UNIWERSYTET EKONOMICZNY W KATOWICACH
WYDZIAŁ INFORMATYKI I KOMUNIKACJI

INFORMATYKA I EKONOMETRIA

ADAM CHRZĄSTEK

REALIZACJA PARADYGMATU PROGRAMOWANIA OBIEKTOWEGO NA PRZYKŁADZIE WYBRANYCH JĘZYKÓW OBIEKTOWYCH

IMPLEMENTATION OF OBJECT-ORIENTED PARADIGM BASED ON CHOSEN OBJECT-ORIENTED PROGRAMMING LANGUAGES

Praca dyplomowa
napisana w Katedrze Informatyki
pod kierunkiem dr Artura Strzeleckiego

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem
i stwierdzam, że spełnia wymogi stawiane pracom dyplomowym.

…….. ……
(data) (podpis promotora pracy dyplomowej)

KATOWICE 2016

SPIS TREŚCI

WSTĘP

Obserwowany na przestrzeni ostatnich kilkudziesięciu lat, gwałtowny i niespodziewanie prędki rozwój technologii informacyjnej na tym świecie, doprowadził nie tylko do automatyzacji wielu dziedzin życia czy upowszechnienia dostępu do wiedzy, ale także spowodował niespotykane dotąd różnice pokoleniowe oraz rozwój nowych chorób cywilizacyjnych. Obsługa urządzeń cyfrowych zapewniających wszystkie te możliwości wpisuje się dziś w podstawowe umiejętności każdego człowieka podążającego za ewolucją technologiczną naszego społeczeństwa.

Na przełomie lat sześćdziesiątych i siedemdziesiątych ubiegłego wieku za bezpośrednią przyczyną wstąpienia technologii informacyjnej na wyższy szczebel rozwoju, pojawiła się dziedzina nauki zwana **inżynierią oprogramowania**. Jej rola do dziś polega na odpowiadaniu na potrzeby użytkowników sprzętu komputerowego, poprzez produkcję oprogramowania sterującego dla tych maszyn. Produktywność i możliwości programistów od lat nie idą w parze z rynkowym zapotrzebowaniem na oprogramowanie, którego wykładniczy przyrost w 1981r. zauważył B. W. Boehm.

Przez te kilkadziesiąt lat obecności sztucznie inteligentnych maszyn cyfrowych w życiu ludzi, tworzenie i rozwijanie oprogramowania przebyło długą i burzliwą drogę ewolucji do obecnej formy, a porównując jej stan początkowy z istniejącym, co poniektórzy z nas byli świadkami narodzin nowego porządku świata. Do dziś, jak grzyby po deszczu, wyrastają nam nowe technologie, konwencje, podejścia i paradygmaty wytwarzania oprogramowania, co utrzymuje szybkie tempo rozwoju tej dziedziny. Profesjonalne programowanie komputerów wymaga rozległej wiedzy technicznej, odpowiedniej dozy kreatywności oraz logicznego myślenia i twórczego rozwiązywania problemów.

Wśród wielu technik programowania na przełomie lat narodziło się mnóstwo różnych systemów i warsztatów temu służących. Pośród niezliczonej ilości języków programowania istnieje tak samo wiele środowisk programistycznych, oraz utartych standardów, których nie sposób wymienić wszystkich w jednej pracy naukowej. Z tego powodu to opracowanie skupia się na dominującym dziś **paradygmacie programowania obiektowego**, który dostarcza możliwość realnego odzwierciedlania świata rzeczywistego w pamięci komputera.

Pierwszym celem poznawczym tego opracowania jest przedstawienie czytelnikowi podstawowych zagadnień z zakresu programowania komputerów, a następnie przybliżenie zasad omawianego tutaj paradygmatu programowania obiektowego. Zaś jako cel metodologiczny obrana została ocena implementacji tego paradygmatu w trzech językach programowania wysokiego poziomu, a wzięte pod szkło zostały **C#, JavaScript** oraz **SmallTalk.**

Pierwszy rozdział pracy realizuje założony cel poznawczy, poprzez wprowadzenie czytelnika w świat programowania. Przybliżone tutaj zostały m.in. pojęcia algorytmu, czyli trzonu każdego programu komputerowego. Kolejno, naturalnym następstwem, opracowanie charakteryzuje wybrane narzędzia tworzenia algorytmów, czyli **języki programowania**, wraz z ich rodzajami. W rozdziale pierwszym ukazano również zarys głównych paradygmatów programowania, konkurujących z głównym tematem tej pracy, czyli paradygmatem obiektowym.

Drugi rozdział tej pracy naukowej odsłania paradygmat obiektowy w pełnej okazałości. Zostały tu opisane jego główne komponenty oraz najważniejsze mechanizmy, przez które rozumie się dziedziczenie, polimorfizm oraz enkapsulację. Wszystkie te zagadnienia znalazły swoją deskrypcję i charakterystykę właśnie w drugim rozdziale.

Ostatnia część opracowania naukowego stanowi konkluzję wysnutą na podstawie zaprezentowanych w poprzedzających rozdziałach informacji teoretycznych. Przedstawiono tutaj sposób implementacji każdej cegiełki obiektowego paradygmatu programowania na przykładzie wybranych języków: C#, JavaScript oraz SmallTalk. W podrozdziale podsumowującym znajduje się obiektywna opinia autora na temat sposobu realizacji tej konwencji wytwarzania aplikacji komputerowych przez poszczególny z wytypowanych języków programowania.

Opracowanie przestrzega ogólnoprzyjętych dobrych praktyk wytwarzania oprogramowania np. poprzez odpowiednie formatowanie kodu w poszczególnym z języków, oraz stosowanie anglojęzycznych nazw używanych w kodzie źródłowym. Przykładowe listingi w rozdziale drugim oparte są na pseudo-kodzie, często wykorzystywanym w celu obrazowania pewnych zjawisk programistycznych, bez angażowania konkretnego języka. Praca została napisana w oparciu głównie o anglojęzyczne i najnowsze pozycje książkowe, z racji krótkich terminów przydatności literatury traktującej o wytwarzaniu oprogramowania. Tekst został zredagowany w programie pakietu Microsoft Office 2010 – Microsoft Word 2010.

1. PROGRAMOWANIE KOMPUTERÓW

Dotychczas, komputery oraz oprogramowanie nimi zarządzające, jest wykorzystywane w wielu różnych dziedzinach życia człowieka: począwszy od kontroli broni i elektrowni jądrowych, a skończywszy na obsłudze prostych mini-gier na platformy mobilne. Z powodu tej znacznej rozbieżności zastosowań, na przestrzeni lat ukształtowało się zarówno wiele różnych języków wytwarzania oprogramowania, jak i konwencji definiujących charakter każdego z nich.

W tym rozdziale przedstawiono czytelnikowi fundamenty programowania komputerów oraz podstawowe prawa rządzące wytwarzaniem aplikacji korzystających z dostarczanej przez sprzęt komputerowy mocy obliczeniowej. Pierwsza część pracy traktuje o wybranych i najważniejszych paradygmatach programowania oraz przybliża cechy kilku języków programowania, reprezentujących różne podejścia do wytwarzania *software'u*. Rozdział zawiera także opis podstaw algorytmów oraz kilka przykładów.

1.1. Czym jest programowanie

Pojęcie programowania zalicza się do tej grupy terminów, których rozległość i mnogość znaczeń powoduje niemałe kłopoty z ustaleniem konkretnej, powszechnie obowiązującej definicji. Nie istnieje jedno słuszne i właściwe określenie idei programowania, a dla różnych osób oznacza ono coś innego. Pomimo znacznych utrudnień związanych z semantyką pojęcia „programowanie" poniżej znajduje się kilka definicji proponowanych przez ekspertów w tej dziedzinie.

Według Adriana i Kathie Kingsley-Hughes programowanie to możliwość rozmowy z komputerem za pomocą języka rozumianego przez obie strony. Dzięki użyciu odpowiedniej składni i gramatyki jesteśmy w stanie przekazać systemowi komputerowemu instrukcje wykonywania pożądanego procesu[1]. Taka metaforyczna definicja na pewno nie przyniesie nikomu problemów ze zrozumieniem. Pokuśmy się jednak na bardziej wyrafinowane określenie programowania zaproponowane przez

[1] A. Kingsley-Hughes, K. Kingsley-Hughes: *Beginning Programming,* Wiley Publishing, Indianapolis 2005, s. 4.

Grzegorza Samołyka: „pod pojęciem programowanie należy rozumieć zespół działań mających na celu rozwiązanie pewnej klasy problemów programistycznych. Działania te składają się na sformalizowany proces tworzenia programu". G. Samołyk wyróżnia także podstawowe czynności składające się na logiczny proces wytwarzania oprogramowania[2]:

- **Projektowanie -** składa się z dwóch głównych faz – zaplanowanie programu poprzez stworzenie szkicu jego ogólnej koncepcji (logika działań, informacje wejściowe i wyjściowe), a następnie skompletowanie dokumentacji technicznej całego projektu.
- **Zapis –** na podstawie sporządzonego wcześniej projektu wybiera się język programowania najlepiej odpowiadający wymaganiom technicznym projektu oraz dedykowany dla niego edytor.
- **Kompilacja i konsolidacja –** pierwsze uruchomienie i eliminacja powstałych błędów i wyjątków w fazie tworzenia.
- **Dystrybucja** – rozprowadzanie programu wraz z utworzoną wcześniej dokumentacją użytkownika.
- **Konserwacja –** utrzymanie programu, które trwa przez cały okres jego życia, wprowadzanie poprawek, ulepszeń oraz usuwanie napotkanych błędów podczas użytkowania.

Podczas rozwoju szkieletu projektu szczególną uwagę należy zwrócić na poprawność doboru technologii i jej możliwości do rozwiązywanego problemu. Powinna ona odpowiadać wymaganiom projektu i być w stanie wykonać każde pojedyncze zadanie zdefiniowane w funkcjonalności tworzonego oprogramowania. Dobrą praktyką jest również sprowadzanie architektury programu do znanych wzorców projektowych oraz sprawdzonych algorytmów.

Testowanie oprogramowania jest nierozerwalnie związane z fazą kompilacji i konsolidacji, choć niektórzy doświadczeni architekci propagują tworzenie testów dla programów jeszcze przed napisaniem pierwszej linijki kodu, jak np. R. C. Martin[3].

[2] G. Samołyk: *Podstawy programowania komputerów dla inżynierów*, Politechnika Lubelska, Lublin 2011, s. 29.

[3] R.C. Martin: *Clean Code. A Handbook of Agile Software Craftsmanship*, Pearson Education, Boston 2015, s. 129 - 131.

Działanie to ma na celu określenie użyteczności nowopowstałego systemu oraz zdemaskowanie nieodpowiednio działających komponentów oprogramowania.

Poprzez pojęcie konserwacji programu należy rozumieć szereg czynności, których wymaga każdy większy system informatyczny. Najważniejsze z nich to świadczenie serwisu gwarancyjnego, wydawanie ulepszeń, łatek, reedycji czy tworzenie nowych wersji całych systemów.

1.2. Algorytmy

Aby jakakolwiek instrukcja przekazana do komputera została wykonana, niezbędny jest swego rodzaju przepis, wzór postępowania, nazywany w informatyce **algorytmem**. Jeśli jakiegoś zadania nie da się opisać za pomocą algorytmu, jest to równoznaczne z tym, że żaden komputer nie jest w stanie wykonać tej czynności. K. Wojtuszkiewicz definiuje algorytm jako ściśle określony sposób postępowania prowadzący do rozwiązania określonej klasy zadań[4]. Natomiast profesjonalna definicja pochodząca z nauki o algorytmach – algorytmiki - brzmi następująco: „algorytm jest metodą rozwiązywania zadań nadającą się do zastosowania w dowolnym komputerze i dowolnym języku programowania. Algorytmy obejmują metody organizacji danych biorących udział w obliczeniach. Obiekty powstałe w ten sposób nazywane są **strukturami danych**[5]”.

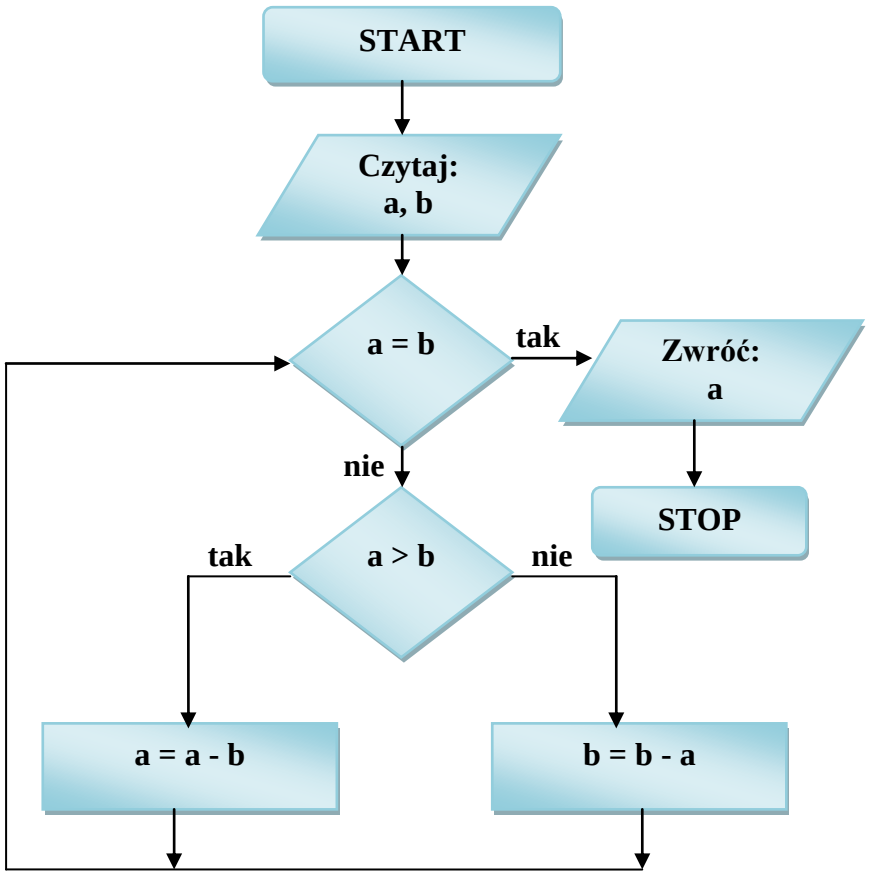
Odpowiednio skonstruowane algorytmy wyróżniają się czterema charakterystycznymi cechami:

- Jeśli algorytm obsługuje dane wejściowe, są one konkretnie zdefiniowane.
- Wynikiem działania algorytmu są zawsze dane wyjściowe.
- Algorytm jest precyzyjnie opisany i podzielony na działania jednostkowe.
- Wynik algorytmu (dane wyjściowe) jest uzyskiwany możliwie najkrótszą drogą oraz przy zredukowaniu do minimum wykorzystania mocy obliczeniowej sprzętu komputerowego.

[4] K. Wojtuszkiewicz: *Programowanie strukturalne i obiektowe*, Wydawnictwo Naukowe PWN, Warszawa 2009, s. 12.

[5] R. Sedgewick: *Algorytmy w C++*, Wydawnictwo Helion, Gliwice 2012, s. 23.

Rysunek 1-1 przedstawia popularny algorytm Euklidesa, który służy do wyznaczania największego wspólnego dzielnika dwóch liczb naturalnych.



Rysunek 1.1: Algorytm Euklidesa.
Źródło: Opracowanie własne.

Po odczytaniu liczb a i b rozpoczyna się pętla działań, która zostanie przerwana, gdy w wyniku działania algorytmu liczby a i b będą sobie równe. Wewnątrz pętli wykonywane są operacje na wprowadzonych danych: pierwsza liczba jest liczbą mniejszą, natomiast druga to różnica między liczbą większą i mniejszą. Proces ten jest powtarzany do chwili uzyskania dwóch równych sobie liczb, które to są zwracane jako największy wspólny dzielnik danych wejściowych. Przy testach działania powyższego algorytmu można dostrzec jego słaby punkt: wykonywanie wielu operacji odejmowania (iteracji pętli), kiedy wprowadzone zostaną dwie liczby, których różnica jest względnie wysoka.

1.3. Paradygmaty programowania

Termin paradygmatu programowania powstał w celu klasyfikacji języków programowania ze względu na styl wytwarzania oprogramowania jaki one reprezentują. Zbadanie najważniejszych cech języków programowania, pozwala określić ich przynależność do konkretnego paradygmatu, przy czym istnieje wiele języków, które kwalifikują się w ramy wielu paradygmatów – nazywamy je językami *wieloparadygmatowymi.* Paradygmaty różnią się miedzy sobą kryteriami, względem których wpisuje się w nie badane języki programowania: niektóre honorują jedynie sposób wykonywania poleceń przez język pod maską, a inne skupiają się na sposobie organizacji kodu źródłowego, czy składni i gramatyki języka.

1.3.1. Paradygmat obiektowy

Obiektowy paradygmat programowania, jak sama nazwa wskazuje, opiera się o koncepcję obiektów, które zawierają dane w postaci pól (czasem nazywanych atrybutami). Oprócz tego, mocno charakterystyczną cechą programowania obiektowego są również bloki kodu zwane metodami, które to umożliwiają dostęp do pól obiektów, oraz pozwalają je modyfikować, a tym samym wpływać na stan obiektu. Wyczerpujący i szczegółowy opis paradygmatu obiektowego znajduje się w drugim rozdziale tego opracowania.

1.3.2. Paradygmat deklaratywny i funkcyjny

Paradygmat deklaratywny to styl budowania i projektowania elementów programów komputerowych w taki sposób, aby wyrażały logikę obliczeń bez ujawniania sterowania przepływu wewnątrz elementów. Wiele języków programowania wpisujących się w paradygmat deklaratywny nie obsługuje lub minimalizuje możliwość wprowadzania do programu skutków ubocznych dla wywołań instrukcji. Deklaratywny styl wytwarzania oprogramowania nie definiuje *jak* program ma osiągnąć zamierzony skutek – skupia się na opisie *co* aplikacja ma zdziałać[6]. Charakterystykę tą rozumie się jako główną różnicę pomiędzy programowaniem deklaratywnym, a imperatywnym, które to jawnie definiuje algorytmy osiągające zamierzony cel. Jednym

[6] M. Gabbrielli, S. Martini: *Programming Lanuguages: Principles nad Paradigms,* Springer, Nowy Jork 2009, s. 62.

z podparadygmatów programowania deklaratywnego, którym warto się zająć jest programowanie funkcyjne.

Paradygmat funkcyjny realizuje się poprzez traktowanie wszystkich zawartych w kodzie źródłowym algorytmów tak, jakby były równaniami lub funkcjami matematycznymi. Oprócz tej najważniejszej cechy, w programowaniu funkcyjnym rzuca się w oczy także brak jakichkolwiek zmiennych wartości. Kolejnym wyznacznikiem funkcyjnego stylu programowania jest wykonywanie poleceń systemowych poprzez wyrażenia, a nie tak jak np. dzieje się to w programowaniu obiektowym – przez deklaracje. Pojawienie się funkcyjnego paradygmatu programowania jest ściśle związane z *rachunkiem lambda* – formalnym systemem służącym opisowi zagadnień matematycznych. Skutkiem pochodzenia omawianego paradygmatu jest możliwość przedstawienia zapisu wielu języków funkcyjnych jako elaboratu rachunku lambda[7].

Spośród wielu różnych definicji programowania funkcyjnego, warto przytoczyć tę, zaproponowaną przez Joshue Backfield'a. Według specjalisty, paradygmat funkcyjny charakteryzuje się następującymi cechami[8]:

- **Typy funkcyjne (*first-class functions*) –** typy, które umożliwiają przekazywanie funkcji jako argumentu lub zwracanie funkcji. Procedura ta znajduje szczególne zastosowanie w wielokrotnym wykorzystaniu kodu oraz przy operacjach na abstrakcjach kodu.
- **Funkcje czyste (*pure functions*) –** funkcje, które nie generują żadnych skutków ubocznych (*side effects*). Przez skutki uboczne rozumie się działania, które mogą zostać wywołane przez konkretną funkcję, a które nie realizują głównego celu tejże funkcji. Przykładem może być wywołanie innej czynności w obrębie funkcji, lub dekrementacja/inkrementacja zmiennej wprowadzonej do funkcji.
- **Rekurencja *(recursion)* –** rekurencja służy konstruowaniu zwięzłych algorytmów, które działają jedynie w oparciu o wprowadzone dane. Dzięki temu rola takiego algorytmu sprowadza się jedynie do przeprowadzenia iteracji i kontynuowania wykonywania do określonego warunku.

[7] R. W. Sebesta (2012): *Concept of Programming Lanuguages 10th Edition,* Pearson Education, New Jersey 2012, s. 73.

[8] J. Backfield: *Becoming Functional,* O'Reilly Media, Sebastopol CA 2014, s. 16.

- **Zmienne niemutowalne *(immutable variables)*** – zmienne, których wartość nie może zostać przekształcona, po jej zadeklarowaniu.
- **Ewaluacja nierygorystyczna *(nonstrict evaluations)*** – cecha programowania funkcyjnego, która pozwala na przetrzymywanie w strukturze programu niezainicjowanych zmiennych (bez wartości), do momentu wystąpienia pierwszego odwołania do nich.

Pomimo tego, że czysto-funkcyjne języki programowania cieszą się największą popularnością i sympatią wśród społeczności akademickich i naukowych, które wykorzystują je do rozwiązywania problemów matematycznych, znalazły zastosowanie także w komercyjnym wytwarzaniu oprogramowania. Języki takie jak OCaml, Haskell, Scala czy F# były i nadal są wykorzystywane przez prominentne firmy z branży informatycznej, m.in.: Nortel, Facebook, Ericsson, Apple.

1.3.3. Paradygmat imperatywny i strukturalny

Najważniejszym zadaniem paradygmatu strukturalnego jest wyniesienie przejrzystości, jakości i czasu wykonywania programów na wyższy poziom, głównie za pomocą instrukcji blokowych, podprogramów czy pętli. Koncepcja ta wyklucza używane przed rozpowszechnieniem programowania strukturalnego popularne instrukcje skoku (np. *goto*), których nadużywanie czyni z kodu źródłowego wysoce nieczytelny, podatny na błędy, skomplikowany i trudny w utrzymaniu twór.

Pierwsze ślady paradygmatu programowania strukturalnego przypadają na dodanie możliwości organizacji kodu źródłowego w schematy blokowe w językach ALGOL 58 i ALGOL 60 pod koniec lat pięćdziesiątych dwudziestego wieku. Początkowo styl ten był wykorzystywany do różnych eksperymentów programistycznych, głównie przez społeczności naukowe i entuzjastów. Z czasem gdy specjaliści opracowali dobre praktyki wykorzystywania programowania strukturalnego, znalazło one zastosowanie także w biznesie, również za sprawą otwartego listu Edsgera W. Dijkstra pt. „*Go To Statement Considered Harmful*”, w którym to duński informatyk ostro skrytykował

instrukcje *goto* i jako pierwszy wypracował wstępną definicję programowania strukturalnego[9].

1.1.1.1. Elementy

W programowaniu strukturalnym, wszystkie programy składają się z następujących elementów[10]:

- **Przepływ sterowania –** według głównego założenia programowania strukturalnego, każdy program jest wypadkową zespolenia ze sobą instrukcji przepływu sterowania. Wyróżniamy następujące:
  - **Sekwencja –** dzięki tej składowej, określone instrukcje lub podprogramy wykonywane są w zadanej kolejności.
  - **Selekcja –** jedna lub więcej instrukcji jest wykonywana jeśli program znajduje w określonym przez selekcję stanie.
  - **Iteracja –** instrukcja lub zbiór instrukcji jest wykonywane dopóty, dopóki nie zostaną spełnione określone warunki.
  - **Rekurencja –** podobna zasada działania do iteracji (wykonywanie instrukcji do określonego warunku), jednak rekurencja może być bardziej efektywna przy obliczeniach matematycznych, z drugiej zaś strony nieumiejętne posługiwanie się rekurencją może doprowadzić do szybkiego przepełnienia pamięci programu.
- **Podprogramy –** przez podprogramy należy rozumieć instrukcje, funkcje, metody i procedury, które mogą zostać wywołane przez jedną instrukcję w wielu miejscach kodu źródłowego.
- **Bloki instrukcji –** umożliwiają grupowanie instrukcji i traktowanie i ich tak, jakby były jedną instrukcją.

Programowanie strukturalne jest podzbiorem paradygmatu imperatywnego. Cechy programowania imperatywnego dają się odczuć w samym paradygmacie strukturalnym, warto jedynie wspomnieć, że programowanie imperatywne wymusza logiczną strukturę

[9] S. Kedar: *Programming Paradigms And Methodology*, Technical Publications, Maharashtra 2008, s. 33.

[10] R. W. Sebesta (2012): *Concept of Programming Lanuguages 10th Edition*, Pearson Education, New Jersey 2012, s. 104.

kodu źródłowego, która ułatwia jego czytanie, zrozumienie oraz ponowne wykorzystanie.

1.3.4. Wieloparadygmatowość

Pośród wielu paradygmatów programowania, także tych niewymienionych w tym opracowaniu, istnieją języki programowania łączące w sobie cechy wielu styli. Ich powstanie z pewnością łączy się z problemami przy doborze odpowiedniego paradygmatu do rozwiązywanego problemu programistycznego – z tego wynika, że języki wpisujące się w więcej niż jeden paradygmat są bardziej uniwersalne i znajdują zastosowanie w większej ilości rozwiązań.

Doskonałym przykładem języka wieloparadygmatowego jest język spod szyldu firmy Microsoft – C#, który został omówiony w rozdziale 1.5.4. Realizuje on przede wszystkim podejście obiektowe, ale oprócz tego da się w nim zauważyć konstrukcje typowe dla programowania funkcyjnego jak np. technologia LINQ, delegaty, czy wyrażenia lambda. Oprócz C#, w podobny sposób wiele paradygmatów realizują też języki takie jak Java, Scala, C++ czy Python.

Obecnie powoli zanika używanie języków ściśle realizujących tylko jeden wybrany paradygmat programowania. Najpotężniejsze i najbardziej efektywne języki tworzenia oprogramowania łączą w sobie najlepsze wybrane cechy spośród różnych koncepcji programowania. Dowodem jest Tabela 1 przedstawiająca średnią pozycję wśród najbardziej popularnych języków programowania w wybranych latach.

Tabela 1.1: Najpopularniejsze języki programowania w wybranych latach.
Źródło: Opracowanie własne na podstawie: http://tiobe.com.

Język programowania	2016	2011	2006	2001
Java	1	1	1	3
C	2	2	2	1
C++	3	3	3	2
C#	4	5	6	10
Python	5	6	7	25
PHP	6	4	4	22
Visual Basic	7	189	-	-

JavaScript	8	9	9	7
Perl	9	7	5	4
Objective-C	10	8	42	-

Powyższe zestawienie idealnie obrazuje długotrwałą dominację wieloparadygmatowych języków programowania od wielu lat. Spośród dziesięciu najbardziej popularnych języków w latach 2016, 2011, 2006 i 2001 jedynie trzy ostatnie, czyli JavaScript, Perl i Objective-C nie wykazują cech wielu styli programowania. Tabela 1 pokazuje również ciekawą ewolucję popularności poszczególnych języków, na jej podstawie można np. przypuszczać, że w przyszłości język C# prawdopodobnie będzie w ścisłej czołówce popularnych języków programowania.

1.4. Języki programowania

Tchnienie wirtualnego życia w surowy algorytm, a tym samym utworzenie z niego rzeczywistego i wykonywalnego programu wymaga przesłania do procesora komputera odpowiednich instrukcji. Zadanie to stawia dewelopera przed nie lada wyzwaniem: język rozumiany przez procesor (zapis maszynowy) jest bynajmniej bardzo kłopotliwy do odczytania przez człowieka. Mostem na tej przepaści między ludźmi i procesorami są **języki programowania,** skonstruowane i zaprojektowane specjalnie na potrzeby tworzenia przejrzystych i klarownych instrukcji procesora. Zapis programu w języku programowania nazywa się **kodem źródłowym** i każdy taki kod musi zostać przetłumaczony na notację rozumianą przez procesor wykonujący. Działanie to określanie jest mianem **kompilacji** i jest wykonywane przez narzędzie zwane **kompilatorem.**

Język programowania to ściśle ustalony zbiór instrukcji, wyrażeń, symboli i operatorów z niezwykle sztywnymi zasadami i normami ich stosowania. W świecie komputerów istnieje wiele języków programowania, których różnice wynikają bezpośrednio z celu ich zastosowania, jednak niektóre posiadają ze sobą cechy wspólne. Najważniejsza właściwość, która łączy wszystkie istniejące języki programowania, polega na ścisłości składni i rygorystycznych zasadach jej stosowania. Tak bardzo zdyscyplinowane i nieugięte reguły używania elementów języków tworzących oprogramowanie, podyktowane są mechaniczną techniką odczytywania instrukcji przez

komputery, które, nie zapominajmy, są tylko bezdusznymi, elektronicznymi maszynami liczącymi. Języki programowania to *narzędzie* wytwarzania oprogramowania, dzięki któremu powstaje kod zrozumiały dla człowieka oraz przetłumaczalny przez kompilator.

1.5. Podział języków programowania

Języki programowania podzielić można na kilka grup, zależnie od branego pod uwagę kryterium. Najważniejszą klasyfikację języków komputerowych rozpoczniemy od prymarnego i najbardziej podstawowego miernika: **poziomu abstrakcji**. Wyróżniamy języki wysokiego i niskiego poziomu[11]. Do tych drugich zalicza się wspomniany już we wcześniejszym podrozdziale **kod maszynowy** oraz język o nazwie **Assembler.**

1.5.1. Języki niskopoziomowe

1. Kod maszynowy

Kod maszynowy to wewnętrzny język procesora i jedyna forma zapisu akceptowana przez maszynę wykonującą. Pozytywne uruchomienie kodu źródłowego napisanego w jakimkolwiek języku programowania wymaga jego przetłumaczenia (kompilacji) na zapis maszynowy. W zapisie maszynowym poszczególne polecenia dla procesora są przekazywane w formie kodów binarnych[12]. Jak wynika z rysunku 1-2 utworzenie lub rozszyfrowanie tego kodu nie należy do najprostszych zadań:

[11] G. Coldwind: *Zrozumieć programowanie,* Wydawnictwo Naukowe PWN, Warszawa 2005, s. 50.

[12] G. Coldwind: *Zrozumieć programowanie,* Wydawnictwo Naukowe PWN, Warszawa 2005, s. 57.

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Rysunek 1.2: Fragment kodu w języku maszynowym.
Źródło: Opracowanie własne na podstawie: http://stackoverflow.com

Bardzo rzadko zdarza się, by programiści tworzyli programy bezpośrednio używając zapisu maszynowego – jeśli jednak zachodzi taka potrzeba, używa się wtedy specjalnych edytorów, które odzwierciedlają kod binarny w zapisie szesnastkowym.

2. Assembler

Assembler to ostatni z dwóch języków programowania niskiego poziomu, który w przeciwieństwie do zapisu maszynowego jest wykorzystywany bezpośrednio przez deweloperów. Jego składnia w porównaniu do kodu maszynowego jest łatwa do opanowania przez człowieka, oraz w porównaniu do języków wysokiego poziomu, zawiera w sobie szczyptę intuicyjności. Poszczególne instrukcje w Assemblerze odpowiadają konkretnym rozkazom wyrażonym za pomocą kodu binarnego. Rysunek 1-3 obrazuje prezencję kodu źródłowego Assemblera dla programu, który wyświetla na ekranie komputera napis „Hello World!!!”.

```
01 DATA SEGMENT
02       MESSAGE DB "HELLO WORLD!"
03 ENDS
04
05 CODE SEGMENT
06     ASSUME DS:DATA CS:CODE
07 START:
08       MOV AX,DATA
09       MOV DS,AX
10       LEA DX,MESSAGE
11       MOV AH,9
12       INT 21H
13       MOV AH,4CH
14       INT 21H
15 ENDS
16 END START
17
```

Rysunek 1.3: Program wyświetlający napis w języku Assembler.
Źródło: Opracowanie własne.

Na pierwszy rzut oka można dostrzec, że dostarczana składnia przez biblioteki Assemblera jest dużo bogatsza niż ta służąca za zapis kodu maszynowego. Assembler zawiera w sobie predefiniowane elementy: zbiór znaków alfanumerycznych (ASCII), słowa kluczowe, stałe, nazwy symboli oraz atrybuty[13].

Efektywne i poprawne posługiwanie się językami programowania niskiego poziomu wymaga od programisty rozległej wiedzy na temat architektury systemów komputerowych, oraz sposobu działania procesora wykonującego. Tworzenie oprogramowania za pomocą opisanego wyżej języka Assembler (zapisu maszynowego także) jest zadaniem niezwykle żmudnym, podatnym na błędy, a oprócz tego wymaga czasu, wiedzy i cierpliwości. Pomimo tych licznych utrudnień, języki programowania niskiego poziomu znajdują swoje zastosowanie i świetnie się sprawdzają w zadaniach, gdzie optymalizacja szybkości działania programu i minimalizacja wykorzystania mocy obliczeniowej komputera są priorytetem.

1.5.2. Języki programowania wysokiego poziomu

W myśl sentencji „*Potrzeba matką wynalazków*", konieczność usprawnienia i ułatwienia procesu tworzenia kodu źródłowego, a także liczne problemy związane z używaniem niskopoziomowych języków programowania spowodowały powstanie **języków wysokopoziomowych.** Także ekspresowe tempo rozwoju komputerów i szeroko pojętej informatyki przyczynilo się do powstawania oprogramowania na niespotykaną dotąd skalę, co z kolei jeszcze bardziej spotęgowało potrzebę zautomatyzowania rozwoju ówczesnych aplikacji. W procesie projektowania języków wysokiego poziomu za cel obrano jak najbliższe upodobnienie kodu źródłowego do języka, którym komunikuje się człowiek. Niestety, owa cecha powoduje znaczne zmniejszenie szybkości wykonywania programów napisanych w językach wysokiego poziomu w porównaniu do Assemblera i języka maszynowego. Spowodowane jest to koniecznością translacji całego kodu źródłowego do kodu maszynowego, aby program mógł zostać wykonany przez procesor.

Obecnie istnieje wiele języków programowania mniej i bardziej popularnych oraz mniej i bardziej skomplikowanych. Różnią się one między sobą przede wszystkim składnią, sposobem działania, przeznaczeniem i realizowanym paradygmatem. Spośród

[13] V. Pirogow: *Asembler. Podręcznik programisty,* Wydawnictwo Helion, Gliwice 2005, s. 37.

istniejących kilku tysięcy różnych języków programowania wysokiego poziomu, agencja TIOBE wyróżnia 20 najbardziej rozpowszechnionych (stan na marzec 2016 r.)[14]:

1. Java
2. C
3. C++
4. C#
5. Python
6. PHP
7. Visual Basic .NET
8. JavaScript
9. Perl
10. Ruby
11. Delphi/Object Pascal
12. Visual Basic
13. Swift
14. Objective-C
15. R
16. Groovy
17. MATLAB
18. PL/SQL
19. D
20. SAS

1.5.3. Java

Pierwszy język programowania, który zostanie wzięty pod lupę to **Java**. Java została stworzona oraz jest obecnie rozwijana przez firmę **Sun Microsystems** pod kierownictwem Jamesa Goslinga. Charakterystyczną cechą Javy jest kompilacja do kodu bajtowego, który to jest wykonywany przez maszynę wirtualną. Powszechna praktyka jawnych zapożyczeń koncepcji z istniejących języków podczas tworzenia nowych, znalazła zastosowanie również przy projektowaniu Javy: uruchamianie

[14] TIOBE Software BV, www.tiobe.com

na maszynie wirtualnej oraz zarządzanie pamięcią zostały zapożyczone z języka programowania Smalltalk, natomiast liczne słowa kluczowe oraz składania pochodzą z C++[15].

Twórcy Javy zwracają uwagę na jej najbardziej charakterystyczne cechy[16]:

1. **Realizacja paradygmatu obiektowego**

Java doskonale wpisuje się w wymagania obiektowości dla języków programowania, a paradygmat ten zostanie szczegółowo opisany w rozdziale drugim tej pracy.

2. **Dziedziczenie**

Każdy deklarowany obiekt w Javie, dziedziczy z najważniejszej bazowej klasy Object. Dzięki temu wszystkie obiekty zawierają predefiniowane, podstawowe i niezbędne funkcje zapewniające różnorakie operacje: kopiowanie, porównywanie, przenoszenie czy niszczenie. Choć twórcy Javy ograniczyli możliwość dziedziczenia tylko do jednej klasy, pozornie utraconą elastyczność rekompensują zapewnione **interfejsy,** które to zawierają zbiór określonych funkcji przypisywanych do obiektu.

3. **Programowanie rozproszone i sieciowość**

Szereg dedykowanych bibliotek Javy, gwarantuje dostęp do funkcji umożlwiających programowanie rozproszone. Dzięki temu widżetowi, programista ma możliwość integracji ze sobą odrębnych aplikacji, które nawet nie muszą być napisane w Javie. Wśród tych bibliotek znajdziemy również takie, które dają możliwość tworzenia programów javowych pracujących na serwerach lub uruchamiających się w przeglądarkach internetowych.

4. **Bezpieczeństwo i sprawność**

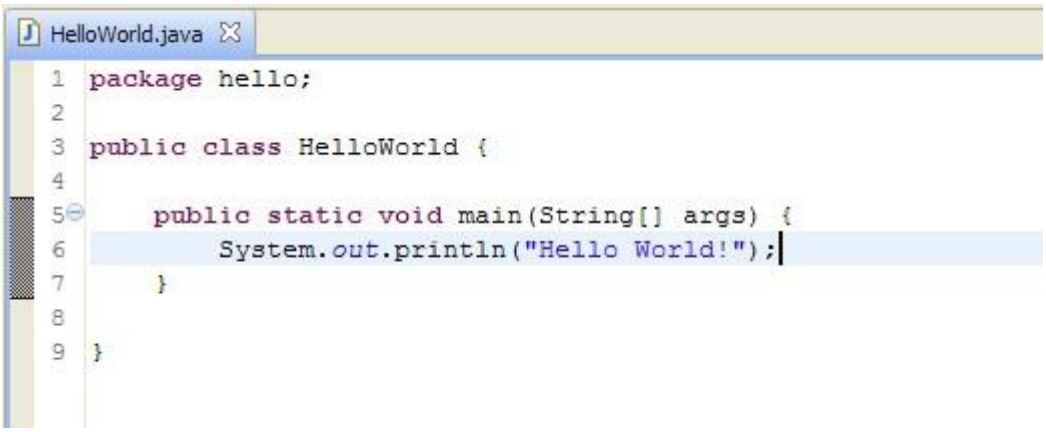
W celu realizacji swojego pomysłu na stworzenie bezpiecznego i niezawodnego języka, twórcy Javy wprowadzili do języka obowiązkową obsługę wyjątków opcjonalnie pojawiających się w czasie wykonywania programu. W związku z tym,

[15] C. S. Horstmann, G. Cornell: *Core Java. Volume I – Fundamentals, Ninth Edition*, Prentice Hall, New Jersey 2013, s. 42 - 43.

[16] B. Eckel: *Thinking in Java, Fourth Edition*, Prentice Hall, New Jersey 2006, s. 65 - 73.

programista jest zmuszony do używania bloków „try {…} catch{…}” w celu przechwytywania niechcianych danych czy operacji. Poza koniecznością deklarowania bloków kodu zarządzających wyjątkami, Java dostarcza jeszcze dwie funkcjonalności, które wynoszą bezpieczeństwo i sprawność kodu Java na jeszcze wyższy poziom. **Asercje** zapewniają programiście możliwość podglądu wartości elementu w kodzie w czasie działania programu, np. w celu upewnienia się czy dzielnik w wyrażeniu na pewno jest różny od 0. Innym zastosowaniem asercji może być sytuacja, kiedy administrator kodu jest świadom błędu w pewnym miejscu. Wtedy, dzięki asercjom może rozkazać kompilatorowi pominięcie wadliwej części. Mechanizm **logowania** natomiast, zapisuje do plików tekstowych przebieg działania programu. W wypadku problemów z lokalizacją błędu w kodzie źródłowym, programista ma możliwość prześledzić szczegółowe opisy czynności programu w plikach logujących.

Grafika numer 1-4 przedstawia program wyświetlający napis „Hello World!”:



Rysunek 1.4: Program "Hello World" w języku Java.
Źródło: Opracowanie własne.

Wraz z rozwojem technologii i komputerów, z Javy wyrosło kilka platform programistycznych, które realizują różne cele programistyczne. Na dzień dzisiejszy wyróżniamy 3[17]:

- **Java Platform, Standard Edition (Java SE)** – korzeń całej filozofii znanej pod nazwą “Java”. Opisuje podstawową strukturę tworzenia oprogramowania w języku Java, które uruchamia się na komputerach stacjonarnych oraz zdalnych serwerach.

[17] C. S. Horstmann, G. Cornell: *Core Java. Volume I – Fundamentals, Ninth Edition*, Prentice Hall, New Jersey 2013, s. 53.

- **Java Platform, Enterprise Edition (Java EE)** – serwerowa platforma programistyczna wywodząca się z Javy SE. Oferuje dostęp do zdefiniowanego wzorca tworzenia aplikacji opartych w wielowarstwową architekturę komponentową. Platforma ta ściśle określa jakie funkcjonalności (Application Programming Interface) serwer powinien udostępniać na zewnątrz, aby mógł sprawnie obsługiwać platformę Java Enterprise Edition.
- **Java Platform, Micro Edition (Java ME) –** okrojona wersja podstawowej platformy programistycznej Javy, utworzona z myślą o tworzeniu aplikacji przeznaczonych dla urządzeń o znacznie ograniczonej mocy obliczeniowej, takich jak urządzenia mobilne (smartphony, tablety).

1.5.4. C# i .NET Framework

Język C# został zaprojektowany jako język obiektowy przez Kanadyjczyka Andersa Hejlsberga na zlecenie firmy Microsoft w 2000 r., a jego korzenie sięgają grupy języków C, która odegrała znaczącą rolę w rozwoju tworzenia oprogramowania. C# ściśle współpracuje z platformą programistyczną **.NET Framework** firmy Microsoft, której to zasada działania polega na kompilowaniu całego kodu źródłowego do pośredniego zapisu o nazwie **Common Inegrated Language** (CIL), wykonywanego we wspólnym środowisku uruchomieniowym **Common Language Runtime** (CLR). Oprócz kompilacji kodu źródłowego C# do kodu CIL, pakiet .NET Framework potrafi robić to dla około 40 różnych języków programowania, np. Visual Basic .NET, F#, Python, COBOL; co w pełni uzasadnia zastosowanie takiego rozwiązania. W związku z tym, niemożliwe jest uruchomienie programu skompilowanego do pośredniczącego kodu CIL na maszynie, która nie posiada zainstalowanego pakietu .NET Framework w odpowiedniej wersji[18]. Poniższa grafika prezentuje wygląd programu wyświetlającego napis „Hello” skompilowanego do kodu Common Integrated Language.

[18] A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework,* SSBM Finance Inc., New York 2015, s. 6 – 32.

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       13 (0xd)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hello"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ret
} // end of method Test::Main
```

Rysunek 1.5: Kod programu wyświetlającego napis "Hello" w formie kodu CIL. Źródło: Opracowanie własne.

Powyższa grafika przedstawia nieskomplikowany program napisany w języku C# i skompilowany do kodu Common Integrated Language. Jak nietrudno zauważyć, zapis CIL nie jest zbyt intuicyjny w użyciu, dlatego środowisko uruchomieniowe automatycznie i samodzielnie sprowadza każdy program platformy .NET do takiej formy. Bardzo rzadko, i tylko w ekstremalnych sytuacjach zdarza się, aby programista w czasie tworzenia oprogramowania był zmuszony do bezpośrednich operacji na kodzie CIL. Pomimo to, warto znać zasadę działania trzonu platformy .NET przy tworzeniu aplikacji z nią współpracujących.

Najważniejsze cechy języka C#

Analizując składnie i sposób budowy programów skonstruowanych w C#, można zauważyć liczne cechy wspólne z takimi językami jak Java czy C++ - niewątpliwie języki te posłużyły dla zespołu Andersa Hejlsberga jako wzór. Wśród charakterystycznych cech można wyróżnić[19]:

- **Automatyczne odśmiecanie pamięci –** usuwaniem niepotrzebnych obiektów i zmiennych zajmuje się środowisko uruchomieniowe CLR (a konkretniej mechanizm zwany ***garbage collector***), dlatego deweloper nie musi się martwić takimi błędami jak wyciek pamięci, czy przeciążenie stosu aplikacji.
- **Obiektowy paradygmat** – podobnie jak w Javie, obiektowy paradygmat języka C# jest realizowany poprzez hierarchiczność elementów. Każdy z nich wywodzi się z klasy bazowej *Object,* co oznacza, że na ten typ możemy rzutować zarówno typy referencyjne jak i wartościowe.

[19] J. Albahari, B. Albahari: *C# 6.0 in a Nutshell,* O'Reilly Media Inc., Sebastopol CA 2015, s. 44 – 49.

- **Biblioteka klas –** bogaty zbiór oficjalnych bibliotek kodu udostępniany przez firmę Microsoft. Zawierają one gotowe typy i funkcje, służące implementacji najważniejszych modułów tworzonych aplikacji. Przykładowe biblioteki: *Windows Presentation Foundation* – biblioteka służąca generowaniu graficznie zaawansowanych interfejsów użytkownika; *ADO.NET* – biblioteka umożliwiająca współpracę z technologiami bazodanowymi; *Windows Communication Foundation* – dostarcza do aplikacji usługi sieciowe; *ASP.NET* – pakiet służący tworzeniu aplikacji internetowych.
- **Dynamiczne generowanie kodu źródłowego –** platforma .NET Framework umożliwia generowanie kodu „w locie", przez co rozumie się produkowanie go w czasie działania programu, na podstawie np. danych wprowadzonych przez użytkownika. Wytworzony kod jest dynamicznie przyłączany do struktury aplikacji.

Poniższy zrzut ekranu prezentuje prosty program języka C#, symulujący rzut sześcienną kostką.

```
using System;

class Dice
{
    static void Main(string[] args)
    {
        int result;
        Random r = new Random();

        Console.WriteLine("Press enter to throw the dice");

        while (true)
        {
            Console.ReadKey();

            result = r.Next(1, 7);

            Console.WriteLine("Result is: {0}", result);
        }
    }
```

Rysunek 1.6: Symulacja sześciennej kości do gry w języku C#.
Źródło: Opracowanie własne.

Z powyższego kodu wynika, że program przechowuje dwa obiekty: zmienną *result*, oraz typ *Random.* Za pomocą obiektu *Random,* który generuje pseudolosowe liczby w zadanym zakresie, w zmiennej *result* umieszczona zostaje wylosowana liczba.

W pętli, której warunek wykonania jest zawsze prawdziwy (pętla nieskończona), program oczekuje na wciśnięcie przez użytkownika przycisku *Enter*, a następnie rzutuje wylosowana liczbe na zmienną *result*. Ostateczna akcja to wyświetlenie użytkownikowi wylosowanej liczby z otwartego przedziału 1 – 6.

1.5.5. SQL

SQL (*Structured Query Language)* to wieloparadygmatowy, strukturalny język zapytań, służący do manipulowania danymi zawartymi w **relacyjnych bazach danych**. Jego struktury umożliwiają m. in. tworzenie baz danych, odpytywanie baz z konkretnych wartości oraz umieszczanie w nich nowych rekordów. SQL to język deklaratywny, co oznacza że odpowiedzialność za sposób przechowywania i zwracania danych leży po stronie systemu zarządzania bazą danych, czyli ***Database Management System (DBMS)***[20]**.**

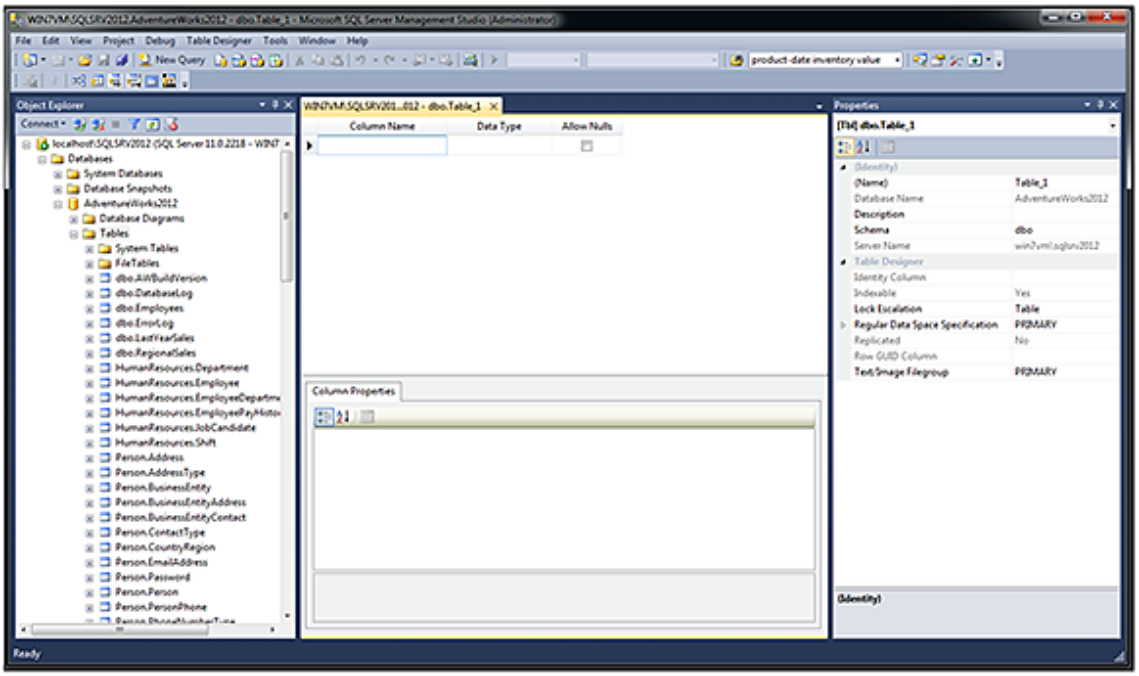
Oprogramowanie i serwery baz danych

Nieodzownym elementem każdego rodzaju bazy danych jest **serwer**, na którym zostaje ona umieszczona. Serwery to swoiste silniki baz, które określa się też mianem „*back-end*" czy *„engine".* Na rynku istnieje wiele różnych silników dedykowanych do hostowania relacyjnych baz danych, począwszy od produktów wielkich marek takich jak Microsoft SQL Server, czy Oracle DBMS, skończywszy na darmowych, dostępnych w Internecie: MySQL, PostgreSQL.

Narzędziem do komunikacji z serwerami przetrzymującymi relacyjne bazy danych jest **oprogramowanie pośredniczące**, nazywane najczęściej „*front-end*". Tego typu oprogramowanie znacznie upraszcza nawiązywanie połączenia z serwerami baz danych oraz dostarcza intuicyjne narzędzia służące manipulacji zawartością zbiorów danych. Wszystkie te możliwości można aktywować za pomocą odpowiednio sparametryzowanych **zapytań SQL**, które to są następnie przetwarzane przez oprogramowanie pośredniczące[21]. Grafika numer 1-7 prezentuje jeden z najbardziej popularnych programów do zarządzania serwerem bazy danych MS SQL Server, SQL Server Management Studio 2014 Express:

[20] I. Ben-Gan: *Microsoft SQL Server 2012. T-SQL Fundamentals,* SolidQ, New York 2012, s. 10.

[21] I. Ben-Gan: *Microsoft SQL Server 2012. T-SQL Fundamentals,* SolidQ, New York 2012, s. 18.



Rysunek 1.7: SQL Server Management Studio 2014
Źródło: Opracowanie własne.

SQL Server Management Studio Express 2014 jest zaprojektowane w dobrze znanym stylu firmy Microsoft, łatwo wychwycić podobieństwa do innych narzędzi tej marki - choćby do pakietu Office. Drzewo po lewej stronie umożliwia przeglądanie wszystkich lokalnych baz danych wraz z ich zawartością (do pewnego wymiaru – aby odczytać konkretne rekordy bazy, należy użyć pełnoprawnych zapytań SQL), oraz baz do których jesteśmy zdalnie podłączeni. Środkowa część programu to główny obszar działań, gdzie z reguły wykonuje się zapytania SQL, których wyniki pojawiają się w środkowo-dolnym oknie. Ostatnia część, która pozostała do omówienia to okno „*Properties*" czyli tzw. właściwości. W tym miejscu można ustalić lub odczytać np. rodzaj danych przechowywanych w danej kolumnie, jej nazwę, opis, klucze i wiele innych parametrów.

Wedlug I. Ben-Gan'a, składnia języka SQL dzieli się na cztery główne podzbiory, utworzone w zależności od wykonywanej akcji przez zapytanie[22]:

- **SQL DQL** (*Data Query Language*) – podzbiór służący tworzeniu zapytań.
- **SQL DDL** (*Data Definiton Language*) – grupa mająca za zadanie tworzenie nowych części baz danych.

[22] I. Ben-Gan: *Microsoft SQL Server 2012. T-SQL Fundamentals*, SolidQ, New York 2012, s. 39.

- **SQL DCL** (*Data Control Language*) – zbiór zapytań kontrolujących dane zawarte w zbiorach.
- **SQL DML** (*Data Manipulation Language*) – podzbiór zapytań wykonujących operacje bezpośrednio na danych w bazach.

1.5.6. Haskell

Haskell to czysto-funkcyjny język programowania, który swoją nazwę zawdzięcza amerykańskiemu naukowcowi – Haskell'owi Curry'emu, którego to badania matematyczne doprowadziły do narodzin koncepcji funkcyjnego paradygmatu programowania. Język Haskell z założenia miał łączyć wszystkie najlepsze cechy języków funkcyjnych i to też udało się osiągnąć jego twórcom w 1990r. Najnowsza jego wersja pochodzi z 2010 roku, choć w planach jest premiera kolejnej postaci na rok 2016[23].

Realizacja paradygmatu programowania funkcyjnego przez Haskell odbywa się m.in. poprzez jego **leniwe wartościowanie.** Oznacza to, że wartość wyrażenia czy zmiennej nie musi być ściśle określona do momentu jawnego jej wywołania w kodzie. Rozwiązanie to pozwala na wyrafinowane rozwiązanie wielu problemów, szczególnie matematycznych. Przykładowa sytuacja, która zobrazuje korzyści wynikające z leniwego wartościowania to problem wyznaczenia liczb pierwszych. Kiedy nie jest znany przedział pożądanych liczb, dzięki leniwemu wartościowaniu możliwe staje się zdefiniowanie funkcji lub rekurencji, która wylicza w nieskończoność liczby pierwsze. Gdy użytkownik określi przedział pożądanych liczb pierwszych, program dokona obliczeń tylko tych żądanych wyników – gwarantuje to minimalne wykorzystanie zasobów komputera oraz uniknięcie niepotrzebnych operacji, co w niektórych stylach programowania jest bardzo pożądane[24].

Kolejną ważną cechą języka Haskell, która także wpisuje się w zasady paradygmatu programowania funkcyjnego jest **silne typowanie.** W zależności od punktu widzenia, właściwość tą uznać można za wadę lub zaletę. Z jednej strony, redukuje szansę na popełnienie błędu programistycznego np. poprzez rzutowanie

[23] B. O'Sullivan, J. Goerzen, D. B. Stewart: *Real World Haskell*, Sebastopol CA 2008, s. 68.

[24] B. O'Sullivan, J. Goerzen, D. B. Stewart: *Real World Haskell*, Sebastopol CA 2008, s. 70-71.

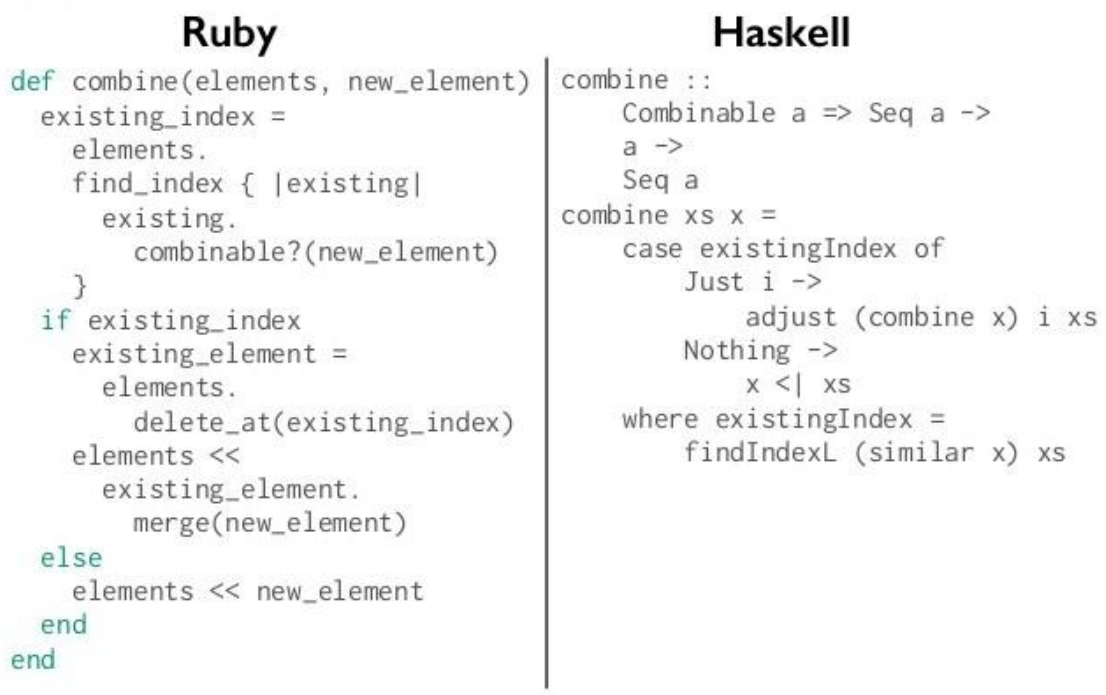
na siebie nieodpowiednich typów danych (np. *string* na *double*), zaś w innym przypadku może np. uniemożliwić konwersję konieczną do osiągnięcia zamierzonego celu, choć istnieją zaawansowane praktyki pozwalające ominąć ten problem. Z silnym typowaniem w Haskellu związane jest także **automatyczne rozpoznawanie typów danych**, co oznacza, że nie ma obowiązku deklarowania rodzaju używanych zmiennych – silnik Haskella zrobi to sam domyślnie.

Podsumowując, najważniejsze cechy czysto-funkcyjnego języka Haskell, według trzech autorów książki „*Real World Haskell",* sprowadzają się do tych kilku punktów[25]:

- **Język czysto-funkcyjny** – nie zezwala na stosowanie jakichkolwiek skutków ubocznych instrukcji.
- **Silne typowanie** – niemożliwa jest konwersja typów.
- **Automatyczne odśmiecanie pamięci** – zarządzanie pamięcią aplikacji należy do silnika Haskella.
- **Lakoniczna składnia** – w porównaniu do języków realizujących paradygmat np. obiektowy, programy napisane w Haskell'u zajmują zdecydowanie mniej miejsca na ekranie.
- **Modularność –** Haskell umożliwia swobodne łączenie ze sobą mniejszych modułów (funkcji), w jeden większy.

Jako dowód zwięzłości programów napisanych w języku Haskell posłuży grafika numer 1-8:

[25] B. O'Sullivan, J. Goerzen, D. B. Stewart: *Real World Haskell*, Sebastopol CA 2008, s. 81-86.



Rysunek 1.8: Porównanie kodu Haskell z Ruby.
Źródło: Opracowanie własne.

Rysunek numer 7 przedstawia dwie identycznie działające funkcje, które operują na kolekcji danych i zostały napisane w dwóch różnych językach – Haskell i Ruby. Algorytm sprawdza czy element wprowadzony do funkcji istnieje w kolekcji danych – jeśli tak – usuwa istniejący obiekt i zapisuje nowy do kolekcji poprzez odnalezienie jego lokalizacji w postaci indeksu. Jeżeli okaże się, że podany element nie jest zapisany w kolekcji, algorytm dodaje go na koniec zbioru. Tak działająca metoda napisana we w pełni obiektowym języku programowania Ruby, zajmuje blisko połowę więcej miejsca niż w funkcyjnym Haskell'u, co doskonale uwidacznia Rysunek 7. Dowodzi to celowej zwięzłości i klarowności składni Haskell'a.

1.5.7. JavaScript

Na wstępie warto zaznaczyć, iż wbrew temu, co mogą sugerować nazwy języków Java oraz JavaScript, ich podobieństwo do siebie zaczyna i kończy się na identycznie brzmiącym członie nazwy – powoduje to wiele pomyłek i nieporozumień, zwłaszcza wśród początkujących i niedoświadczonych informatyków.

JavaScript (JS) to wysokopoziomowy, skryptowy i interpretowalny język programowania, pozbawiony zdefiniowanych wbudowanych typów danych. Jest

wypadkową zażartego wyścigu z lat dziewięćdziesiątych ubiegłego wieku między firmami Netscape i Microsoft, o przodownictwo w dostarczaniu technologii webowych, z których to ta pierwsza zaprojektowała język JavaScript w maju 1995r. Jest to najczęściej używana technologia w procesie tworzenia witryn i aplikacji internetowych opartych na protokole WWW (*World Wide Web*), zaraz obok języków webowych HTML i CSS[26]. Zdecydowana większość dostępnych dziś kontentów internetowych działa w oparciu o język JavaScript, jak również wszystkie najpopularniejsze przeglądarki internetowe wspierają tą technologię. JavaScript realizuje po części kilka paradygmatów: m.in. poprzez posiadanie klas bazowych definiujących prototypy obiektów, wpisuje się w programowanie zorientowane obiektowo oraz poprzez traktowanie funkcji jako typów pierwszoklasowych realizuje paradygmat funkcyjny.

Oprócz szerokiej gamy zastosowań JavaScript'u w rozwiązaniach webowych, znajduje on też swoje miejsce w systemach takich jak dokumenty PDF, widżety desktopowe oraz gry komputerowe. Także zastosowanie języka JS w celu tworzenia platform obsługujących maszyny wirtualne poszerza zakres jego zastosowań.

Oto najważniejsze cechy języka JavaScript[27]:

- **Dynamiczność –** jak wiele języków skryptowych, JavaScript posiada dynamiczne typowanie zmiennych. Dzięki temu typy wartości programu mogą być definiowane podczas wykonywania kodu, a także istnieje możliwość przypisywania tej samej wartości do różnych zmiennych.
- **Prototypy –** inaczej niż w innych popularnych językach typowo obiektowych, które stosują dziedziczenie celem implementacji paradygmatu obiektowego i wielokrotnego wykorzystania kodu, JavaScript wykorzystuje system **prototypów**.
- **Funkcyjność –** paradygmat funkcyjny daje się we znaki w języku JavaScript poprzez możliwość definiowania funkcji jako tzw. typów pierwszoklasowych. Oznacza to, że funkcja może być traktowana jako tzw. funkcja wyższego rzędu, umożliwiając zwracanie przez funkcję lub wprowadzanie do funkcji innych metod, co znacznie uelastycznia kod.

[26] D. Crockford: *JavaScript: The Good Part*, Sebastopol CA 2008, s. 15.

[27] D. Crockford: *JavaScript: The Good Part*, Sebastopol CA 2008, s. 19-21.

- **Delegacje –** w języku JavaScript pozwalają na definiowanie obiektów, które z kolei polegają na innych obiektach w celu zapewnienia specyficznego zestawu funkcjonalności.
- **Rozszerzenia –** dynamiczny rozwój branży IT, zmusza fundację Mozilla do cyklicznego wydawania licznych rozszerzeń dla języka JavaScript, w celu utrzymania go wciąż atrakcyjnym dla web-deweloperów i nie tylko.

Najbardziej popularnym zastosowaniem języka ze stajni Netscape jest zapewnienie funkcjonalności po stronie klienta dla stron www napisanych w HTML. Oto konkretne przykłady, w których JavaScript spełnia swoją rolę[28]:

- Technologia AJAX, która poprzez język JavaScript pozwala na pozwala na aktualizację pożądanych danych na stronie, bez całkowitego jej przeładowania.
- Animowanie elementów graficznych znajdujących się na stronie WWW, zmiana ich rozmiaru czy miejsca wyświetlania.
- Interaktywna zawartość witryny, np. w postaci mini-gry.
- Walidowanie danych wprowadzanych przez użytkownika strony WWW, np. poprawność adresu PESEL, czy kodu pocztowego.
- Dysponowanie informacjami na temat zachowań i preferencji klienta, pomiędzy różnymi serwisami internetowymi w postaci plików *cookies.*

Ciekawostką jest, że aby odczytać kod JavaScript każdej strony www, należy kliknąć prawym przyciskiem myszy na layout strony i wybrać opcję „Pokaż źródło strony". Zrzut ekranu przedstawiony na grafice numer 1-9, pokazuje skutek takiej operacji:

[28] J. Duckett: *Effective JavaScript: 68 Specific Ways To Harness The Power Of JavaScript*, New Jersey 2013, s. 31-32.

```
<script type="text/javascript">

    (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
    (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
    m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
    })(window,document,'script','//www.google-analytics.com/analytics.js','ga');

    ga('create', 'UA-36116321-5', 'mozilla.org');
    ga('set', 'anonymizeIp', true);



    // dimension9 == "Section editing"


    (function() {
        // http://cfsimplicity.com/61/removing-analytics-clutter-from-campaign-urls
        var win = window;
        var removeUtms = function(){
            var location = win.location;
            if (location.href.indexOf('utm') != -1 && win.history.replaceState) {
                win.history.replaceState({}, '', location.pathname);
            }
        };

        ga('send', 'pageview', {'hitCallback': removeUtms});
    })();
</script>
```

Rysunek 1.9: Przykładowy skrypt JavaScript obecny na oficjalnej stronie dla deweloperów.
Źródło: Opracowanie własne na podstawie: http://developer.mozilla.org.

Powyższe zdjęcie ekranu prezentuje kod JavaScript strony fundacji Mozilla traktującej właśnie o tym języku. Podobny widok, czyli kod źródłowy każdej strony internetowej, można uzyskać poprzez kliknięcie prawym przyciskiem myszy w dowolnym miejscu strony internetowej oraz wybranie opcji (zależnie od przeglądarki) „Pokaż źródło strony". Oczywiście widok ten umożliwia jedynie odczyt danych bez możliwości edycji i może być bardzo użyteczny dla web-deweloperów, ponieważ umożliwia podgląd sposobu działania każdej strony internetowej dostępnej w sieci.

1.5.8. SmallTalk

SmallTalk to przede wszystkim jeden z pierwszych w pełni obiektowych języków programowania, dodatkowo posiadający dynamiczne typowanie i refleksję typów. Pomysł na język tego typu narodził się w firmie Xerox, we wczesnych latach siedemdziesiątych, a specjaliści którzy czuwali nad jego rozwojem, mieli już na swoim koncie badania nad takimi narzędziami jak graficzne interfejsy użytkownika, mysz komputerowa czy telekonferencje[29]. Według niepotwierdzonych źródeł informacji, tak

[29] K. Beck: *SmallTalk Best Practice Patterns*, New Jersey 1997, s. 11.

wpływowa koncepcja języka SmallTalk powstała w wyniku *zakładu* pomiędzy pracownikami firmy Xerox, polegającym na utworzeniu w pełni obiektowego języka programowania.

Jak wiele innych obiektowych języków programowania takich jak Java, Python, Ruby czy Objective-C, SmallTalk bazuje na języku Simula. Jego calkowita obiektowość polega na tym, że w przeciwieństwie np. do Javy i C# nie ma w nim żadnej różnicy pomiędzy typami wartościowymi, a referencyjnymi (obiektami) – „wszystko jest obiektem".

Znany amerykański wizjoner informatyki i ekspert wytwarzania programowania Kent Beck, wspomina następujące mechanizmy SmallTalk'a[30]:

Refleksja (*reflection)*

SmallTalk jest prekursorem refleksji, która to oznacza, iż program opatrzony tą cechą, musi zawierać obiekty zdefiniowane i opisane w tym właśnie języku. Dzięki refleksji program posiada także dostęp do informacji o swoich własnych elementach, oraz jest w stanie je modyfikować w czasie działania programu. Obecną powszechność refleksji i występowanie jej w większości języków programowania, zawdzięcza się przede wszystkim SmallTalk'owi, który to wprowadził i upowszechnił ten mechanizm. Konkretnym przykładem obrazującym działanie i zastosowanie refleksji może być np. możliwość programistycznego wykrycia „w locie" wszystkich instancji określonego obiektu. Zależnie od zdefiniowanego wyjścia programu, rezultatem takiego działania może być np. wyświetlenie na ekranie komputera wszystkich miejsc oraz ilości wystąpień metody „*new"* definiującej nowy obiekt.

Wiadomości (*messages)*

System wiadomości to fundamentalna i najbardziej wyróżniająca język cecha spośród innych. SmallTalk domyślnie posiada obsługę synchronicznej, dynamicznej i pojedynczej wysyłki wiadomości, w przeciwieństwie do innych języków obiektowych, które zarządzają wiadomościami wysyłanymi wielokrotnie i asynchronicznie. Wiadomości wywołują odpowiednie metody i funkcje przypisane do używanego obiektu, przykładowo komenda „*factorial"* połączona z odpowiednim obiektem, wysyła do niego żądanie obliczenia silni (wywołanie funkcji silnia) z możliwością zapisania

[30] K. Beck: *SmallTalk Best Practice Patterns,* New Jersey 1997, s. 21-30.

zwracanej wartości w nowym obiekcie. Wiadomości mogą przesyłać także argumenty, które są dostarczane do obiektów, np. argumentem wiadomości może być liczba przez którą należy pomnożyć obiekt wysyłający wiadomość.

Wyrażenia (*expressions)*

Wyrażenia odnoszą się do kwintesencji języka SmallTalk – wiadomości. Składnia ta pozwala na tworzenie właśnie wyrażeń, składających się z wielu wiadomości. Każda wiadomość ma swój domyślny priorytet wykonania, a dzieje się to w tej kolejności: pojedyncze wiadomości (jednokrotne polecenia), binarne wiadomości (działania arytmetyczne) i na końcu wiadomości w postaci słów kluczowych (*and, or, if).* Kolejność wykonywania wiadomości w wyrażeniach, może być porządkowana wedle uznania programisty za pomocą par nawiasów, analogicznie jak ma to zastosowanie w działaniach arytmetycznych.

Image-based persistence

Najbardziej popularne systemy programistyczne oddzielają statyczne części programu (klasy, funkcje, procedury) od dynamicznej części, czyli środowiska wykonującego program. W tym podejściu system ładuje surowy kod źródłowy, a wszelkie wcześniej zapisane stany programu muszą być jawnie odczytane z baz danych czy plików konfiguracyjnych. Taki styl rejestracji stanu programu m.in. nie pozwala na późniejsze odczytanie historii wykonywanych operacji.

Zgoła odmienne podejście do tego problemu zastosowano w języku SmallTalk, które nie traci danych historii działań czy pozycji kursora. Silnik tego języka nie oddziela kodu programu (klas) od danych (obiektów), dlatego stan całego programu SmallTalk jest zapisywany i przechowywany w pliku zwanym **obrazem *(image).*** Przy wykonywaniu programu, obraz jest ładowany do wirtualnej maszyny, która przywraca zapisany w pliku wcześniejszy stan programu. Oprócz korzyści wynikających z tego rozwiązania istnieją także przeszkody. Jednym z problemów, jakie niesie ze sobą ta metoda, są liczne utrudnienia przy ukrywaniu szczegółów logicznych programu, co zwłaszcza w komercyjnych projektach jest często pożądane wśród deweloperów.

2. PARADYGMAT PROGRAMOWANIA OBIEKTOWEGO

Programowanie obiektowe jako stosunkowo młode podejście do wytwarzania oprogramowania komputerowego, posiada wiele istotnych i przydatnych cech, które zostaną skrupulatnie omówione w tym rozdziale. Bieżący rozdział zaznajomi czytelnika z historią tego popularnego dziś paradygmatu oraz przede wszystkim przedstawi jego trzy główne filary: dziedziczenie, polimorfizm i enkapsulację. Wszystko to pozwoli na pełne zrozumienie przesłania i modelowania rzeczywistości jakie dostarcza obiektowy paradygmat programowania.

2.1. Historia

Koncepcja programowania oparta o obiekty najprawdopodobniej pierwszy raz pojawiła się na przełomie lat pięćdziesiątych i sześćdziesiątych ubiegłego wieku w Instytucie Technologicznym w Massachusetts. Termin „obiektu" zaczął być tam używany m.in. w celu opisu przedmiotów identyfikowanych przez sztuczną inteligencję, pojawił się we wczesnych wersjach języka ALGOL przygotowanego przez instytut, a także wystąpił w rewolucyjnym programie „*Sketchpad*" napisanym przez członka Instytutu Technologicznego w Massachusetts – Alana Kay'a[31].

Formalnie obiektowy paradygmat programowania został wprowadzony wraz z językiem **Simula 67**, który został zaprojektowany w Norweskim Centrum Komputerowym przez Ole-Johana Dahl'a oraz Kristen'a Nygaard'a. Język Simula 67 czerpał garściami z nowoczesnych modułów programistycznych jak obiekty, klasy, instancje czy metody i w ten sposób doprowadził do wyabstrahowania nowatorskiego paradygmatu programowania obiektowego. Nowe możliwości dostarczane przez Simule 67 były wykorzystywane m.in. do symulacji ruchu morskiego, co pozwoliło na znaczne usprawnienia w transporcie wodnym. Ostatecznie język Simula stał się inspiracją do utworzenia tak znaczących języków jak Object Pascal, C++ czy szczegółowo charakteryzowany w niniejszym studium SmallTalk[32].

[31] R. W. Sebesta: *Concept of Programming Languages,* New Jersey 2012 s. 29-32.

[32] R. W. Sebesta: *Concept of Programming Languages,* New Jersey 2012 s. 20.

Jak już wspomniano w podrozdziale 1.5.8 język SmallTalk był pierwszym językiem, który poprzez swoją strukturę wymuszał obiektowe wytwarzanie oprogramowania. Pomimo, że główną inspiracją dla autorów języka SmallTalk był w dużym stopniu język Simula, to ten pierwszy jest w pełni obiektowy i w pełni dynamiczny. SmallTalk wraz ze swoim obiektowym paradygmatem został przedstawiony szerszej publiczności w sierpniu 1981 roku[33]. Szczegółowy opis tego języka znajduje się w rozdziale 1.5.8 oraz 3.2.2.

Już we wczesnych latach dziewięćdziesiątych obiektowe projektowanie oprogramowania stało się dominującym stylem, także ze względu na szeroką dostępność języków programowania takich jak C++ czy Delphi, które dostarczały niezbędne do tego narzędzia. Kolejną przyczyną płynnego rozwoju stylu obiektowego było ekspresowe tempo wzrostu popularność technologii **graficznych interfejsów użytkownika** (*GUI – Graphic User Interface*), która to swoje fundamenty opierała właśnie na obiektowych technikach programowania[34]. Przykładem współpracy graficznej biblioteki z obiektowym językiem programowania jest system *Mac OS X*, który to został napisany w języku Objective-C.

Najnowsza historia obiektowego wytwarzania software'u skupia się na językach obiektowych wspierających także proceduralny i funkcyjny paradygmat. Nie bez powodu najważniejsze i najczęściej używane komercyjnie języki naszych czasów wpisują się idealnie w ten schemat – prostota projektowania i szerokość zastosowań czynią z nich najbardziej wszechstronne narzędzia, a konkretnymi przykładami w tym przypadku są np. Java (Sun Microsystems) czy C# (Microsoft, platforma .NET).

2.2. Najważniejsze cechy

Według K. Wojtuszkiewicza, programowanie obiektowe oferuje kilka istotnych i niezwykle użytecznych cech[35]:

- Możliwość realnego odzwierciedlania rzeczywistości w kodzie poprzez struktury **klas i obiektów**. Proces ten odbywa się poprzez modelowanie danych

[33] S. Kedar: *Programiing Paradigms and Methodology,* New York 2008, s. 42.

[34] B. Meyer: *Programowanie zorientowane obiektowo*, Gliwice 2005, s. 51.

[35] K. Wojtuszkiewicz: *Programowanie strukturalne i obiektowe,* Warszawa 2010, s. 11.

za pomocą funkcji, które operują na obiektach, będącymi reprezentantami klas. W kontekście obiektów używa się terminów **właściwości oraz metod**, które zostaną omówione w dalszej części pracy.

- Znaczne uproszczenie podziału dużych projektów programistycznych na mniejsze fragmenty z możliwością modyfikacji poszczególnych modułów w przyszłości. Wiąże się to m.in. z terminami **hermetyzacji** oraz **dziedziczenia.**
- Wielokrotne wykorzystanie kodu, poprzez ustawiczne posługiwanie się zdefiniowanymi klasami oraz funkcjami, a także tworzenie podklas w imię dziedziczenia. Za śmiałym korzystaniem z tej sposobności przemawia dobrze znana wśród software deweloperów zasada *DRY*, której rozwinięcie brzmi *Don't Repeat Yourself.*

Wymienione cechy i terminy w połączeniu z pozostałymi mechanizmami programowania obiektowego, ściśle współgrają tworząc przystępny wzór wytwarzania oprogramowania. Kolejne rozdziały pozwolą uporządkować i jasno zrozumieć podstawowe terminy rządzące tym popularnym paradygmatem.

2.2.1. Czym właściwie jest obiekt?

Programy i aplikacje stworzone w oparciu o obiektowy paradygmat programowania zbudowane są głównie z obiektów, a nawet można pokusić się o stwierdzenie, że w gruncie rzeczy są **kolekcjami obiektów.** Obiekty przechowują w swojej strukturze dane, które reprezentują stan obiektu, a fachowe słownictwo programistyczne określa te dane jako **atrybuty**[36]. Przykładowo, w systemie zarządczym przedsiębiorstwa, obiektem może być abstrakcja w postaci pracownika. Obiekt pracownik najprawdopodobniej zawierałby wtedy atrybuty takie jak imię i nazwisko, stanowisko, wiek, płeć, numer pesel i tym podobne. Wartości atrybutów służą również jako rozróżnianie między sobą obiektów wywodzących się z tej samej klasy – niemożliwym jest, aby w przedsiębiorstwie pojawiło się dwóch pracowników o tym samym adresie oraz nazwisku.

Oprócz atrybutów obiekty charakteryzuje także zachowanie, które określa jakie operacje na danych obiekt może wykonywać. Obiektowy paradygmat wyraża

[36] M. Weisfeld: *Myślenie obiektowe w programowaniu*, Gliwice 2014, s. 68-69.

zachowanie poprzez **metody,** które mogą być wywołane poprzez wysłanie wiadomości o odpowiedniej treści do odpowiedniego obiektu[37]. Trzymając się przyjętej konwencji obiektu jako pracownika w systemie przedsiębiorstwa, obiekt taki mógłby posiadać metody pozwalające np. na odczytywanie oraz zapisywanie wartości poszczególnych atrybutów. W celu poprawnego i efektywnego wywołania zdefiniowanej metody konkretnego obiektu, użytkownik musi znać:

- Poprawną nazwę metody,
- Parametry przesyłane do metody,
- Typ zwracany przez metodę.

Brnąc dalej w przykład fikcyjnej aplikacji pracowniczej, załóżmy na potrzeby zrozumienia paradygmatu obiektowego, iż zawiera ona także obiekty reprezentujące rachunek dla konkretnego pracownika. Użyteczność takiego obiektu byłaby zdeterminowana przez metodę o nazwie *CountSalary*(), która zwraca wysokość wynagrodzenia dla konkretnego pracownika. Funkcja kalkulująca pensję pracowniczą wymaga przesłania do niej obiektu konkretnego pracownika, z którego to później wyciągane są informacje o pracowniku za pomocą wezwań metod zwracających wartości jego atrybutów. Listing numer 2.1 przedstawia przykładowy wygląd metody *CountSalary*() sporządzonej w pseudo-kodzie.

```
ComputeSalary(employee)-returnType
{
     salary = (employee.Rate * employee.WorkedTime) +
     employee.Bonus;
     return-salary;
}
```

Listing 2.1 Metoda *CountSalary*().
Źródło: Opracowanie własne.

Najważniejsza część metody czyli jej **sygnatura** mieszcząca się w pierwszej linii, zawiera w sobie wiele przydatnych informacji na temat tej funkcji. Po pierwsze typ zwracany przez tę funkcję to *double* – typ liczbowy przechowujący liczby

[37] M. Weisfeld: *Myślenie obiektowe w programowaniu*, Gliwice 2014, s. 87.

zmiennoprzecinkowe (pensja nie musi być liczbą całkowitą). Kolejna część sygnatury dostarcza nazwę metody (*ComputeSalary()*) oraz typ przesyłany do metody, niezbędny do jej wywołania – niestandardowy typ *Pracownik,* zdefiniowany przez programistę. Natomiast ciało metody zawierające się w nawiasach klamrowych służy jako kontener jej działań. W tym przypadku metoda definiuje typ *double* o nazwie pensja, którego wartość oblicza poprzez wywoływanie odpowiednich atrybutów przesłanego obiektu typu *Pracownik*, ostatni etap to zwrócenie do procedury wywołującej obliczonej wartości.

2.2.2. Co to jest klasa?

Z racji niemożności wyjaśnienia terminu „obiekt" bez używania słowa „klasa" i na odwrót, w poprzednim podrozdziale posłużono się obiema wyrażeniami, a pomimo tego schemat wyjaśniający termin „obiekt" nie został zaburzony. Przechodząc do rzeczy, **klasa** jest formą dla obiektu. Kiedy używamy słowa kluczowego *new* celem alokowania nowego obiektu określonego typu, pośrednio używamy jego klasy, która definiuje jak obiekt jest zbudowany. Posługując się realnym przykładem: „auto" jest klasą, natomiast samochód znajomego Audi A6 o konkretnym kolorze, modelu i numerze rejestracyjnym to już rzeczywisty obiekt. Dlatego właśnie klasę określa się wzorcem, formą, modelem czy szablonem dla obiektów.

Klasy składają się z **pól,** które w obiektach reprezentowane są w postaci atrybutów, oraz metod, które to z kolei określają zachowanie obiektu tworzonego przez tą klasę[38]. Przykładową klasę niestandardowego typu *Car* przedstawia listing numer 2.2:

```
class-Car
{
    field-mark;
    field-model;
    field-color;
    field-price;

    function-ShowInfo()
```

[38] B. McLaughlin: *Head First. Object-Oriented Analysis and Design,* Sebastopol CA 2008, s. 75.

```
        {
            return-("Car {0} {1}, Color{2} i
            cenie {3}zł", this.Mark, this.Model,
            this.Color, this.Price);
        }

        function-ChangePrice(newPrice)
        {
             this.Cena = newPrice;
        }
    }
```

Listing 2.2 Przykładowa klasa typu *Car*.
Źródło: Opracowanie własne.

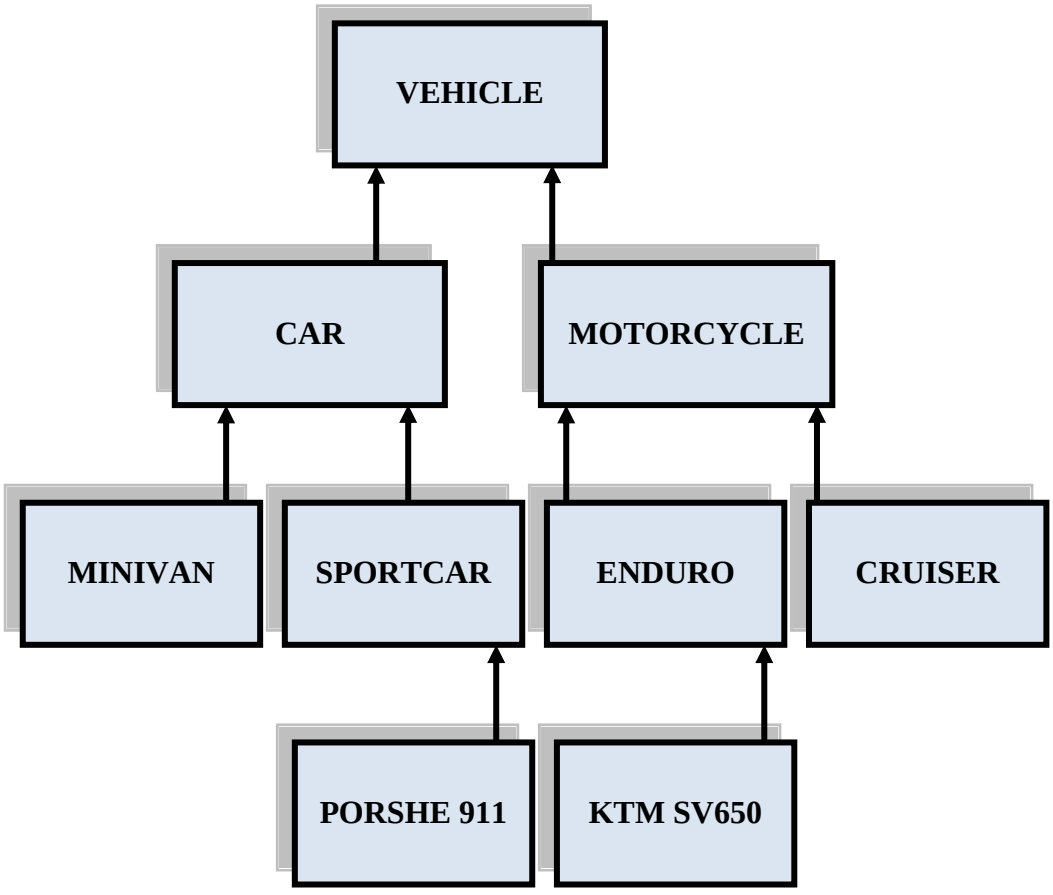
W tym przykładzie klasa posiada cztery pola, które po wywołaniu instancji obiektu, utworzą jego atrybuty z automatycznymi właściwościami (metody odczytujące i zapisujące wybrany atrybut). Te pola to trzy wartości w postaci łańcucha znaków oraz jedna typu liczbowego – *Price*. Oprócz pól, klasa dostarcza do obiektu również dwie metody: *ShowInfo()* – wyświetla na ekranie dane obiektu, dla którego wywołana została metoda, oraz *ChangePrice()* – koryguje cenę samochodu o przesłaną wartość liczbową do metody.

2.3. Dziedziczenie

Znane już z programowania proceduralnego, wielokrotne wykorzystanie raz napisanego kodu, to najprawdopodobniej jedna z tych właściwości, które czynią obiektowy paradygmat programowania tak bardzo uniwersalnym i elastycznym. Choć **wielokrotne wykorzystanie kodu** znane jest w świecie programistów znacznie dłużej niż samo istnienie pojęcia obiektowości, paradygmat ten, wynosi je na znacznie wyższy poziom. Dzieje się to poprzez możliwość definiowania relacji pomiędzy klasami, dzięki czemu otrzymujemy kod dobrze zorganizowany, klarowny i pozbawiony dużej ilości

podobnych do siebie klas. Wszystko to zapewnia jeden z trzech filarów obiektowości - **dziedziczenie**[39]**.**

Zasada działania tego mechanizmu polega na *dziedziczeniu* przez wybraną klasę lub obiekt atrybutów oraz metod po innej, określonej klasie lub obiekcie. W ten sposób pojawia się możliwość ograniczenia ilości zbędnych struktur, poprzez definiowanie jednej, zawierającej w sobie wspólne składowe dla określonej rodziny typów. Powodowane przez dziedziczenie relacje pomiędzy obiektami lub klasami prowadzą z kolei do tworzenia się hierarchii struktur w systemach informatycznych. Przykładową hierarchię klas utrzymaną w konwencji motoryzacyjnej przedstawia poniższy schemat:



Rysunek 2.1: Przykładowy diagram klas.
Źródło: Opracowanie własne.

Powyższa hierarchia klas utrzymana w konwencji motoryzacyjnej zawiera cztery poziomy obiektów. W realnym systemie informatycznym, klasa zajmująca najwyższe miejsce w hierarchii (w tym przypadku *Vehicle*) zawiera pola i metody niezbędne dla

[39] B. McLaughlin: *Head First. Object-Oriented Analysis and Design*, Sebastopol CA 2008, s. 133-135.

wszystkich obiektów po niej dziedziczących i analogicznie w dół struktury organizacyjnej. Zgodnie z tą zasadą, w hierarchii mieszczącej się na rysunku 2.2, klasa *Vehicle* powinna zawierać metody pozwalające np. na uruchomienie silnika (każdy abstrakcyjny pojazd posiada silnik), zatrzymanie pracy silnika czy zmiana kierunku jazdy. Z kolei konkretne obiekty z czwartego poziomu abstrakcji, oprócz dziedziczenia wszystkich składowych ze swoich nadklas, zawierałyby unikalne dla siebie metody i pola: przykładowo klasa *Porshe911* mogłaby korzystać z metody *TurnOnNitro().*

2.3.1. Rodzaje dziedziczenia

Termin dziedziczenia jest używany w światku informatycznym zarówno w przypadku dziedziczenia opartego na klasach jak i na prototypach, a przez ogromną ilość języków programowania oraz podejść ich twórców do obiektowego paradygmatu wyróżnia się kilka rodzajów dziedziczenia. M. Weisfeld zaproponował następujący podział[40]:

- **Pojedyncze** – w dziedziczeniu pojedynczym klasa lub obiekt może otrzymywać metody i pola tylko jednej wybranej struktury, a także nie ma możliwości dalszego dziedziczenia po zdefiniowanej podklasie. Najbardziej ograniczony i rzadko spotykany typ dziedziczenia.
- **Wielokrotne –** ten rodzaj dziedziczenia pozwala klasie lub obiektowi dziedziczyć cechy z wielu różnych struktur. Przez niektórych specjalistów uważane za niemożliwe do efektywnej implementacji, a czasem traktowane jako znaczne utrudnienie np. w języku C++.
- **Wielopoziomowe –** ma miejsce wtedy, gdy podklasa ma możliwość dziedziczenia cech z innej podklasy. Najczęściej spotykane i najbardziej efektywny rodzaj dziedziczenia w paradygmacie obiektowym. Dziedziczenie wielopoziomowe jest wspierane przez języki takie jak Java, C# czy Visual Basic, a wizualne zarys tego typu przedstawia rysunek 2.2.
- **Hybrydowe –** połączenie dwóch lub więcej wymienionych typów dziedziczenia.

[40] M. Weisfeld: *Myślenie obiektowe w programowaniu,* Gliwice 2014, s. 95-97.

2.3.2. Podklasy i nadklasy

Terminy zawarte w tytule tego podrozdziału pojawiły się już wcześniej w tym opracowaniu, jednak nie jest możliwym wytłumaczenie pewnych pojęć bez stosowania fachowej terminologii. Zjawisko to doskonale potwierdza popularne wśród programistów powiedzenie „Aby dowiedzieć się czegokolwiek, musisz wiedzieć wszystko" (*Before you can know anything, you have to know everything*).

Pojęcie podklasy (*subclass*) może być stosowane zamiennie z takimi terminami jak klasa dziedzicząca (*derived class*) czy klasa-potomek (*child class*). Jest to klasa, która przejmuje jedną lub więcej składowych z innych klas, które z kolei zwane są nadklasami (*superclasses*), klasami bazowymi (*base classes*) lub klasami-rodzicami (*parent classes*)[41]. Oprócz nadklas i podklas w niektórych językach programowania istnieje możliwość zdefiniowania klasy, która nie ma możliwości udostępnienia swoich składowych dla kolejnej podklasy (*uninheritable class*). Przykładowo w Javie klasę taką należy opatrzyć słowem kluczowym *final*[42], podczas gdy język C# wymaga do tej czynności wyrażenia *sealed*[43].

Zgodnie z powyższym podklasa dziedziczy członków swojej nadklasy, a ogólny wzór implementacji podklasy przedstawia poniższy listing:

```
visibility-mode derived-class-name : parent-class-name
{
    //class members…
}
```

Listing 2.3: Wzór implementacji klasy dziedziczącej.
Źródło: Opracowanie własne.

Powyższy model prezentuje sposób definiowania klasy dziedziczącej cechy nadklasy. W pierwszej kolejności zapisuje się opcjonalny modyfikator dostępu (domyślnie zawsze jest to *private),* następnie nazwę tworzonej klasy, natomiast po dwukropku niezbędne jest wskazanie nazwy klasy bazowej. Miejsce pomiędzy nawiasami klamrowymi służy do definiowania składowych klasy czyli metod i pól,

[41] M. Weisfeld: *Myślenie obiektowe w programowaniu,* Gliwice 2014, s. 100.

[42] C. S. Horstmann, G. Cornell: *Core Java. Volume I – Fundamentals, Ninth Edition*, New Jersey 2013, s. 203.

[43] A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework*, New York 2015, s. 184.

które ta będzie zawierać. Niektóre języki programowania obiektowego wspierają dziedziczenie dla klas lub obiektów również z innych struktur, jak np. dziedziczenie interfejsów w C# (szerzej omówione w podrozdziale X.X). Powyższy listing to jedynie wzór, dlatego konkretne implementacje mogą posiadać nieznaczne różnice, jednak na pewno nie odstają znacznie od przedstawionego modelu.

2.3.3. Nadpisywanie funkcjonalności

Wiele języków programowania realizujących obiektowy paradygmat tworzenia aplikacji, pozwala na nadpisywanie dziedziczonych cech (najczęściej zachowania) poprzez definiowanie nowej implementacji tej samej metody. Nadpisywanie metod wprowadza pewną komplikacje, a mianowicie, która implementacja metody jest wywoływana: bazowa czy dziedziczona. Na to pytanie istnieją różne odpowiedzi zależnie od wybranego języka programowania.

Dla przykładu, aby w języku C# uczynić metodę nadpisywalną, należy jej sygnaturę opatrzyć słowem kluczowym *virtual* lub *abstrac*[44]*t.* Poniższy listing przedstawia wzór nadpisywania dziedziczonej metody:

```
visibility-mode override-keyword function-name
{
    base-class-name.base-function-name
    //new implementation…
}
```

Listing 2.4: Wzór nadpisywania dziedziczonej metody.
Źródło: Opracowanie własne.

Jak wynika z powyższego listingu, w procesie nadpisywania metody również istnieje możliwość ustalenia modyfikatora dostępu. Koniecznym natomiast jest użycie odpowiedniego dla używanego języka słowa kluczowego, które definiuje funkcję jako nadpisywaną. W ciele metody najczęściej wywołuje się bazową formę dziedziczonej metody, przed lub po dodanej implementacji (w tym przypadku przed), oraz dopisuje nową, pożądane zachowanie. Również jak w przypadku klas, metody można pieczętować, uniemożliwiając w ten sposób nadpisywanie ich poprzez użycie odpowiednich dla danego języka wyrażeń (*final*, *sealed*, *frozen*).

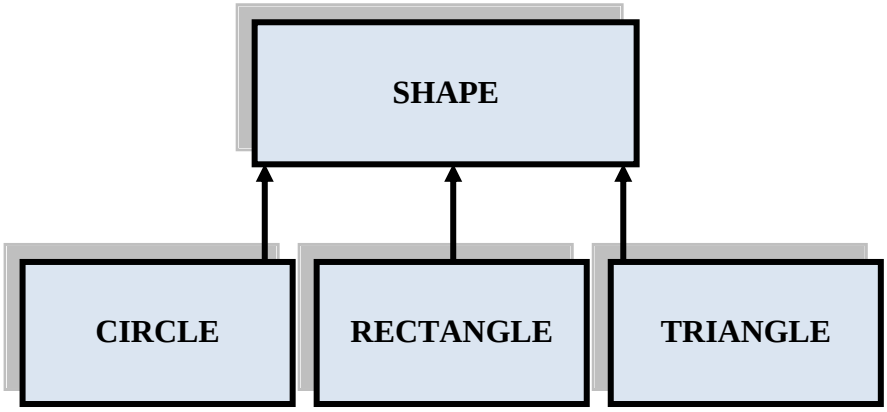
[44] A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework*, New York 2015, s. 195

Zagadnienie nadpisywania funkcjonalności w klasach dziedziczących prowadzi do osobnego, równie rozległego tematu, jakim jest polimorfizm, opisany w kolejnym podrozdziale.

2.4. Polimorfizm

Wyrażenie polimorfizm pochodzi od greckiego słowa *„polymorphism”,* które to w wolnym tłumaczeniu oznacza „**wielopostaciowość**”[45]. Pomimo tego, że polimorfizm w obiektowym programowaniu aplikacji wywodzi się i jest ściśle związany z dziedziczeniem, większość źródeł technicznych kwalifikuje go jako odrębny i samodzielny filar programowania obiektowego.

Poprzedni podrozdział przybliżył zasadę działania mechaniki dziedziczenia: wszystkie podklasy dziedziczą metody oraz atrybuty ze swoich klas bazowych. Kiedy wysyłamy do obiektu odpowiednie żądanie, wykonywana jest wybrana przez użytkownika funkcja. W tym sensie dziedziczenie nie zapewnia rozwiązania dla powstającego problemu: klasy na samym dole hierarchii, są osobnymi, różnymi od siebie encjami, co powoduje konieczność uzyskiwania różnych odpowiedzi na te same instrukcje wysyłane do obiektu. Prosta hierarchia klas realizująca dziedziczenie pojedyncze z poniższego rysunku, posłuży jako narzędzie zrozumienia idei polimorfizmu.



Rysunek 2.2: Hierarchia klas kształtów.
Źródło: Opracowanie własne.

[45] M. Gabrielli, S. Martini: *Programming Languages: Principles and Paradigms*, New York 2010, s. 178.

Posługując się przykładową hierarchią klas przedstawioną na rysunku numer 2.3, założyć można, że wszystkie obiekty dziedziczą po głównej klasie kształtu metodę rysującą figurę na ekranie: *Draw()*. Oczywistym jednak jest, że kształty takie jak koło, prostokąt czy trójkąt, wymagają odmiennego sposobu wyrysowania go na monitorze komputera. Powstaje więc pytanie jak zapewnić taką samą nazwę metody, dla zachowania spójności i klarowności systemu, implementując tym samym różne działanie funkcji? Na to zagadkowe pytanie odpowiedzią jest właśnie polimorfizm.

Polimorfizm w obiektowym paradygmacie programowania objawia się poprzez możliwość **nadpisywania (przesłaniania)** dziedziczonych funkcji z klas bazowych, po uprzednim oznakowaniu tych metod w odpowiedni sposób. Metoda (funkcja) klasy bazowej może zostać oznakowana odpowiednim słowem kluczowym, które zapewnia możliwość przesłonienia metody w klasie potomnej. Dzięki temu wymuszamy w klasie potomnej możliwość przesłonienia tej metody poprzez oznaczenie jej słowem kluczowym, np. *overrides* w języku Visual Basic platformy .NET. Jeśli nadpisywalna funkcja nie zostanie przesłonięta, klasa potomna otrzyma jej oryginalną implementację, co daje użytkownikowi obiektu możliwość decyzji jaką implementację funkcji obiekt powinien posiadać[46].

Praktyczne przedstawienie powyżej opisanej funkcjonalności znajduje się na następujących listingach, które wykorzystują przedstawioną hierarchię klas kształtów. Pierwszy kod źródłowy zawiera klasę *Shape* z dwiema metodami przeznaczonymi do nadpisania w klasie potomnej:

```
class Shape
{
    field-middlePoint;
    field-size;
    field-color;

    overridable-function-Draw()
    {
        //base implementation
```

[46] B. McLaughlin: *Head First. Object-Oriented Analysis and Design*, Sebastopol CA 2008, s. 152-154.

```
    }

    overridable-function-GetInfo()
    {
        return-middlePoint-size-color;
    }
}
```

Listing 2.5: Bazowa klasa *Shape*.
Źródło: Opracowanie własne.

Powyższa klasa bazowa *Shape* posiada trzy pola reprezentujące kolejno środkowy punkt figury, rozmiar oraz kolor – każda kolejna klasa wywodząca się z *Vehicle* będzie posiadała te składowe. Oprócz wymienionych atrybutów, klasa definiuje również zachowanie poprzez metodę *Draw()*, wyświetlającą kształt na ekranie, oraz metodę *GetInfo()*, zwracającą podstawowe informacje na temat obiektu – implementacje oby tych metod mogą zostać przesłonięte w klasie dziedziczącej z *Shape.*

Samo zadeklarowanie klasy z nadpisywalnymi metodami w kodzie nie przynosi jeszcze żadnych korzyści, pozostaje poprawna deklaracja klasy potomnej, jak na listingu numer 2.6:

```
class-Circle-inherited-Shape
{
     field-radius;

     override-function-Draw()
     {
         //drawing a circle
     }

     override-function-GetInfo()
     {
         Base-function-name;
         Return radius;
```

```
        }
}
```

Listing 2.6: Potomna klasa *Circle*.
Źródło: Opracowanie własne.

Powyższy przykład klasy okręgu pokazuje jakie zalety niesie ze sobą definiowanie nadklas dla obiektów wpisujących się w ogólną kategorię. Oprócz pól dziedziczonych z klasy *Shape,* czyli punktu środkowy, rozmiaru oraz koloru, obiekt klasy *Circle* otrzymuje to, czego potrzebuje każdy okrąg – długość promienia. Dodatkowo klasa *Circle* nadpisuje dwie dziedziczone metody: renderującą kształt na ekranie specjalnie dla okręgu, oraz wyświetlającą informację na temat obiektu. W przypadku funkcji wirtualnej wykorzystany jest mechanizm wywoływania bazowej implementacji metody z klasy bazowej, a następnie dodane jest zwrócenie informacji o promieniu obiektu klasy *Circle.* Dzięki temu zabiegowi funkcja *GetInfo()* w klasie *Circle* zwraca wartości pól zarówno dziedziczonych z *Shape* jak i dodatkowe pole promienia tego okręgu, bez powielania kodu źródłowego z bazowej klasy w klasie potomnej.

Omówiony przykład w pełni przedstawia ogrom korzyści płynących z szeroko rozumianego dziedziczenia i polimorfizmu, definiowania hierarchii klas, metod umożliwiających nadpisywanie oraz przesłanianie ich. W zaprezentowanym systemie, dziedziczenie pól wspólnych dla wszystkich obiektów jednej kategorii, oraz nadpisywanie wirtualnych i abstrakcyjnych metod, oszczędziło zbędnego kopiowania kodu, doszukiwania się nazw odpowiednich metod, doszukiwania się odpowiednich nazw metod i atrybutów, a także nie naruszono zasady *DRY*. Dziedziczenie razem z polimorfizmem pozostaje dziś znakiem firmowym oraz najpotężniejszym widżetem obiektowego paradygmatu programowania, który pozwala modelować systemy informatyczne na wzór opisywanej rzeczywistości.

2.5. Enkapsulacja

Jednym z trzech głównych komponentów paradygmatu programowania obiektowego jest enkapsulacja, która polega na tym, iż obiekty nie ujawniają na zewnątrz wszystkich swoich zachowań i szczegółowej implementacji. W dobrze zaprojektowanym systemie wspierającym zasady obiektowości, obiekty udostępniają

użytkownikowi jedynie interfejs operacji niezbędnych do interakcji z nim, a szczegóły niewymagane do jego poprawnego wykorzystania są ukrywane i niedostępne[47].

Kluczowym elementem realizacji enkapsulacji w modelu obiektowym są **modyfikatory dostępu**, czyli słowa kluczowe określające stopień dostępu do składowej obiektu (atrybutu bądź metody). Zależnie od języka programowania istnieje wiele dyrektyw, które w rożny sposób określają stopień dostępności wybranej składowej. Najpopularniejsze z nich to wyrażenia, które pozwalają na dostęp do składowej tylko wewnątrz jej rodzimej klasy oraz te, które udostępniają obiekt na zewnątrz. Obok tych dwóch najczęściej spotykanych technik, w niektórych językach istnieje też możliwość deklarowania elementów jako dostępnych jedynie w klasie potomnej, czy w tej samej bibliotece kodu. Dobrą praktyką paradygmatu obiektowego jest pieczętowanie wszystkich możliwych składowych modyfikatorem *private,* jednak nie zawsze jest to możliwe i dlatego właśnie powstały różne techniki enkapsulacji danych[48].

Podczas gdy większość języków programowania umożliwia kontrolę dostępu do pól obiektu właśnie poprzez wspomniane wcześniej modyfikatory dostępu, niektóre podejścia do enkapsulacji umożliwiają dostęp do składowych obiektu jedynie z poziomu specjalnie do tego służących metod, jak na przykład szeroko opisywany w tym opracowaniu język SmallTalk[49]. Technika ta jest powszechna również w językach, które w pełni wspierają modyfikatory dostępu i służy sztucznemu ukrywaniu szczegółowej implementacji zachowań obiektów, przed użytkownikiem zewnętrznym. Ukrywanie wewnętrznych szczegółów obiektów programistycznych ma na celu zapobieganie nieautoryzowanej modyfikacji danych, ponieważ nieodpowiedzialna reorganizacja implementacji algorytmów, prowadzi do zakłócenia spójności
i prawidłowości systemu informatycznego. Niewątpliwą zaletą tego rodzaju ukrywania danych jest zmniejszenie złożoności systemów, poprzez redukcję zależności i powiązań

[47] B. Meyer: *Programowanie zorientowane obiektowo,* Gliwice 2005, s. 196.

[48] B. McLaughlin: *Head First. Object-Oriented Analysis and Design,* Sebastopol CA 2008, s. 111-112.

[49] K. Beck: *SmallTalk Best Practice Patterns,* New Jersey 1997, s. 92-93.

pomiędzy poszczególnymi modułami[50]. Poniższy pseudo-kod przedstawia obrazową implementację enkapsulacji danych:

```
class-Add
{
    private-field-result;
    private-function-Compute(x, y)
    {
        result = x + y;
        return-result;
    }

    public-function-GetValue(x, y)
    {
        return-result(x, y);
    }
}
```

Listing 2.7: Obrazowy przykład enkapsulacji danych.
Źródło: Opracowanie własne.

W powyższym przykładzie krystalizują się wszystkie poprzednio ustalone priorytety enkapsulacji obiektowego paradygmatu programowania. Klasa służąca dodaniu dwóch liczb naturalnych posiada jedno prywatne pole jako kontener dla wyniku. Ponadto obecne są dwie metody, o dostępie kolejno prywatnym i publicznym. Metoda *Compute()* dostępna tylko z poziomu jej klasy to wewnętrzna implementacja głównego algorytmu tej klasy, słusznie ukrytego przed zewnętrznymi modułami, co zapobiega modyfikacji sposobu obliczania wyniku. Publiczna metoda *GetValue()* to interfejs klasy dla zewnętrznego użytkownika, która wywołuje metodą prywatną oraz zwraca jej wynik. Dzięki temu rozwiązanie zaimplementowane w klasie *Add* jest w pełni zabezpieczone, z zachowaniem dostępności efektywnego wykorzystania jej funkcjonalności. Zaprojektowanie klasy w ten sposób dostarcza jej użytkownikowi możliwość utworzenia obiektu, oraz wykorzystania publicznej metody *GetValue(),* a więc wszystko co niezbędne do skutecznego używania tego typu.

[50] B. Meyer: *Programowanie zorientowane obiektowo*, Gliwice 2005, s. 207.

3. REALIZACJA PARADYGMATU PROGRAMOWANIA OBIEKTOWEGO NA PRZYKŁADZIE WYBRANYCH JĘZYKÓW

Obszerność i rozgałęzienie paradygmatu programowania obiektowego oraz jego ewolucja na przestrzeni lat doprowadziły do różnorakich implementacji tego popularnego dziś wzorca w poszczególnych językach programowania. Wśród najbardziej rozpowszechnionych obecnie języków programowania, mianujących się jako typowo obiektowe, często okazuje się, że po części wpisują się one także w programowanie funkcyjne, strukturalne, a także wprowadzają własne modyfikacje paradygmatu obiektowego. Owe poczynania doprowadziły do powstania nieoficjalnego, jeszcze rzadko dziś używanego terminu „języków hybrydowych", które przodują wśród komercyjnych usług, i do których to z pewnością należy przyszłość programowania aplikacji.

W ostatnim, trzecim rozdziale pracy dokonano skrupulatnego porównania implementacji paradygmatu programowania obiektowego w wybranych trzech językach - C#, JavaScript oraz SmallTalk. Na podstawie otrzymanych różnic wysnuto odpowiednie wnioski, które przedstawiono na końcu rozdziału.

3.1. Alokowanie obiektów w pamięci

3.1.1. C#

Jak sama nazwa omawianego paradygmatu wskazuje, każdy język posiadający jego cechy operuje na materiach zwanych obiektami. Jak przystało na technologie informatyczne, każdy obiektowy język stosuje inną technikę zapisywania obiektów do pamięci programu. Według A. Troelsena i P. Japikse, w języku C# platformy .NET instancja obiektu wygląda tak[51]:

```
Type objectName = new Type();
```

[51] A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework*, SSBM Finance Inc., New York 2015, s. 133.

Utworzenie obiektu rozpoczynamy od wybrania typu obiektu (*string, int, double itd.*), a następnie unikatowej nazwy obiektu, którą będziemy się posługiwać w dalszej części kodu źródłowego. Za znakiem równości natomiast, używa się słowa kluczowego *new,* które wywołuje konstruktor obiektu. Pusty nawias oznacza wywołanie domyślnego konstruktora obiektu, bez inicjalizacji jego pól. Istnieje także możliwość przekazania do konstruktora niestandardowego (jeśli istnieje) argumentów wypełniających pola obiektu.

3.1.2. JavaScript

Język JavaScript udostępnia programiście kilka różnych możliwości utworzenia nowego obiektu. Jako, że ten webowy język skryptowy nie posiada jako takich klas, podczas tworzenia obiektu, konieczne jest zdefiniowanie żądanych pól obiektu.

```
var object = new Object();
```

Podobnie jak w C#, należy wywołać konstruktor obiektu, z tą różnicą, iż dynamiczność typowania w JavaScript nie wymusza na użytkowniku jawnej deklaracji typu obiektu – stąd wyrażenie *var*. Oprócz klasycznego zapisania nowego obiektu, istnieje możliwość skorzystania z tzw. dosłownej notacji (*literal notation*):

```
var object = { a: 23, b: „text”, c: {} };
```

Wyżej przedstawiony przykład umożliwia jednoczesną inicjalizację pól tworzonego obiektu, poprzez podanie nazwy pola w postaci dowolnej nazwy oraz jego wartości, której typ zostanie automatycznie nadany dzięki dynamicznemu typowaniu[52].

3.1.3. SmallTalk

Język SmallTalk jako jeden z pierwszych podejść w pełni obiektowych do programowania, oferuje jedną opcję alokacji obiektu w pamięci maszyny wykonującej.

```
Variable1 := Stack new
```

[52] J. Duckett: *Effective JavaScript: 68 Specific Ways To Harness The Power Of JavaScript*, Addison-Wesley, New Jersey 2015, s. 87.

Powyższy przykład pokazuje przypisanie do zmiennej o nazwie *Variable1* obiektu w postaci kolejki o regulaminie *Last In First Out (Stack)*. Wartość obiektu przypisywanego do zmiennej obliczana jest w wyniku wyrażenia stojącego po prawej stronie operatora przypisania w SmallTalk, czyli znaku dwukropka i równości[53].

3.2. Definicje klas

3.2.1. C#

Definiowanie klas w języku C# najczęściej nie stwarza problemów i zagwozdek programistom, ponieważ specjaliści firmy Microsoft uczynili ten proces wysoce intuicyjnym, dlatego też poniższy przykład zawiera kilka urozmaiceń.

```
sealed class User : UserTemplate { }
```

W celu deklaracji nowej klasy wystarczy użycie słowa kluczowego *class* i nazwy klasy, oraz umieszczenie jej ciała w nawiasach klamrowych. W tym przypadku dzięki słowu kluczowemu *sealed* sygnatura tworzy klasę zapieczętowaną (beż możliwości dziedziczenia po niej) oraz pobiera ona cechy klasy *UserTemplate* właśnie poprzez dziedziczenie. Przy deklaracjach klas mogą się pojawić także inne wyrażenia jak *abstract, partial* czy modyfikatory dostępu[54].

3.2.2. JavaScript

Zgoła odmienne podejście do żonglowania obiektami w kodzie programu preferuje język JavaScript, a główną cechą charakterystyczną tego rozwiązania jest całkowity brak klas w tym języku. Zamiast tego konieczne jest zdefiniowanie odrębnej funkcji-konstruktora dla obiektów, która to działa jak standardowe konstruktory klasowe obiektów z języków takich jak Java, C#, C, C++; z tą różnicą, iż konstruktor w JavaScript istnieje samodzielnie (nie potrzebuje kontenera - klasy)[55].

---

[53] K. Beck: *SmallTalk Best Practice Patterns,* Prentice-Hall Inc., New Jersey 1997, s. 44.

[54] A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework*, SSBM Finance Inc., New York 2015, s. 154, 170-173.

[55] V. Antani: *Mastering JavaScript*, Packt Publishing, Birmingham, s. 86-87.

```
function Object(ID, name) {
      this.ID = ID;
      this.name = name;
}
```

Powyższy przykład prezentuje funkcję w języku JavaScript, pełniącą rolę konstruktora obiektu o typie *Object.* Konstruktor inicjalizuje dwa pola obiektu: *ID* oraz *name* wartościami przesłanymi do tej funkcji. Instancja obiektu wykonana za pomocą powyższego konstruktora wygląda tak jak na następującym przykładzie:

```
var myObject = new Object(01, 'master');
```

Powyższy wzorzec prezentuje używanie konstruktorów obiektów w JavaScript. W ten sposób powstaje obiekt typu *Object* utworzony za pomocą funkcji-konstruktora o tej samej nazwie. Obiekt zawiera w jawnie sobie dwa pola: *ID* – wypełnione przez konstruktor przesłaną wartością 01, *name* – zainicjalizowane na słowo *master* oraz niejawnie **prototyp** funkcji, który swoje zastosowanie znajduje w dziedziczeniu.

3.2.3. SmallTalk

W SmallTalk'u nowe klasy muszą być zdefiniowane jako podklasy klas już istniejących, a wyjątkiem jest predefiniowana systemowa klasa *Object,* po której tak naprawdę dziedziczą wszystkie tworzone przez użytkownika klasy oraz te predefiniowane w systemie[56]. Poniższy zapis przedstawia wzór, którego znajomość jest wymagana do poprawnej deklaracji nowej klasy w języku SmallTak:

```
base-class-name subclass: #class-name
```

Na początku należy wskazać nazwę klasy bazowej wraz ze słowem *subclass,* a następnie po dwukropku podać nazwę tworzonej klasy wraz ze znakiem *#.* Jak wcześniej wspomniano, w SmallTalk'u wszystkie klasy dziedziczą składowe bazowej, predefiniowanej klasy *Object,* dlatego podczas deklaracji nowej, samodzielnej klasy należy jawnie wskazać nadklasę właśnie jako *Object:*

```
Object subclass: #DataBaseConnection
```

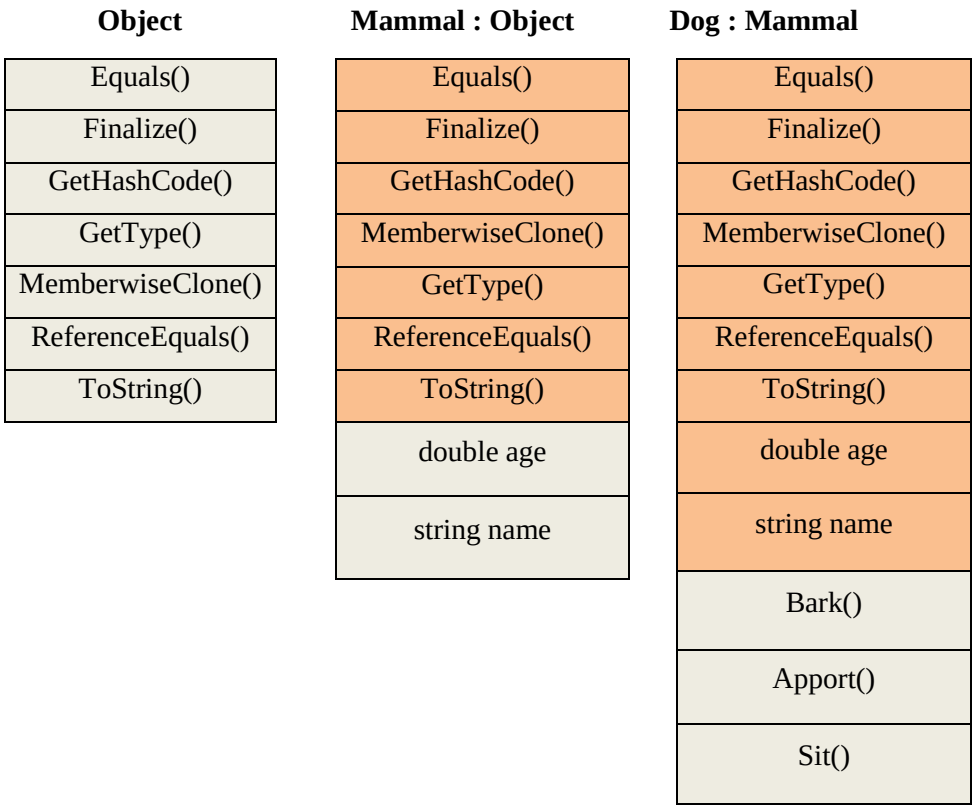
[56] S. Lewis: *The Art and Science of SmallTalk,* Prentice Hall, Hertfordshire, s. 61.

Powyższa dyrektywa przedstawia sygnaturę przykładowej klasy obsługującej połączenie z bazą danych, która zgodnie z zasadami SmallTalk'a jawnie dziedziczy z klasy *Object.*

3.3. Dziedziczenie

3.3.1. C#

C# wykorzystuje znane wśród języków obiektowych podejście do dziedziczenia oparte na bazowej klasie *Object*, z której niejawnie dziedziczą wszystkie klasy: zarówno predefiniowane oraz te niestandardowe, tworzone przez deweloperów. Każda klasa definiowana na platformie .NET może dziedziczyć członków tylko jednej klasy bazowej, jednak dziedziczenie jest przechodnie[57]. W praktyce oznacza to, że kiedy klasa C wywodzi się z klasy B i klasa B jest podklasą klasy A, to klasa C dziedziczy składowe zadeklarowane w klasach B i A. Poniższa grafika przedstawia mechanizm dziedziczenia w C#:



[57] C. Nagel: *C# 6.0 and .NET Core 1.0*, John Wiley & Sons, Indianapolis, s. 239-240.

Rysunek 3.1: Przykład dziedziczenia w C#.
Źródło: Opracowanie własne.

Jak wynika z powyższego zestawienia, główna klasa całej platformy .NET zawiera kilka natywnych metod, które odpowiadają np. za zwrócenie typu obiektu (*GetType()*), nazwy obiektu *(ToString())*, czy wskazanie referencji do obiektu (*ReferenceEquals()*). Siłą rzeczy każda klasa napisana w języku C# będzie niejawnie zawierała w sobie te metody. Klasa *Mammal* (ssak) dziedziczy po *Object* wszystkie to funkcjonalności, oraz definiuje pole wiek i nazwę. Następnie z klasy *Mammal* wyprowadzona została klasa *Dog*, definiująca psa. Zawiera ona metody z klasy *Object*, *Mammal*, a także wprowadza umiejętności każdego psa: szczekanie, aportowanie i siadanie.

Abstrakcyjne klasy bazowe

Rola klas bazowych w rozbudowanych systemach informatycznych, często ogranicza się do definiowania jedynie wspólnych składowych dla klas pochodnych, pomijając bezpośrednie ich wykorzystywanie. Z tego powodu język C# dostarcza **abstrakcyjne klasy bazowe**, które różnią się od zwykłych klas tym, iż nie ma możliwości wywołania instancji takiej klasy[58]. Ograniczenie to nabiera sensu biorąc pod uwagę przykładową hierarchię klas z podrozdziału 2.3 traktującego o polimorfizmie. Instancja klasy *Shape* nie ma żadnego zastosowania w systemie oprócz dostarczenia do potomków odpowiednich składowych, dlatego dobrą praktyką jest oznaczenie tej klasy jako abstrakcyjnej.

Jak twierdzi John Sharp, specjalista Microsoft, abstrakcyjne klasy bazowe w języku C# definiuje się w sposób ukazany na poniższym przykładzie[59]:

```
abstract class Shape
{
      //members…
}
```

[58] C. Nagel: *C# 6.0 and .NET Core 1.0*, John Wiley & Sons, Indianapolis, s. 266.

[59] J. Sharp: *Microsoft Visual C# Step by Step. Eight Edition*, Microsoft Press, Redmond, s. 178.

Jak wynika z powyższego, aby klasa stała się abstrakcyjną, bez możliwości utworzenia jej obiektu, należy dodać przed dyrektywą *class* słowo kluczowe *abstract.* W języku C#, oprócz klas, abstrakcyjne mogą być też metody.

Interfejsy

Z definicji interfejsy w .NET to „nazwany zbiór abstrakcyjnych składowych", wymuszający zachowanie na klasach lub strukturach, w których jest implementowany. W owym zbiorze mogą się znaleźć jedynie sygnatury metod oraz właściwości, bez modyfikatorów dostępu (wszystkie domyślnie są publiczne)[60]. Poniżej znajduje się kod predefiniowanego interfejsu w bibliotekach .NET o nazwie *IDisposable,* jeden z najbardziej popularnych, implementujący w klasie lub strukturze sprzątanie pamięci po utworzonym obiekcie tej klasy[61]:

```
interface IDisposable
{
    Dispose();
}
```

W kulturze programistów platformy .NET istnieje niepisana zasada, aby interfejsy nazywać z wielkiej litery „I", aby uniknąć pomyłki z klasą przy implementacji, co rozjaśnione jest w przykładowej jego implementacji poniżej. Interfejs ten zawiera sygnaturę metody *Dispose(),* która usuwa z pamięci aktualny obiekt swojej klasy. Jak wspomniano wcześniej, istnieje także możliwość zawarcia dowolnej właściwości w polu interfejsu. Celem przejścia do konkretów, poniżej widnieje sposób implementacji interfejsu w klasie:

```
class DBConnection : IDisposable
{
   //Members…
   public void Dispose()
   {
        this.Dispose();
```

[60] A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework,* SSBM Finance Inc., New York 2015, s. 154, 280-282.

[61] A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework,* SSBM Finance Inc., New York 2015, s. 154, 284.

```
        }
    }
```

Z powodu takiego samego sposobu implementacji interfejsu w klasie, jak i zapisu dziedziczenia, przed nazwą interfejsu stosuje się wielką literę „I". Kiedy klasa implementuje interfejs, koniecznym jest zdefiniowanie ciała wszystkich funkcji i właściwości mieszczących się w tym interfejsie – przeciwnie zostanie zgłoszony błąd już na etapie kompilacji programu. Implementowanie interfejsów w klasach pozwala nie tylko na dodawanie do nich nowych funkcji ale przede wszystkim traktowanie ich jako instancji tego interfejsu, co pozwala np. dodać do generycznej kolekcji wszystkie klasy implementujące określony zbiór zachowań. Interfejsy to w pewnym sensie również rekompensata ograniczonego dziedziczenia na platformie .NET, bo tylko pojedyncze, ponieważ klasa lub struktura może implementować **nieograniczoną liczbę interfejsów**[62]**.**

3.3.2. JavaScript

Jak każdy szanujący się język programowania, który w choć małym stopniu realizuje paradygmat obiektowy, JavaScript również dostarcza mechanizm dziedziczenia w swoich strukturach, pomimo, iż nie zawiera, wydawać by się mogło, jednego z wyznaczników obiektowego podejścia, a mianowicie klas. Zamiast tego, JavaScript oferuje swój własny sposób na tworzenie hierarchii obiektów oparty na **prototypach** *(prototype)*[63].

Prototypy

Prototyp to domyślnie i automatycznie dodawany „kontener", właściwość czy składowa do każdego obiektu tworzonego za pomocą konstruktora, a dzięki nim istnieje możliwość szybkiego dodawania nawet do istniejących już typów nowych pól oraz funkcji. Z racji, iż w JS większość materii jest obiektem, prototyp jest dodawany zarówno do klasycznych obiektów jak i do funkcji[64]. Przykładowo, możliwe jest dodanie do predefiniowanego w języku JavaScript typu *String* swojej własnej metody

[62] J. Albahari, B. Albahari: *C# in a Nutshell,* O'Reilly Media Inc., Sebastopol, s. 252.

[63] D. Crockford: *JavaScript: The Good Part,* O'Reilly Media Inc., Sebastopol, s. 197.

[64] D. Crockford: *JavaScript: The Good Part,* O'Reilly Media Inc., Sebastopol, s. 201.

poprzez prototyp, co poskutkuje posiadaniem jej przez wszystkie kolejno tworzone obiekty tego typu. Celem zobrazowania tego systemu poniższy listing zawiera dodanie nowej właściwości do konstruktora języka JavaScript z podrozdziału 3.2.2.:

```
Object.prototype.color = „blue”;
var myObject = new Object();
var color = myObject.color;
```

W ten sposób do obiektu tworzonego za pomocą konstruktora *Object* została dodana właściwość przetrzymująca kolor obiektu bez modyfikacji wcześniej utworzonego kodu konstruktora. Jak łatwo się domyśleć, zmienna *color* z powyższego przykładu będzie zawierała wartość *blue* nowoutworzonego obiektu *myObject.*

Dziedziczenie przez prototypowanie

Zastosowanie prototypów nie kończy się na zaprezentowanym wyżej przykładzie, ponieważ przede wszystkim, odgrywają one ważną rolę w tworzeniu hierarchii obiektów w języku JavaScript. Prototyp obiektu może być wykorzystany do wyprowadzenia nowego obiektu z już istniejącego, co odbywa się za pomocą predefiniowanych funkcji *create()* oraz *getProtoypeOf()*[65]. Poniższy przykład ilustruje wyprowadzenie nowego obiektu z używanego już wcześniej w tym podrozdziale obiektu *myObject:*

```
    var customizedObject =
Object.create(Object.getPrototypeOf(myObject), {
    “size” :{value: “medium”}
    });
```

Poprzez wykorzystanie dwóch domyślnych funkcji głównego obiektu w JavaScript, czyli *create()* i *getPrototypeOf()* wyprowadzony został nowy typ z *myObject.* Przedstawiony kod źródłowy przypisuje do zmiennej *customizedObject* nowy typ, utworzony poprzez załadowanie prototypu obiektu nadrzędnego (myObject). *CustmizedObject* obiekt zawiera zarówno właściwości obiektu, z którego dziedziczy, więc *ID* oraz *name,* a także nowe pole o nazwie *size* zainicjowane na wartość *medium.*

[65] J. Duckett: *JavaScript & JQuery: Interactive Front-End Web Development*, John Wiley & Sons Inc., Indianapolis, s. 137.

Choć dziedziczenie w JavaScript nie jest tak przejrzyste i intuicyjne jak np. w przedstawionym wcześniej C#, jest to pełnoprawny mechanizm służący w tym języku do tworzenia hierarchii obiektów oraz wielokrotnego wykorzystania kodu. Powodem, dla którego dziedziczenie w JS można nazwać okrojonym jest również przeznaczenie tego języka, które nie skupia się na tworzeniu skomplikowanych logicznie niskopoziomowych struktur systemów, ale ma służyć dodawaniu funkcjonalności do wierzchnich warstw aplikacji webowych, oraz czynić je *user-friendly,* jak wspomniano w pierwszym rozdziale tego opracowania dokładnie charakteryzującym JavaScript.

3.3.3. SmallTalk

Siłą rzeczy dziedziczenie w SmallTalk'u zostało przedstawione w podrozdziale opisującym deklarowanie klas w tym języku, ponieważ każda z nich wymaga wskazania klasy nadrzędnej dla definiowanej. Również w przypadku implementacji samodzielnej klasy, jako rodzica należy wskazać główną klasę bazową systemu SmallTalk – *Object*[66]. Poniższy listing przedstawia przykład nieskomplikowanej klasy w języku SmallTalk:

```
Object subclass: #User
       instanceVariableNames:
              'ID nickName password'
       classVariableNames:
              'userType'
! User class methods !
```

[66] J. Pletzke: *Advanced SmallTalk,* Addison-Wesley, New Jersey, s. 86.

```
newID: a newNick: b newPass: c
                | user |
                user := user new.
                user ID: a.
                user nickname: b
                user password: c
                ^user
```

Listing 3.1: Przykładowa klasa języka SmallTalk.
Źródło: Opracowanie własne.

Zgodnie z wymogami omawianego języka programowania, utworzona klasa obiektów *User* jest podklasą głównej klasy *Object,* co wyraża pierwsza linia powyższego zapisu. Następnie w klasie zawarto zmienne instancyjne i klasowe dla obiektów W tym przykładzie zmienne instancyjne to *ID, nickName* oraz *password,* z kolei zmienna klasowa to *userType.* Klasa posiada również jedną metodę klasową, która tworzy nowy obiekt oraz inicjalizuje jego zmienne. Sygnatura tej metody informuje w jak wygląda sposób jej wywoływania: należy podać kolejno wartość zmiennej *ID, nickname* oraz *password.* Wewnątrz metody jest zadeklarowana tymczasowa zmienna o nazwie *user,* której kolejne pola są inicjalizowane zmiennymi przesłanymi do metody. Ostatecznie funkcja zwraca do procedury wywołującej nowy obiekt za pomocą znacznika ^. Oprócz zadeklarowanych składowych w klasie *User,* znajdują się tam niejawnie zmienne oraz metody otrzymane z klasy *Object.*

Ponieważ SmallTalk, zaraz po języku Simula jest jednym z prekursorów implementacji paradygmatu obiektowego, reprezentuje on mocno klasyczne podejście do dziedziczenia[67]. Deklaracja klasy dziedziczącej modelowo realizuje wzór przedstawiony w rozdziale II tego opracowania, a także przypomina technikę stosowaną w C#. Oprócz metod wirtualnych oraz standardowych, warto wspomnieć o charakterystycznych dla SmallTalka **zmiennych klasowych i instancyjnych.**

Typy klasowe i typy instancyjne

[67] J. Pletzke: *Advanced SmallTalk,* Addison-Wesley, New Jersey, s. 79.

SmallTalk pozwala na deklarowanie w ciele swoich klas dwóch rodzajów zmiennych: klasowych oraz instancyjnych, co bardziej doświadczonym programistom pozwala wskazać różnice między nimi już na podstawie samych nazw. Zmienne instancyjne są prywatną pamięcią każdego pojedynczego obiektu tworzonego z klasy je posiadającej, co oznacza, że ich wartości mogą być całkowicie różne w różnych obiektach. Deklaracja zmiennych instancyjnych w klasie wymaga użycia dyrektywy *instanceVariableNames:,* oraz wymienienia kolejno pożądanych nazw zmiennych[68].

Z kolei zmienne klasowe przetrzymują informacje wspólne dla wszystkich obiektów danej klasy. Jest to więc część wspólna pamięci dla obiektów z tej samej rodziny i jednocześnie przeciwieństwo zmiennych instancyjnych. Zarówno zmienne instancyjne jak i klasowe podlegają podstawowym prawom dziedziczenia: każda klasa potomna otrzyma od swojej nadklasy zadeklarowane w niej oba rodzaje zmiennych. Warto zaznaczyć, iż zmienne klasowe będą wspólne dla wszystkich obiektów również w górę hierarchii klas[69].

Ponieważ w SmallTalku „wszystko jest obiektem", oprócz istnienia dwóch rodzajów zmiennych, istnieje również taka sama potrzeba dla metod. Metody klasowe są wywoływane i wykonywane w dziedzinie klas. Metoda z listingu numer 3.2 jest metodą klasową i wywołuje się ją poprzez wysłanie wiadomości do klasy jako obiektu (wszystko jest obiektem), beż konieczności posiadania obiektu utworzonego w tej klasie. Natomiast metody instancyjne, analogicznie do zmiennych instancyjnych, działają obiektach utworzonych z innych obiektów (klas)[70]. Gdyby klasa *User* zawierała metodę instancyjną, nie byłoby możliwości wywołania jej dla klasy, a jedynie dla obiektu zwracanego przez jedyną obecną tu metodę klasową.

[68] J. Hunt: *SmallTalk and Object Orientation: An Introduction*, JayDee Technology Ltd., Wiltshire, s. 104.

[69] J. Hunt: *SmallTalk and Object Orientation: An Introduction*, JayDee Technology Ltd., Wiltshire, s. 107.

[70] S. Lewis: *The Art and Science of SmallTalk*, Prentice Hall, Hertfordshire, s. 97-98.

3.4. Polimorfizm.

3.4.1. C#

Według informacji zawartych w drugim rozdziale tego opracowania, polimorfizm pozwala podklasie zdefiniować własne wersje dziedziczonych metod, pod warunkiem, iż wcześniej zostały one zaimplementowane z możliwością nadpisywania. Platforma .NET w języku C# dostarcza dwa sposoby deklarowania metod jako nadpisywalnych w klasie dziedziczącej, a odbywa się to za pomocą dwóch słów kluczowych: *abstract* i *virtua*[71]*l.*

Kiedy metoda w klasie bazowej zostanie opieczętowana wyrażeniem *virtual*, daje to opcjonalną możliwość przesłonienia jej działania w klasie potomnej. Oznacza to, że klasa potomna otrzymuje metodę z implementacją bazową oraz możliwością jej przesłonienia. Najczęściej stosowaną praktyką jest przesłanianie metod wirtualnych, poprzez wywoływanie bazowej funkcjonalności oraz dodawanie nowej, pożądanej specjalnie dla klasy dziedziczącej. Z kolei słowo kluczowe *abstract* może odnosić się tylko do sygnatur metod zdefiniowanych w klasach abstrakcyjnych. Jeśli klasa abstrakcyjna w języku C# posiada jakąkolwiek sygnaturę metody opatrzoną słowem *abstract* wymusza to na klasie potomnej implementację ciała tej metody – w przeciwnym razie zostanie zgłoszony błąd już na etapie kompilacji programu[72].

Poniżej zaprezentowano poprawne korzystanie z wirtualnych i abstrakcyjnych metod języka C#:

```
Abstract class Person
{
    public virtual void GetInfo()
    {
        //base implementation…
    }
    public abstract void DoJob();
}
```

[71] J. Sharp: *Microsoft Visual C# Step by Step. Eight Edition*, Microsoft Press, Redmond, s. 229.

[72] C. Nagel: *C# 6.0 and .NET Core 1.0*, John Wiley & Sons, Indianapolis, s. 286.

```
Class Student : Person
{
    Public override void GetInfo()
    {
        base.GetInfo()
        //new implementation…
    }
    public override void DoJob()
    {
        //new implementation…
    }
}
```

Listing 3.2: Używanie metod wirtualnych i abstrakcyjnych w języku C#.
Źródło: Opracowanie własne.

Listing numer 3.1 obrazuje zasadę działania metod wirtualnych i abstrakcyjnych stosowanych w języku C#. Abstrakcyjna klasa *Person* posiada wirtualną metodę *GetInfo(),* która może, lecz nie musi, zostać przesłonięta w klasie potomnej, która tę funkcjonalność otrzyma. Oprócz tego w klasie znajduje się sygnatura abstrakcyjnej metody *DoJob()*, która nie posiada implementacji (nie ma takiej możliwości w przypadku metod abstrakcyjnych), oraz musi zostać zaimplementowana w ewentualnej podklasie.

Podklasa dla *Person* to utworzona w tym samym przykładzie klasa *Student*, która odpowiednio użytkuje otrzymane metody. Wirtualna odziedziczone funkcja *GetInfo()* została prawidłowo oznaczona słowem kluczowym *override,* co pozwoliło na dokonanie zmian w jej implementacji. W tym przypadku wpierw zostaje wywołana bazowa funkcjonalność metody *GetInfo()* (z klasy *Person),* po czym pozostaje możliwość dopisania szczególnej implementacji dla klasy *Student.* Nie ma konieczności wywoływania bazowej implementacji funkcji w procesie przesłaniania metod wirtualnych, jednak jest to najczęściej spotykana praktyka. Abstrakcyjna metoda *DoJob()* również została nadpisana, a właściwie zaimplementowana, z racji braku jej ciała w klasie bazowej – przeciwnie kompilacja programu nie powiodłaby się.

3.4.2. JavaScript

Polimorfizm w języku JavaScript nie został tak jawnie i dosłownie zaimplementowany jak np. w omówionym wcześniej C#, co oznacza, że nie spotkamy w jego strukturach np. dedykowanych słów kluczowych służących realizacji polimorfizmu, czy automatycznych uzupełnień składni przez środowisko programistyczne. Pomimo tego, polimorfizm w języku JS jest obecny i podobnie jak dziedziczenie został zaimplementowany w oparciu o prototypy[73].

Jako sposób wyrażenia idei polimorfizmu w JavaScript posłuży poniższy listing napisany w tym języku:

```
function Car(x) {
   this.name = x;
   this.showInfo = function() {
       document.writeln("Car: ", this.name);
  };
}
function SportCar(y) {
   this.prototype = new Car(y);
   this.prototype.turbo = true;
   this.prototype.showInfo = function() {
        document.writeln("SportCar: ", this.name,
        "Turbo: ", this.turbo);
  };
}
var myCar = new Car("Mercedes");
var mySportCar = new SportCar("McLaren");
myCar.showName();
mySportCar.showName();
```

Listing 3.3: Polimorfizm w JavaScript.
Źródło: Opracowanie własne.

[73] M. Haverbeke: *Eloquent JavaScript: A Modern Introduction to Programming*, No Starch Press Inc., San Francisco, s. 254-255.

Kod źródłowy języka JavaScript przedstawiony na listingu numer 3.3 składa się z dwóch konstruktorów obiektów *Car* i *SportCar*, dyrektyw deklarujących instancje tych obiektów oraz wywołania wewnętrznych metod tych obiektów. Pierwszy prosty konstruktor listingu tworzy obiekt typu *Car* z jedną właściwością przechowującą nazwę obiektu, oraz metodę wyświetlającą w przeglądarce nazwę obiektu wraz z jego typem. Kolejny, ciekawszy konstruktor obiektu *SportCar* podejmuje pierwsze kroki w stronę zobrazowania mechaniki polimorfizmu w JavaScript. Do prototypu obiektu zostaje dodany poprzedni obiekt *Car*, co oznacza, że *SportCar* przejmuje wszystkie jego składowe poprzez dziedziczenie. Ponadto obiekt otrzymuje nową, niejawnie typizowaną zmienną *boolean*, opisującą turbodoładowanie samochodu, mogącą przyjmować dwie wartości: *true* lub *false*. Kolejna dyrektywa konstruktora obiektu *SportCar* obrazuje polimorfizm języka JavaScript w czystej postaci. Poprzez dodanie do prototypu obiektu *Car* otrzymał on dostęp do wszystkich składowych włącznie z metodą *showInfo()*, która jednak nie wyświetla informacji o tym, czy auto posiada turbo, czy też nie. Dlatego niezbędnym było przedefiniowanie metody, dodając do jej implementacji uwzględnienie właściwości *turbo*. Dzięki temu, końcowa część skryptu z listingu 3.3 poprawnie alokuje dwa nowe obiekty poszczególnych typów, oraz ilustruje zasadę działania polimorfizmu w języku JavaScript. Po wywołaniu powyższego kodu np. w przeglądarce internetowej Google Chrome oczom programisty ukaże się poniższy zapis:

```
Car: Mercedes
SportCar: McLaren, Turbo: true
```

Dzięki nadpisaniu metody obiektu *Car* możliwe jest wyświetlenie wszystkich pożądanych przez dewelopera właściwości obiektu *SportCar*. Zabieg ten zapewnił również używanie identycznej nazwy metody dla tej samej rodziny obiektów, co nie powoduje zaciemnienia kodu źródłowego, oraz zdecydowanie upraszcza korzystanie ze składowych tych dwóch typów.

3.4.3. SmallTalk

Polimorfizm rozumiany jako możliwość nadpisywania ciała dziedziczonych metod, nie mógł zostać pominięty w języku postrzeganym jako prekursor obiektowego stylu programowania. SmallTalk również zawiera mechanizmy wspierające

przesłanianie dziedziczonych funkcji, jednak nie w tak rozbudowanym stopniu jak młodsze języki obiektowe, które poniekąd wyrosły na technologii wbudowanej w system SmallTalk. Podczas gdy obecnie wykorzystywane języki programowania obiektowego dostarczają m.in. specjalnych słów kluczowych wspierających polimorfizm (np. *overridable* w Visual Basic), czy kilku sposobów przesłaniania dziedziczonych metod - w SmallTalk'u przesłanianie funkcjonalności klasowych odbywa się w pewnym sensie niejawnie, ponieważ brak jest tego typu udogodnień[74]. Z uwagi na swoiste wybrakowanie składni SmallTalk'a pod tym względem, język ten realizuje idee polimorfizmu w najbardziej podstawowej wersji, co wbrew pozorom nie stanowi przeszkody w drodze do efektywnego wykorzystania tego filaru programowania obiektowego.

Następujący listing to zobrazowanie zastosowania polimorfizmu w języku SmallTalk:

```
Object subclass: #Person
     instanceVariableNames: 'name age sex'
     classVariableNames: ''
     poolDictionaries: ''

  initialize: n, a, s
       | person |
       person.name := n
       person.age := a
       person.sex := s
       ^person

Person subclass: #Student
     instanceVariableNames: 'faculty ID'
     classVariableNames: ''
     poolDictionaries: ''

   initialize: n, a, s, f, ID
        | student |
```

[74] K. Beck: *SmallTalk Best Practice Patterns*, Prentice Hall Inc., New Jersey, s. 123-128.

```
student := super initialize: n, a, s
student.faculty := f
student.ID := ID
^student
```

Listing 3.4: Nadpisywanie funkcjonalnościw w SmallTalk'u.
Źródło: Opracowanie własne.

Celem przybliżenia mechaniki nadpisywania dziedziczonych metod, a zarazem polimorfizmu w SmallTalk, posłużono się dwoma, połączonymi relacją rodzic-potomek klasami. Pierwsza klasa reprezentuje osobę w wirtualnej rzeczywistości za pomocą kodu źródłowego. Zawiera ona trzy podstawowe właściwości opisujące obiekt: płeć, wiek oraz imię. Pośród właściwości znajduje się również jedna, obrazowa metoda, która inicjalizuje obiekt typu *Person* wraz z przesłanymi do niej wartościami pól, oraz zwraca do procedury wywołującej nowopowstały egzemplarz klasy. W kolejnej klasie *Student* zaobserwować można działanie przesłaniania metod bazowych. *Student* dziedziczy po *Person,* a więc zawiera wszystkie jej właściwości oraz metodę *initialize.* Dodatkowo dla osoby studenta zadeklarowano nowe pola w postaci wydziału akademickiego oraz ID. Jak wcześniej wspomniano klasa *Student* posiada wszystkie składowe swojej nadklasy, a więc także metodę *initialize.* Przesłonienie tej metody w podklasie polega na wywołaniu bazowej funkcjonalności za pomocą słowa kluczowego *super,* czyli przypisanie do zmiennej typu *Student* gotowego obiektu *Person* zwracanego przez bazową metodę *initialize.* Następnie zostały zainicjalizowane dodatkowe pola klasy *Person,* czyli *faculty* oraz *ID.* W ten sposób dwa obiekty różnego typu zwracają tę samą odpowiedź na wywołanie dla nich metody *initialize* – użytkownik otrzymuje gotowy obiekt z wypełnionymi polami.

3.5. Enkapsulacja

3.5.1. C#

C# jako stosunkowo młody język programowania wspiera enkapsulację, zarówno w postaci modyfikatorów dostępu jak i poprzez przesłanianie pól obiektów metodami dostępu.

Modyfikatory dostępu obecne do użytku w C# przedstawia poniższa tabela:

Modyfikator dostępu	Kompatybilność	Dostęp do elementu
public	typy i składowe	Brak ograniczeń
private	składowe i typy zagnieżdżone	Jedynie z poziomu klasy, lub struktury definiującej.
protected	składowe i typy zagnieżdżone	Możliwy w klasie definiującej, oraz w klasach potomnych.
internal	typy i składowe	Tylko wewnątrz definiującej biblioteki .NET.
protected internal	składowe i typy zagnieżdżone	Połączenie działania modyfikatorów *protected* i *internal*.

Tabela 2: Modyfikatory dostępu w C#.
Źródło: Opracowanie własne na podstawie: A. Troelsen, P. Japikse: *C# 6.0 and the .NET 4.6 Framework,* SSGM Finance Inc., New York 2016, s. 152.

Powyższe zestawienie charakteryzuje wszystkie cztery modyfikatory dostępu obecne w języku C# platformy .NET. Warto również wspomnieć, iż domyślnie, wszystkie składowe są **niejawnie prywatne**, co oznacza, że brak modyfikatora dostępu pieczętuje każdą składową jako prywatną – niedostępną na zewnątrz[75]. Pomimo tego, dobrą praktyką programowania jest jawne oznaczanie składowych jako prywatnych, co pozwala zmniejszyć ryzyko błędu, a zarazem polepszyć czytelność kodu źródłowego.

Przykładowo, obiekt którego zadaniem jest przekształcenie wartości temperatury powietrza wyrażonej w stopniach Celsjusza na stopnie Fahrenheita i odwrotnie, celem realizacji enkapsulacji, powinien dostarczyć użytkownikowi jedynie interfejs umożliwiający otrzymanie wyniku. Równocześnie sposób, oraz algorytm dokonujący przekształceń, nie powinny być dostępne dla procedury wywołującej. Klasę obiektu dokonującego tych obliczeń z zachowaniem enkapsulacji prezentuje listing 1.3:

```
class TempChange
{
    private double result;

    public double GetTempAsC(double far)
```

75

```
        {
            result = Calculate(far);
            return result;
        }

        private double Calculate(double far)
        {
            return (f - 32.0) * (5.0 / 9.0);
        }
    }
```

Listing 3.5: Klasa C# dokonująca przekształceń temperatury z zachowaniem zasad enkapsulacji danych.
Źródło: Opracowanie własne.

W powyższym przykładzie enkapsulacja wyraża się poprzez sygnowanie pola i metod klasy odpowiednimi modyfikatorami dostępu (*public* i *private*), oraz dzięki sprytnemu ukryciu algorytmu dokonującego obliczeń. Składniki, które dla zewnętrznego użytkownika obiektu nie będą widoczne, to pole *result* oraz metoda zawierająca wzór obliczeń *Calculate()*. Obiekt typu *TempChange* będzie udostępniał na zewnątrz jedynie metodę *GetTempAsC()*, która zwraca do procedury wywołującej już gotową wartość w stopniach Celsjusza. W ten sposób realizuje się tu wspomniane kilka akapitów wyżej upublicznienie interfejsu, pozwalającego na efektywne wykorzystanie obiektu bez zbędnych dodatkowych informacji.

Poprawne użycie obiektu *TempChange* wymaga od programisty znajomości:

- Sposobu utworzenia nowego obiektu tego typu,
- Formę przesyłania do obiektu danych wejściowych,
- Techniki przechwytywania danych zwracanych przez obiekt.

Wedle powyższych punktowań, prawidłowe korzystanie z obiektu typu *TempChange* widnieje na listingu 1.4:

```
static int Main(string[] params)
{
     double tempInFahrenheit = 100;
```

```
        TempChange myObject = new TempChange();
        double tempInCelsius =
        myObject.GetTempAsC(tempInFahrenheit);
        return tempInCelsius;
    }
```

Powyższy listing zawiera się w ciele metody *Main()*, która jest miejscem wejściowym wykonywania każdego programu w C#. Alokacja nowego obiektu typu *TempChange* odbywa się za pomocą słowa kluczowego *new*. Kluczowa akcja wykonywana w tym kodzie źródłowym to przypisanie do zmiennej *tempInCelsius* wartości zwracanej z metody *GetTempAsC()*, poprzez wywołanie jej dla utworzonego obiektu i przesłania argumentu w postaci zmiennej *tempInFahrenheit.* Zgodnie z poprzednimi założeniami, użytkownik obiektu nie jest świadom wewnętrznej **implementacji** algorytmu obliczającego, a pomimo to jest w stanie użyć obiekt z pozytywnym skutkiem.

3.5.2. JavaScript

Jak wynika z poprzednich części tego rozdziału, JavaScript implementuje fundamenty obiektowego paradygmatu programowania (dziedziczenie, polimorfizm), na swój własny, specyficzny sposób i nie inaczej jest z enkapsulacją w tym skryptowym języku programowania. JavaScript bowiem, nie dostarcza jawnie żadnych modyfikatorów dostępu dla deweloperów, choć i tak umożliwia definiowanie składowych zarówno publicznych jak i prywatnych. Brak modyfikatorów dostępu implikuje konieczność realizacji enkapsulacji za pomocą metod dostępu do składowych[76].

Zmienne definiowane w języku JavaScript widnieją w pamięci programu dopóty, dopóki istnieją funkcje, które się do nich odnoszą, co oznacza, iż zmienna zadeklarowana w konstruktorze będzie istnieć tak samo długo jak obiekt zawierający metody dostępu do niej. Najważniejszą jednak cechą wartości definiowanych wewnątrz obiektów, jest to, że są one **niejawnie prywatne**[77]. Celem dostępu do tych składowych

[76] V. Antani: *Mastering JavaScript*, Packt Publishing, Birmingham, s. 211.

[77] J. Duckett: *Effective JavaScript: 68 Specific Ways To Harness The Power Of JavaScript*, Addison-Wesley, New Jersey, s. 197.

implementuje się publiczne funkcje dostępu, które widoczne są z zewnątrz konstruktora.

Listing numer 23.3 powstał na podstawie wyżej przytoczonych zasad enkapsulacji w JavaScript:

```
function Check() {
      var member1 = "alfa";
      this.member2 = "beta";

      function metod1() { return member1 };
      this.method2 = function(){
                  return this.member2 + method1()};
      this.method3 = function(val){ member1 = val };
}
```

Listing 3.6: Przykład enkapsulacji danych w JavaScript.
Źródło: Opracowanie własne.

Przykładowy kod JavaScript widoczny powyżej składa się z konstruktora obiektu o nazwie *Check,* wewnątrz którego zdefiniowano kilka różnych składowych. Jak wcześniej wspomniano zmienne deklarowane wewnątrz konstruktorów są prywatne, więc to prawo dotyczy również zmiennej o nazwie *member1*. Kolejna linia kodu przypisuje do obiektu tworzonego w tym konstruktorze właściwość o nazwie *member2* z wartością tekstową „beta". Dostęp do tej właściwości jest dostępny z zewnętrznych modułów (dostęp publiczny), tak więc obiekt *Check* posiada składową prywatną oraz publiczną, na których operują kolejne metody w imię enkapsulacji.

Pierwsza z metod to klasyczna metoda dostępu do prywatnej zmiennej obiektu. Ponieważ nie ma możliwości bezpośredniego uzyskania wartości zmiennej *member1* spoza konstruktora, konieczne jest zdefiniowanie publicznej metody umożliwiającej zewnętrzny dostęp do wartości. Metoda o nazwie *method1* w bezpieczny sposób zwraca wartość składowej *member1* do procedury wywołującej. Kolejna funkcja udowadnia możliwość tworzenia metod dostępu przeznaczonych zarówno dla składowych prywatnych, jak i publicznych – jej implementacja pozwala na odczytanie stanu pól *member1* oraz *member2*. Natomiast ostatnia metoda konstruktora obiektu *Check* pozwala na ustawienie stanu prywatnego elementu *member1* poprzez przesłanie odpowiedniej wartości. Publiczna metoda przypisuje do prywatnej zmiennej wartość przesłaną do funkcji za pomocą jej argumentów.

Celem przetestowania skryptu z listingu 3.5 należy uruchomić kod, który przedstawia poniższy zapis:

```
var myObject = new Check();

var result1 = myObject.method1();
var result2 = myObject.method2();
myObject.method3("gama");
```

Po utworzeniu obiektu typu *Check* o nazwie *myObject* następuje wywołanie jego wewnętrznych metod. W drugiej linii kodu następuje przypisanie do zmiennej *result1* wartości zwracanej przez metodę *method1*, czyli wartość prywatnej zmiennej *member1* obiektu, co pozwala na odczytanie pozornie niedostępnej wartości obiektu. Następnie zmienna *result2* inicjalizowana jest zawartością dwóch pól obiektu: prywatnego i publicznego. Ostatnia dyrektywa zmienia wartość prywatnej składowej *member2* na zapis przesłany w argumencie funkcji, więc po wywołaniu tej metody, pole to będzie posiadało wartość „gama".

3.5.3. SmallTalk

W czasach, kiedy język SmallTalk był projektowany, a więc w latach osiemdziesiątych ubiegłego wieku, obiektowy paradygmat projektowania aplikacji komputerowych był dopiero w powijakach. Z tego głównie powodu, implementacja enkapsulacji danych, jednego z trzech głównych komponentów programowania obiektowego, nie doczekała się najnowszych rozwiązań i udogodnień tym języku. Konkretnie oznacza to, iż enkapsulacja danych w SmallTalk'u polega głównie na tworzeniu metod, których specjalnym przeznaczeniem jest zapewnienie bezpiecznego dostępu do danych. Ponadto SmallTalk nie posiada żadnych modyfikatorów dostępu, a wszystkie składowe klas czy to instancyjne, czy klasowe są niejawnie domyślnie prywatne – niedostępne dla zewnętrznych modułów[78]. Pomimo tych braków względem języków obiektowych, powstałych choćby już w latach dziewięćdziesiątych, SmallTalk zapewnia pełną enkapsulację danych w swoich strukturach.

[78] J. Hunt: *SmallTalk and Object Orientation: An Introduction*, JayDee Technology Ltd., Wiltshire, s. 160-163.

Poniższy listing przedstawia enkapsulację w SmallTalk'u za pomocą banalnej klasy punktu dwuwymiarowego układu współrzędnych:

```
Object subclass: #Point
     instanceVariableNames: 'x y'
     classVariableNames: ''
     poolDictionaries: ''

  getAposition
     ^x
  getBposition
     ^y
  setXposition: newX
     x := newX
  setYposition: newY
     y := newY
```

Listing 3.7: Klasa SmallTalk enkapsulująca dane.
Źródło: Opracowanie własne.

Listing numer 3.6 potwierdza wszystkie zasady używania enkapsulacji w obiektowym języku programowania SmallTalk. Klasa *Point* reprezentuje obiekt punktu na układzie współrzędnych, który posiada dwie współrzędne: x i y, co jest zamodelowane w postaci dwóch zmiennych instancyjnych. Jak wcześniej wspomniano zmienne klas są domyślnie niejawnie prywatne, więc celem zapewnienia możliwości odczytu wartości tych zmiennych zaimplementowane zostały odpowiednie metody dostępu w klasie *Point.* Pierwsze dwie metody pozwalają na odczyt wartości poszczególnych zmiennych, poprzez zwrócenie do procedury wywołującej ich wartości. Charakterystyczny w SmallTalk'u znak „^" oznacza tyle co popularne *return* w językach typu Java czy C#, co zwraca w tym przypadku wartość wybranej zmiennej. Oprócz tych dwóch metod dostępu, widnieją również metody dostarczające możliwość ustawienia liczby przypisanej do zmiennej *x* lub *y,* poprzez przesłanie nowej wartości do argumentu metody. Konieczność pisania metod dostępu w języku SmallTalk powoduje znaczny rozrost kodu źródłowego, oraz jego zaciemnienie, dlatego zaleca się unikanie nadużywania tej techniki, poprzez modelowanie funkcji tylko dla tych zmiennych, gdzie jest to niezbędnie obowiązkowe.

3.6. Wnioski

Przeprowadzona w tym rozdziale analiza implementacji paradygmatu obiektowego przez każdy z trzech wybranych języków wyłoniła wstępny obraz ich obiektowości, jednak nie dostarczyła jednoznacznego werdyktu, który z nich wpisuje się w ramy paradygmatu w najwyższym stopniu. Z tego powodu konieczna jest interpretacja wyników analizy, która przeprowadzona zostanie w tej części pracy.

Najbardziej podstawowy mechanizm, obowiązkowo obecny w każdym języku programowania mianującym się jako obiektowy to rzecz jasna tworzenie nowych obiektów. W tym aspekcie analiza nie dowiodła znaczących różnic między językami: we wszystkich, aby alokować w pamięci nowy obiekt, należy podać jego typ (za wyjątkiem JavaScript, gdzie typ ustalany jest automatycznie za pomocą dynamicznego typowania), unikatową nazwę oraz wywołać funkcję tworzącą obiekt. W tym ostatnim etapie C# i SmallTalk wywołują klasowy konstruktor obiektu, podczas gdy
w JavaScript może on istnieć samodzielnie. Oprócz tych kosmetycznych odmienności, wywoływanie obiektów nie różni się znacząco w tych trzech językach.

W przypadku definicji klas, od omawianej grupy szczególnie odstaje JavaScript, z racji, że brak jest w tym języku bezpośrednich sposobów deklarowania nowych klas. W tym webowym języku, obiekty konstruowane są tylko na podstawie wspomnianych już w poprzednim akapicie konstruktorów. Konstruktory zachowują się dokładnie tak jak klasy w C# i SmallTalk, i w odróżnieniu od konstruktorów tych dwóch języków, mogą przechowywać pola obiektów. Z kolei klasy w pozostałych dwóch językach tworzy się bardzo intuicyjnie. W C# wymagane jest właściwie podanie tylko nazwy klasy (pomijając szereg opcjonalnych modyfikacji). SmallTalk natomiast wymaga podania dla nowej klasy, jej rodzica, co może rodzić problemy podczas definiowania nowych, niestandardowych klas. W tym przypadku od klasycznej definicji obiektowości zdecydowanie najbardziej odbiega język JavaScript, podczas gdy C# pozostaje z nią w zgodzie w najwyższym stopniu.

Wszystkie wybrane języki poddawane analizie obowiązkowo wspierają dziedziczenie i tak jak w przypadku poprzednich implementacji obiektowości, istnieją na tym polu pewne różnice. C# oraz SmallTalk umożliwiają dziedziczenie wielopoziomowe, przy czym ten pierwszy zawiera dodatkowo mnóstwo opcjonalnych modyfikacji wpływających na dziedziczenie składowych. Mowa tutaj

np. o pieczętowaniu klas, deklarowaniu klas abstrakcyjnych, czy dziedziczeniu interfejsów. Celem dziedziczenia z nadklasy w języku C# należy jedynie wskazać w sygnaturze klasy potomnej po dwukropku jej nazwę. SmallTalk natomiast dostarcza klasyczne możliwości dziedziczenia składowych do podklas, poprzez wskazywanie nadklas i brak jest tutaj rozlicznych opcji modyfikacji otrzymywanej z nadklasy spuścizny. Warto przypomnieć, iż w języku SmallTalk nadklasę należy deklarować w każdej nowej definicji formy obiektu. Całkiem inaczej sprawa wygląda w języku JavaScript, gdzie dziedziczenie oparte jest na prototypowaniu, które jest znakiem firmowym tego języka. Prototypy realizują dziedziczenie z identycznymi skutkami jak robi to C# czy JS, jednak odbywa się to zgoła odmiennym sposobem. Zamiast dziedziczenia składowych bezpośrednio z klas, w JavaScript należy dodać do prototypu konstruktora pożądane elementy do przejęcia przez kolejny obiekt. W ten właśnie sprytny sposób JavaScript implementuje dziedziczenie bez używania klas w jakiejkolwiek formie.

Kolejny element obiektowego paradygmatu programowania omówiony w trzecim rozdziale opracowania, a mianowicie polimorfizm, również objawia się w wybranej puli języków. Najbardziej rozbudowaną mechanikę polimorfizmu z pewnością zawiera w sobie język C# platformy .NET. Metody abstrakcyjne, wirtualne o raz interfejsy tworzą z polimorfizmu w C# bardzo potężne i dające duże możliwości narzędzie. W przeciwieństwie do języka firmy Microsoft, sprawa ma się w SmallTalk'u. Ponieważ jest to dużo starszy język i implementacja w nim obiektowości była swego rodzaju eksperymentem, nie posiada on tak rozbudowanej infrastruktury polimorfizmu. W SmallTalku brak jest specjalnych słów kluczowych związanych z polimorfizmem, a metody przesłania się w nim w pewnym sensie niejawnie. Pomimo tego wybrakowania SmallTalk pozwala na efektywne wykorzystanie nadpisywania metod, jednak braki w składni tego języka mogą stworzyć problemy z czytelnością i klarownością kodu źródłowego. Polimorfizm w JavaScript oparty został, podobnie jak dziedziczenie, o prototypy. Przedefiniowanie ciała dziedziczonej metody w tym języku możliwe jest jedynie z poziomu prototypu. Poprzez kontener prototypowy, istnieje możliwość dodania nowej funkcjonalności do wybranej metody, bądź całkowite jej przeobrażenie. Podobnie jak z dziedziczeniem, język JavaScript implementuje polimorfizm na swój własny, oryginalny sposób, który mało ma wspólnego z tradycyjnym podejściem, a mimo to ten istotny komponent obiektowego paradygmatu programowania realizuje się z powodzeniem.

Jako ostatni z trzech fundamentów obiektowości, rozdział trzeci opisuje zasadę działania enkapsulacji. Podobnie jak w poprzednich przypadkach, najbardziej rozbudowany system enkapsulacji dostarcza język C#. Zawiera on szereg opisanych w trzecim rozdziale modyfikatorów dostępu, które współgrają z wieloma składowymi systemu jak klasy, metody, czy pola klas. Z kolei dwa pozostałe języki, czyli SmallTalk oraz JavaScript realizują ideę enkapsulacji w znacznie uboższy sposób. Brak jest w tych językach przede wszystkim modyfikatorów dostępu, co wymusza definiowanie specjalnych metod dostępu do składowych, które nie powinny być wystawiane na publiczny widok. Powoduje to zagrożenie zaciemnienia i zaśmiecenia kodu źródłowego systemów poprzez nadmiar metod dostępu, dlatego wymagane jest umiejętne ich rozplanowanie.

Powyższe dywagacje ujawniają klarowny obraz implementacji paradygmatu obiektowego przez C#, JavaScript oraz SmallTalk, a także obsadzają trzymiejscowe podium obiektowości wymienionymi językami. Poczynając od trzeciego miejsca, uplasował się na nim zdecydowanie najstarszy z branych pod lupę języków – SmallTalk. Ostatnie miejsce zajęte przez ten język, spowodowane jest przede wszystkim znaczną różnicą wieku pomiędzy SmallTalk'iem, a pozostałymi dwoma badanymi obiektami. Implementacja obiektowości w SmallTalk w latach osiemdziesiątych ubiegłego wieku była po w pewnym sensie eksperymentem, a także umotywowana chęcią przetestowania paradygmatu odzwierciadlającego świat rzeczywisty w kodzie źródłowym za pomocą klas i obiektów. Pomimo porażki SmallTalka w tym zestawieniu, kategorycznie nie wolno umniejszać jego roli w obiektowym tworzeniu aplikacji w latach osiemdziesiątych, a także należy podkreślić znaczący wkład w rozwój i popularyzację obiektowego paradygmatu programowania.

Na drugim, środkowym miejscu, uplasował się webowo-skryptowy język JavaScript. Jego specyficzna implementacja mechaniki dziedziczenia oraz polimorfizmu spowodowała niemałe trudności w ocenie tego podejścia do obiektowości. Choć system oparty o prototypy spełnia wszystkie podstawowe założenia paradygmatu obiektowego, nie jest to podejście, które w pełni wpisuje się w książkową definicję tego stylu programowania. Spośród wielu istniejących języków obiektowych, jedynie JavaScript realizuje dziedziczenie oraz przesłanianie metod za pośrednictwem prototypów, co powoduje, iż nawet doświadczony deweloper tradycyjnych języków obiektowych może napotkać przeszkody w dogłębnym zrozumieniu idei prototypów i ogólnej obietkowości w JavaScript. Owa specyficzność i unikalność języka JavaScript,

nie przeszkodziła mu jednak w osiągnięciu sukcesu na polu technologii informacyjnych, a dziś, przy wykorzystaniu licznych dedykowanych bibliotek dla tego języka (np. AngularJS od firmy Google) oraz w połączeniu z technologiami takimi jak HTML, CSS czy AJAX; JavaScript jest jednym z najważniejszych narzędzi tworzenia aplikacji przeglądarkowych.

Tryumfalne miejsce na podium obiektowości przypadło najmłodszemu językowi zestawienia, językowi, który powstał w firmie Microsoft i którego środowiskiem uruchomieniowym jest platforma .NET – mowa oczywiście o C#. Jako bezpośrednią przyczynę takiego wyniku wskazać można ogrom funkcjonalności i udogodnień zawartych w implementacji polimorfizmu, enkapsulacji i dziedziczenia w tym języku. Wykorzystanie każdego z tych trzech komponentów obiektowości w C#, może odbyć się na wiele różnych sposobów, m. in. za sprawą wielu słów kluczowych ułatwiających pracę: *public, private, internal* w przypadku enkapsulacji; *abstract, virtual, override* dla polimorfizmu, czy *sealed* i *partial* w dziedziczeniu. W tej konfrontacji język C# posiadał również wstępną przewagę z racji tego, że powstał stosunkowo niedawno, w porównaniu do konkurentów oraz jest konsekwentnie rozwijany od pierwszego dnia istnienia przez Microsoft. Zestawienie ze sobą C#, JavaScirpt oraz SmallTalk celem porównania implementacji obiektowego paradygmatu programowania pozornie wydaje się niesprawiedliwym wyborem, jednak celem tego studium bynajmniej nie było wyniesienie któregoś z języków na piedestał, a przybliżenie różnych podejść do obiektowości w programowaniu i ich obiektywna ocena.

ZAKOŃCZENIE

Paradygmat obiektowy odgrywa obecnie istotną rolę wśród technologii realizujących powstawanie systemów informatycznych. Jego dominacja w środowiskach zarówno akademickich jak i komercyjnych, dowodzi jego uniwersalności, potędze i potencjale. Obok paradygmatu obiektowego wśród społeczności informatycznych prosperuje także paradygmat funkcyjny programowania, choć notuje dużą stratę w liczbie zastosowań względem przodownika. Sytuacja ta powoduje, iż najnowsze języki programowania oraz te na bieżąco aktualizowane, nazywać można hybrydowymi, gdyż ich twórcy implementują w ich strukturach zarówno obiektowość jak i funkcyjność, celem odpowiedzi na wymagania deweloperów i samych systemów informatycznych.

Niniejsze studium dążyło do zrealizowania dwóch celów: po pierwsze szczegółowego i zrozumiałego opisania teorii programowania, jego konwencji i języków, przy szczególnym zwróceniu uwagi na obiektowy paradygmat programowania. Kolejny cel to przedstawienie czytelnikowi implementacji obiektowości w trzech językach: C#, JavaScript i SmallTalk oraz obiektywna i poparta faktami ocena realizacji tego paradygmatu przez wybrane technologie. Poprzez skrupulatne przedstawienie polimorfizmu, dziedziczenia i enkapsulacji w tych trzech językach, autor wysnuł odpowiednie wnioski przedstawione w końcowej części rozdziału trzeciego.

Pierwszy rozdział tejże pracy naukowej wprowadził odbiorcę w świat programowania komputerów, przedstawiając najbardziej popularne języki programowania, paradygmaty oraz podstawowe pojęcia. Zrozumienie zawartych w nim pojęć nie powinno stanowić problemów nawet dla osób na co dzień nie związanych z technologiami informacyjnymi, czy komputerami w ogóle, co udało się osiągnąć dzięki zastosowaniu licznych analogii do życia codziennego.

W drugiej części został rozłożony na czynniki pierwsze obiektowy paradygmat programowania – główny temat tego opracowania. W tym miejscu autor drobiazgowo charakteryzuje trzy główne ogniwa obietkowości: dziedziczenie, polimorfizm i enkapsulację. Przykłady przekazane w formie pseudo-kodu pozwolą na zrozumienie koncepcji każdemu, niezależnie od używanego i preferowanego języka programowania.

Po uważnej lekturze rozdziału drugiego, adresat będzie w stanie krótko omówić zasady oraz główne mechanizmy obiektowości w programowaniu.

Ostatni, trzeci rozdział dogłębnie przedstawił mechanizmy obiektowości wprowadzone w trzech badanych językach, a także zawiera ocenę stopnia realizacji opisywanego paradygmatu na ich przykładzie. Kolejno dla języków C#, JavaScript oraz SmallTalk drobiazgowo przedstawiono implementację dziedziczenia, polimorfizmu oraz enkapsulacji za pomocą przykładowych kodów źródłowych w tym języku. Do każdego kodu dodane zostało rzeczowe omówienie, co pozwoli uniknąć nieporozumień i problemów z odczytaniem zamysłu programistycznego. Końcowa część rozdziału przekazuje podsumowanie przeprowadzonych badań oraz dostarcza obiektywnej oceny badanego zagadnienia.

Wynik oceny na podstawie zawartych w niniejszym opracowaniu obserwacji, nie jest zaskoczeniem dla osób posiadających nawet niekoniecznie szeroką wiedzę techniczną. Paradygmat obiektowy naturalną siła rzeczy jest najbardziej efektywny w młodych językach programowania, stale rozwijanych i zorientowanych na komercyjne wykorzystanie, a ich twórcy coraz częściej pokuszają się o rozszerzanie funkcjonalności ich produktów poprzez realizację kilku paradygmatów programowania. Najczęstszą integracją paradygmatów jest powiązanie obiektowego i funkcyjnego, które realizują już języki takie jak Scala, Ruby czy w pewnym sensie też C# z technologią LINQ. Do tego rodzaju wariacji należy przyszłość systemów informatycznych, a także uważa się, iż wieloparadygmatowość języków programowania z czasem wejdzie do powszechnego użytku.

BIBLIOGRAFIA

1. Abelson H. (2002): *Struktura i interpretacja programów komputerowych*, Wydawnictwo Naukowo-Techniczne, Warszawa.
2. Albahari J., B. Albahari (2015): *C# 6.0 in a Nutshell,* O'Reilly Media Inc., Sebastopol CA.
3. Antani V. (2016): *Mastering JavaScript,* Packt Publishing, Birmingham.
4. Backfield J. (2014): *Becoming Functional*, O'Reilly Media Inc., Sebastopol CA.
5. Beck K. (1997): *SmallTalk Best Practice Patterns*, Prentice-Hall Inc., New Jersey.
6. Ben-Gan I. (2012): *Microsoft SQL Server 2012. T-SQL Fundamentals*, SolidQ, New York.
7. Boehm A., J. Murach (2015): *Murach's C# 2015*, Mike Murach & Associates, Fresno.
8. Coldwind G. (2015): *Zrozumieć programowanie,* Wydawnictwo Naukowe PWN, Warszawa.
9. Crockford D. (2008): *JavaScript: The Good Part*, O'Reilly Media Inc., Sebastopol CA.
10. Duckett J. (2013): *Effective JavaScript: 68 Specific Ways To Harness The Power Of JavaScript,* Addison-Wesley, New Jersey.
11. Duckett J. (2014): *JavaScript & JQuery: Interactive Front-End Web Development*, John Wiley & Sons Inc., Indianapolis.
12. Eckel B. (2006): *Thinking in Java, Fourth Edition*, Prentice Hall, New Jersey.
13. Frankowski K. (2012): *Śladami nauczycieli programowania,* Wydawnictwo Uniwersytetu Łódzkiego, Łódź.
14. Fulmański P., Ś. Sobieski (2004): *Wstęp do informatyki*, Uniwersytet Łódzki, Łódź.
15. Gabbrielli M., S. Martini (2010): *Programming Languages: Principles and Paradigms,* Springer, New York.
16. Haverbeke M. (2015): *Eloquent JavaScript: A Modern Introduction to Programming,* No Starch Press Inc., San Francisco.
17. Horstmann C. S., G. Cornell (2013): *Core Java. Volume I – Fundamentals, Ninth Edition*, Prentice Hall, New Jersey.

18. Hunt J. (1998): *SmallTalk and Object Orientation: An Introduction,* JayDee Technology Ltd., Wiltshire.
19. Kedar S. (2008): *Programming Paradigms And Methodology*, Technical Publications, Maharashtra.
20. Kingsley-Hughes A., Kingsley-Hughes K. (2005): *Beginning Programming*, Wiley Publishing, Indianapolis.
21. Lewis S. (1999): *The Art and Science of SmallTalk,* Prentice Hall, Hertfordshire.
22. Martin R. C. (2008): *Agile. Programowanie zwinne. Zasady, wzorce i praktyki zwinnego oprogramowania w C#*, Wydawnictwo Helion, Gliwice.
23. Martin R. C. (2009): *Clean Code. A Handbook of Agile Software Craftmanship*, Pearson Education, Boston.
24. McLaughlin B., Pollice G., D. West (2008): *Head First. Object-Oriented Analysis and Design*, O'Reilly Media, Sebastopol CA.
25. Meyer B. (2005): *Programowanie zorientowane obiektowo,* Wydawnictwo Helion, Gliwice.
26. Myers M. (2014): *A Smarter Way Learn to JavaScript. The new tech-assisted approach that requires half the effort,* CreateSpace Independent Publishing Platform, New York.
27. Nagel C. (2016): *C# 6.0 and .NET Core 1.0,* John Wiley & Sons, Indianapolis.
28. O'Sullivan B., J. Goerzen, D. Bruce Stewart (2008): *Real World Haskell*, O'Reilly Media Inc., Sebastopol CA.
29. Pirogov V. (2005): *Asembler. Podręcznik programisty,* Wydawnictwo Helion, Gliwice.
30. Pletzke J. (1999): *Advanced SmallTalk,* Addison-Wesley, New Jersey.
31. Samołyk G. (2011): *Podstawy programowania komputerów dla inżynierów,* Politechnika Lubelska, Lublin.
32. Sebesta R. W. (2012): *Concept of Programming Languages 10th Edition*, Pearson Education Inc., New Jersey.
33. Sedgewick R. (2012): *Algorytmty w C++*, Wydawnictwo Helion, Gliwice.
34. Sharp J. (2015): *Microsoft Visual C# Step by Step. Eight Edition,* Microsoft Press, Redmond.
35. Troelsen A., Japikse P. (2015): *C# 6.0 and the .NET 4.6 Framework*, SSBM Finance Inc, New York.

36. Weisfled M. (2014): *Myślenie obiektowe w programowaniu*, Wydawnictwo Helion, Gliwice.
37. Witold M., Szwoch M. (2008): *Metodologia i techniki programowania,* Wydawnictwo Naukowe PWN, Warszwa.
38. Wojtuszkiewicz K. (2010): *Programowanie strutkuralne i obiektowe,* Wydawnictwo Naukowe PWN, Warszawa.
39. http://stackoverflow.com.
40. *http://tiobe.com*.

SPIS RYSUNKÓW

SPIS KODÓW ŹRÓDŁOWYCH