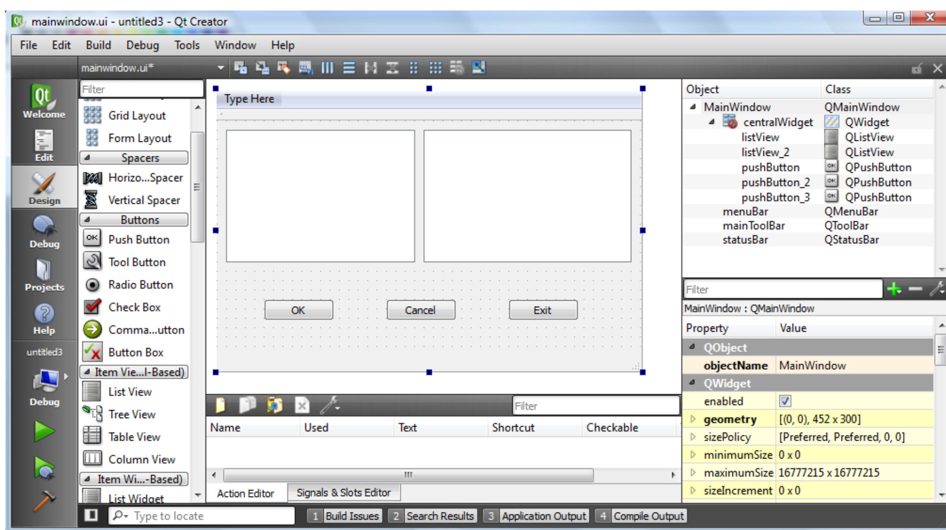
```
        TextOut(hdc,5, 5, greeting, _tcslen(greeting));
        EndPaint(hWnd, &ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    case WM_COMMAND:
        switch(wParam) {
           case ID_BUTTON0:
           SetWindowText(edit,"Komunikat 1");
           break;
          case ID_BUTTON1:
           SetWindowText(edit,"Komunikat 2");
           break;
        }
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
        break;
  }
    return 0;
}
```

Rysunek 8.3. Główne okno graficznego interfejsu użytkownika

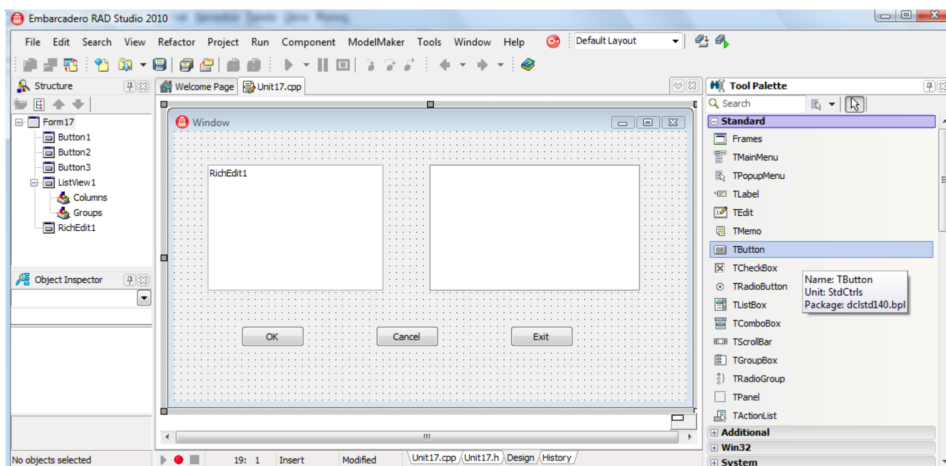
8.3. Konstruowanie GUI. Podejście RAD

Proces budowania GUI w ujęciu RAD przedstawiony został schematycznie na rysunku 8.1. Rozpoczyna się od określenia danych, które w ramach interfejsu

mają zostać wyświetlone. Na bazie tej specyfikacji tworzony jest interfejs w postaci zbioru komponentów graficznych (widgetów). Poszczególne komponenty umieszczane są w obrębie głównego okna (formularza) aplikacji metodą „przeciągnij i upuść” (ang. *drag-and-drop*). Na rysunkach 8.4 i 8.5 pokazano odpowiednio wygląd zintegrowanego środowiska programisty IDE udostępnianego przez Qt Creator oraz C++ Builder 2010.



Rysunek 8.4. IDE Qt Creator



Rysunek 8.5. IDE Embarcadero C++ Builder 2010

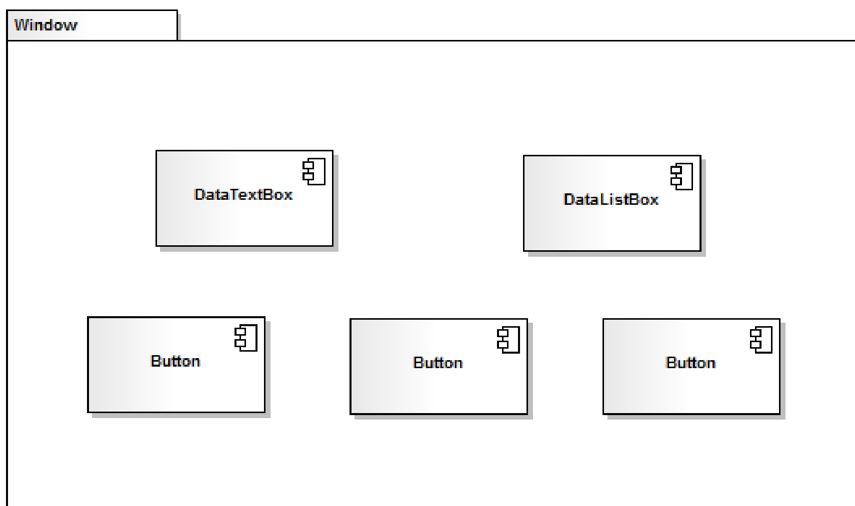
Każdy z umieszczanych w obrębie formularza (głównego okna GUI) komponentów wyrażany jest przez odpowiedni zestaw klas. Każda z klas zawiera

implementacje realizujące poszczególne akcje oraz niepowiązane zmienne, które w przyszłości staną się odwołaniami do odnośnych obiektów zarządzających. Reguły te interpretowane są przez kompilator, który fizycznie buduje interfejs użytkownika wykonując dla każdego komponentu dwie zasadnicze czynności:

- Tworzy okno za pomocą wewnętrznego pakietu graficznego.
- Konstruuje wewnętrzne mechanizmy umożliwiające interakcję okna z aplikacją. W szczególności dla każdego okna tworzony jest odrębny watek, zaś każdemu widgetowi (komponentowi) okna przyporządkowany jest obiekt zarządzający. Każda procedura (funkcja) widgetu automatycznie kojarzona jest ze zdarzeniami, które są wywoływane przez użytkownika lub inne części aplikacji.

8.4. Przydatne techniki modelowania GUI

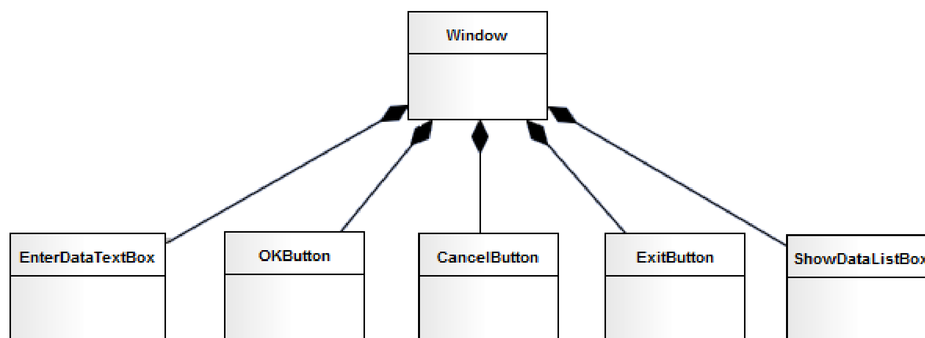
Istnieje wiele sposobów konstruowania i modelowania graficznego interfejsu użytkownika w oparciu o diagramy UML [18,19,22]. Do najczęściej wykorzystywanych należy zaliczyć modelowanie oparte na diagramach komponentów, diagramy klas oraz diagramy hybrydowe. W UML nie istnieje oddzielny symbol obrazujący formularz (okno interfejsu graficznego), dlatego też obszar formularza zwyczajowo modelowany jest symbolem pakietu, czyli bytu, który grupuje inne logicznie powiązane elementy stając się ich właścicielem, tak jak pokazano to na rysunku 8.6.



Rysunek 8.6. Wygląd formularza wymodelowanego na podstawie diagramu komponentów

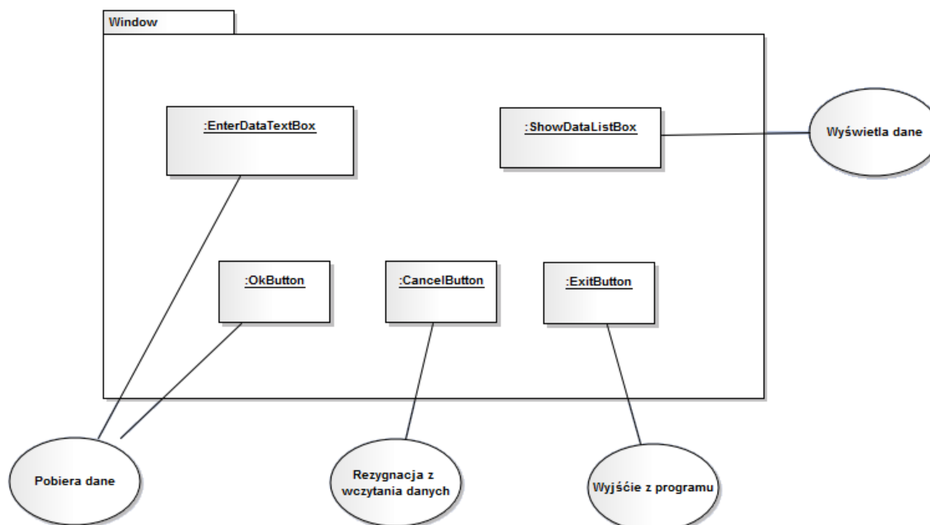
Na rysunku 8.7 pokazano to samo GUI wymodelowane przy pomocy diagramu klas. Klasa `Window` jest formą całkowicie agregującą wszystkie pozostałe klasy. Na tym diagramie zaznaczono jedynie to, że formularz aplikacji składa się

z określonych elementów, bez wnikania w dalsze szczegóły.



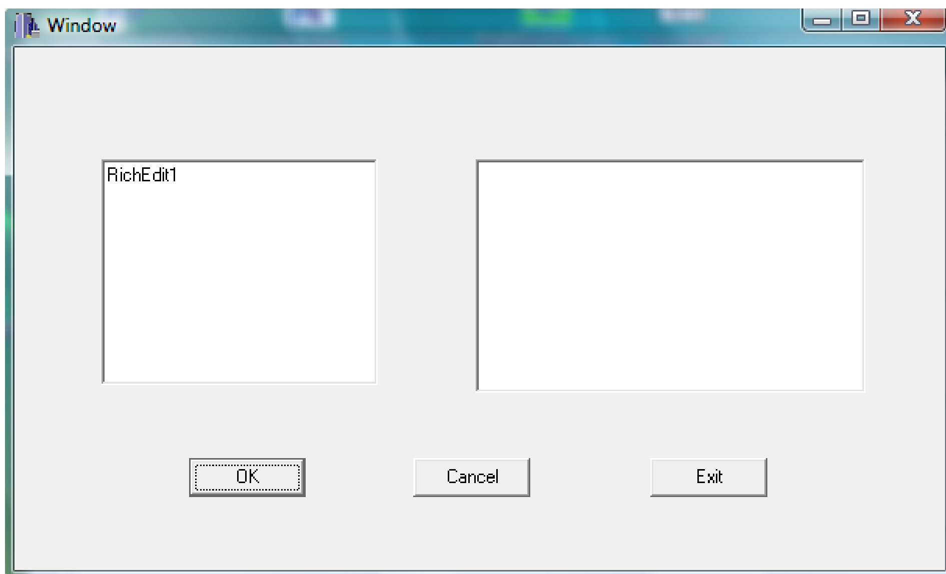
Rysunek 8.7. Modelowanie GUI oparte na diagramie całkowicie agregujących klas

Pomimo że diagramy z rysunków 8.6 – 8.7 niosą dość istotne informacje na temat wyglądu i składu GUI, jednak w wielu wypadkach mogą okazać się niewystarczające. Uniwersalnym sposobem jest podejście hybrydowe. Diagramy hybrydowe, stosowane do modelowania wyglądu graficznego interfejsu użytkownika, łączą diagramy obiektów oraz dokładnie opisane przypadki ich użycia, co jednoznacznie określa ich rozmieszczenie, nazewnictwo oraz rolę w programie, tak jak pokazuje to rysunek 8.8.



Rysunek 8.8. Modelowanie GUI w oparciu o diagram hybrydowy

Na rysunku 8.9 zaprezentowano odpowiadający rysunkowi 8.8 wygląd GUI realnie działającego programu.



Rysunek 8.9. Wygląd GUI działającej aplikacji typu RAD

Podsumowanie

Obecny rozdział zawierał krótkie wprowadzenie do zagadnień związanych z projektowaniem graficznego interfejsu użytkownika. Zaprezentowano kilka praktycznych sposobów dokumentowania GUI. Wiele innych metod projektowania oraz konstruowania GUI można znaleźć w bogatej literaturze przedmiotu [36-41].

DODATEK A

PRZYKŁADOWE ĆWICZENIA DO SAMODZIELNEGO WYKONANIA

Dodatek zawiera przykładowe zestawy zadań do samodzielnego wykonania. Oczywiście, do każdego z rozdziałów można zaproponować bardzo wiele różnych ćwiczeń. Poniższe zestawy należy traktować jako ogólne, ramowe propozycje, które można samodzielnie, w dowolny sposób rozszerzać na wszystkie zagadnienia poruszane w podręczniku.

Zadania do rozdziału 3

1. Przetestuj kody z rozdziału 3. Postaraj się samodzielnie, tam gdzie to jest możliwe w postaci diagramów sekwencji przedstawić dynamiczne zachowanie się działających systemów.

2. Zaimplementuj kod z listingu 3.9 w ten sposób, aby można było wykonać enumerację aktualnie przyłączonych do magistrali USB urządzeń (patrz przykład z listingu 6.1).

3. Poeksperymentuj ze statycznym polimorfizmem. Postaraj się zastosować diagram z rysunku 3.6 do problemu enumeracji urządzeń USB i ew. Bluetooth.

Zadania do rozdziału 4

1. Zaimplementuj diagram z rysunku 4.7 w ten sposób, aby program mógł obliczać pole powierzchni wybranych figur geometrycznych.

2. Po przetestowaniu, kod zmodyfikuj tak, aby interfejsy posługiwały się identyfikatorami GUID.

Zadania do rozdziału 5

1. Na listingu 6.1 pokazano konstrukcje przykładowych funkcji umożliwiających wykonanie enumeracji urządzeń aktualnie dołączonych do magistrali USB. Kod ten zaimplementuj w postaci singletonu.

2. Na listingu 6.2 pokazano konstrukcje przykładowych funkcji umożliwiających wykonanie enumeracji urządzeń Bluetooth. Kod ten zaimplementuj w postaci singletonu.

3. Zmodyfikuj kody z listingów 6.1 oraz 6.2 w ten sposób, aby można je było zastosować do wzorca fabryki abstrakcyjnej.

4. Zmodyfikuj kod wzorca fabryki abstrakcyjnej w tren sposób, aby interfejsy były identyfikowane za pomocą identyfikatorów GUID.

Zadania do rozdziału 6

1. Zmodyfikuj kod z listingów 4.2-4.3 w ten sposób, aby komponent `MyComp` mógł być eksportowany z biblioteki dołączanej dynamicznie *(.dll)*.

2. Zmodyfikuj kod z listingu 5.6 w ten sposób, aby obiekt klasy `TWINTimer` mógł być eksportowany z biblioteki *dll*.

3. Zmodyfikuj kod z listingu 5.7 w ten sposób, aby obiekt klasy `TWINTimer` mógł być eksportowany z biblioteki *dll*.

4. W systemach operacyjnych Windows i Linux zaimplementuj odpowiednio bibliotekę *.dll* i *.so* eksportujące odpowiednio obiekty klas `TSytsemA` oraz `TSystemB`. Zmodyfikuj kod a listingów 5.9-5.10 w ten sposób, aby program klienta automatycznie ładował odpowiednią bibliotekę w zależności od systemu operacyjnego, w którym będzie kompilowany.

Zadania do rozdziału 7

1. Zmodyfikuj kod z listingów 4.2-4.3 w ten sposób, aby w programie klienta można było przechwytywać wyjątki powstałe w wyniku nieprawidłowego wprowadzenia danych wejściowych do kalkulatora figur geometrycznych.

2. Zaprojektuj i zaimplementuj procedury strukturalnej obsługi wyjątków, które mogą być generowane w trakcie prób dynamicznego łączenia programu wykonywalnego z potencjalnie nieistniejącymi bibliotekami współdzielonymi.

3. Zaprojektuj i zaimplementuj procedury strukturalnej obsługi wyjątków, które mogą być generowane w trakcie prób importowani błędnie określonych funkcji eksportowanych z bibliotek współdzielonych.

4. Postaraj się samodzielnie zaimplementować mechanizmy strukturalnej obsługi wyjątków do wybranych wzorców projektowych. Tak uzupełnione wzorce przedstaw w postaci nowych diagramów klas.

Zadania do rozdziału 8

1. Uzupełnij kod z listingu 8.1 w ten sposób, aby stanowił warstwę GUI dla komponentu kalkulatora figur geometrycznych. Kod tego komponentu został przedstawiony na listingach 4.2-4.3.

2. W oparciu o listing 8.1 zbuduj interfejs GUI dla przykładowej implementacji wzorca stanu przedstawionej na listingu 5.15.

BIBLIOGRAFIA

[1] J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, W. Premerlani, Object Oriented Modeling and Design, Prentice Hall 1990.

[2] I. Jacobson, Object Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley 1992.

[3] I. Jacobson, M. Ericsson, A. Jacobson, The Object Advantage: Business Process Reengineering With Object Technology, Addison-Wesley 1994.

[4] G. Booch, Object Oriented Analysis and Design with Applications, Addison-Wesley 1993.

[5] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley 1998. Wyd. polskie: UML przewodnik użytkownika, WNT 2002.

[6] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Modeling Language Reference, 2nd Edition, Pearson Education 2005.

[7] M. Śmiałek, Zrozumieć UML 2.0. Metody modelowania obiektowego, Helion 2005.

[8] J. A. Hoffer, J. F. George, J. S. Valacich, Modern Systems Analysis and Design, Addison-Wesley 1999.

[9] R. C. Martin, The Liskov Substitution Principle, C++ Report, Vol. 8, 1996.

[10] B. Meyer, Object-Oriented Software Construction Edition, Prentice Hall 1988, 2nd Edition 1997.

[11] J. Warmer, A. Kleppe, OCL - precyzyjne modelowanie w UML, WNT 2003.

[12] OMG, Model Driven Architecture Guide Version 1.0.1, 2003.

[13] D.S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Publishing, 2003.

[14] OMG, Meta Object Facility 2.0 Query/View/Transformation Specification.

[15] OMG, MOF 2.0 / XMI Mapping Specification, v2.1.1

[16] OMG, Object Constraint Language (OCL) Specification, version 2.0.

[17] OMG, Unified Modeling Language version 2.2. Superstructure.

[18] A. Daniluk, Model-Driven Development for scientific computing. Computations of RHEED intensities for a disordered surface. Part I, Comput. Phys. Commun. 181 (2010) 707, doi:10.1016/j.cpc.2009.11.009

[19] A. Daniluk, Model-Driven Development for scientific computing. Computations of RHEED intensities for a disordered surface. Part II, Comput. Phys. Commun. 181 (2010) 709, doi:10.1016/j.cpc.2009.11.011

[20] M. Huybrechts, G. Pauwels, S. van Baelen, Agile MDA: Presentation, Validation, Debugging, Generation, AGILE Project Report D.2.6, ITEA-AGILE Consortium 2006.

[21] B. Stroustrup, Programming: Principles and Practice Using C++, Addison-Wesley, 2008. Wyd. polskie: Programowanie. Teoria i praktyka z wykorzystaniem C++, Helion, 2010.

[22] A. Daniluk, C++Builder Borland Developer Studio 2006. Kompendium programisty, Helion, 2006.

[23] J. Coplien, Curiously Recurring Template Patterns, C++ Report (1995) 24.

[24] A. Daniluk, USB. Praktyczne programowanie z Windows API w C++, Helion, 2009.

[25] A. Daniluk, RS 232C - praktyczne programowanie. Od Pascala i C++ do Delphi i Buildera. Wydanie III, Helion, 2006.

[26] P. Van Roy, S. Haridi, Concepts, Techniques, and Models of Computer Programming, Massachusetts Institute of Technology 2004. Wyd. polskie: Programowanie, koncepcje, techniki i modele, Helion 2005.

[27] A. Jaszkiwicz, Inżynieria oprogramowania, Helion 1997.

[28] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley & ACM Press, 1999. Wyd. polskie: Oprogramowanie komponentowe. Obiektowe to za mało, WNT 2001.

[29] Ch. Alexander, S. Ishikawa and M. Silverstein, A Pattern Language,

Buildings, Construction, Oxford University Press, 1977.

[30] M. Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley 1997.

[31] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994. Wyd. polskie: Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku, Helion 2010.

[32] A. A. Stepanov, M. Lee, The Standard Template Library. ISO Programming Language C++ Project, Doc. No. X3J16/94-0095, WG21/N0482, 1994.

[33] S. Meyers, Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library, Addison-Wesley Professional 2001. Wydanie polskie: STL w praktyce. 50 sposobów efektywnego wykorzystania, Helion 2004.

[34] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison--Wesley 2001.

[35] A. Shalloway, J. R. Trott, Design Patterns Explained. A new Perspective on Object-Oriented Design, 2nd Edition, Addison Wesley 2004. Wyd. polskie: Projektowanie zorientowane obiektowo. Wzorce projektowe. Helion 2005.

[36] P. Beynon-Davies, S. Holmes, Design breakdowns, scenarios and rapid application development, Information and Software Technology 44 (2002) 579.

[37] A. Agah, K. Tanie, Intelligent graphical user interface design utilizing multiple fuzzy agents, Interacting with Computers 12, (2000) 529.

[38] D. Draheim, G. Weber, Modelling form-based interfaces with bipartite state machines, Interacting with Computers 17 (2005), 207.

[39] S. V. Mudiam, G. C. Gannod, T. E. Lindquis, Synthesizing and integrating legacy components as services using adapters, Science of Computer Programming, 60 (2006) 134.

[40] C. J. Scogings, C. H. E. Phillips, Linking tasks, dialogue and GUI design: a method involving UML and Lean Cuisine+, Interacting with Computers, 14 (2001) 69.

[41] A. Dix, J. Finlay, G. Abowd, R. Beale, Human-Computer Interaction, Prentice Hall, 3rd Edition 2004.

SKOROWIDZ

A

adres, 41, 45, 83, 84, 88, 219
Agregacja, 1, 24, 25, 28, 30, 129
Akcesor, 11
algorytm, 109, 177, 195
Analiza, 57, 60
API, IV, 75, 126, 163, 164, 205, 206, 207, 210, 217, 218, 219, 239, 240, 254
Architektura sterowana modelami, 73
Artefakt, 67
Atrybut, 62

B

Biblioteki, IV, 205, 206, 207, 210, 221

C

C#, VII, 9, 11, 12, 13, 35, 38, 64, 65, 66, 68, 70, 71, 72, 74, 75, 101, 113, 118, 171, 172, 174, 203, 206, 224
całość-część, 24
całości, 3, 6, 24, 25, 26, 140, 195, 227
części, 5, 12, 24, 25, 35, 52, 59, 61, 78, 79, 115, 137, 138, 141, 154, 195, 245

D

Definicja klasy, 14, 15, 219
Dekomponowanie, 137
delegat, 115
Delegowanie, III, 77, 113, 114, 117, 128, 129
Destruktor, 11, 26, 129
destruktor wirtualny, 86, 87, 89, 90
diagram komponentów, 137
Diagramy sekwencji, 1, 44, 66
Dziedziczenie, III, 15, 50, 51, 77, 78, 100, 101, 103, 113, 132, 133, 158

E

element składowy, 84
Enkapsulacja, 9

F

Finalizator, 11
Funkcja składowa, 107
Funkcje wirtualne, III, 77, 85

G

Generalizacja, 1, 14
generowanie wyjątku, 89
GNU/Linux, IV, VIII, 205, 206, 218, 221
GUI, IV, 235, 236, 237, 238, 239, 240, 243, 244, 245, 246, 247, 252, 255
`GUID`, 121, 122, 208, 250, 251

I

ikona klasy, 7, 9
Interfejs, 34, 118, 121, 145, 160, 182, 189, 202, 236
interfejs programistyczny, 163
interfejsy C++, 35, 119
Iterator, IV, 11, 139, 141, 199, 200
Iteratory, 200, 201

J

Java, VII, 9, 11, 35, 65, 66, 71, 75, 101, 118, 138, 171, 172, 174, 203, 236
język, III, VII, VIII, 1, 3, 5, 54, 55, 58, 73, 74, 75, 136, 138, 195, 236

K

Klasa, 1, 6, 12, 14, 16, 19, 20, 26, 32, 33, 34, 38, 40, 61, 62, 95, 97, 111, 120, 122, 132, 133, 153, 160, 165, 167, 170, 180, 192, 193, 195, 200, 201, 240, 245
Klasy abstrakcyjne, 1, 13, 118
Klasy finalne, 15
Klasy kontenerowe, 28
komponent, 34, 48, 136, 137, 237, 251
kompozycja, 25, 136, 138
Konceptualizacja, 60
Konstruktor, 10, 26, 93, 129

L

łańcuch, 54, 194
liczebność, 9, 10, 28, 30
licznik, 111, 121, 122, 126

M

Mechanizm, III, 77, 78, 95, 113, 148, 166, 218
Metametamodel, 74
Metamodel, 57, 73
Metoda, IV, 7, 62, 115, 124, 126, 139, 141, 150, 151, 173, 174, 175
metoda obiektu, 113
metody abstrakcyjne, 13
Metodyka, 3, 61
Model, 2, 5, 21, 51, 57, 60, 66, 72, 253, 254
model dziedzinowy, 72
Modelowanie, IV, 2, 15, 235, 246
Modelowanie systemów, 2
Mutator, 11

N

nagłówek, 68, 70
niezmienniki, 65
nową wersję, 127, 128
nowego komponentu, 126

O

Obiekt, 26, 38, 39, 46, 92, 114, 133, 141, 166, 167, 173, 177, 192, 201
Obiekty funkcyjne, III, 77, 92, 114
obiekty klas, 38, 129, 172, 177, 188, 222, 251
Object Pascal, VII, 11, 35, 65, 66, 71, 101, 118, 174
obsługa, IV, 210, 223, 224
Okna, 237, 240
Operacja statyczna, 12
operator, 11, 41, 92, 93, 107, 109, 114, 115, 126

P

Pakiet, VIII
perspektywa implementacyjna, 5
perspektywa procesowa, 5
perspektywa projektowa, 5
perspektywa przypadków użycia, 5
perspektywa wdrożeniowa, 5
plik implementacyjny, 68, 70
Polimorfizm, 98, 116
Powiązanie, 1, 19, 22, 24
procedura obsługi, 225
Proces projektowania, 58
proces wytwarzania, 76
Programowanie, III, VII, 26, 116, 117, 132, 138, 203, 254
projekt klas, 61
Propagacja, 229
Przebieg, 51
Przypadki użycia, 1, 50, 60

R

realizacja, 34, 117, 134
RS 232C, 101, 145, 254
Run-Time Type Information, 107

S

schemat, 50, 72, 167
Składowa chroniona, 64
Składowa prywatna, 63
Składowa publiczna, 65
składowe klasy, 8, 83, 99
stereotyp, 36, 55
struktury, 11, 12, 34, 35, 44, 57, 61, 70, 75, 92, 122, 128, 138, 140, 160, 170, 174, 237, 240
Symbol, 43
System komputerowy, 3

T

Transfer, 101
typ danych, 6, 132
typ funkcji, 109
typy wyliczeniowe, 1, 35

U

Unified Modeling Language, 2, 253, 254
USB, 101, 145, 208, 250, 254
użytkownik, 11, 22, 50, 66, 143, 156, 163, 174

V

Virtual Method Table, 92

W

warunki końcowe, 65, 66

warunki początkowe, 65
widgety, 237, 239
Wielodziedziczenie, 101, 103
Windows, IV, VIII, 95, 122, 126, 156, 163, 164, 165, 205, 206, 207, 208, 210, 211, 212, 214, 218, 219, 239, 251, 254
Własność, 12
wskaźnik, 17, 19, 79, 80, 82, 83, 84, 85, 88, 89, 97, 107, 109, 114, 116, 119, 126, 160, 181, 193, 201, 228, 232
Wyjątki, 224, 231, 233
wywołanie polimorficzne, 188
wzorce analityczne, 140
wzorce architektoniczne, 140
wzorce konstrukcyjne, 140
wzorce operacyjne, 140
wzorce programowania, 140
wzorce projektowe, VII, 140, 141, 203
wzorce strukturalne, 140
wzorzec, 6, 95, 140, 141, 143, 144, 148, 151, 158, 170, 171, 172, 173, 174, 177, 180, 186, 192, 199, 201, 202
Wzorzec klasy, 36

Z

zakres dostępu, 63
Zarządca, 141, 237, 240
Zdarzenia, 237
Zunifikowany język modelowania, 3, 5