# Wizualny język programowania ogólnego zastosowania

Krzysztof Osiecki

5 czerwca 2015

# Spis treści

# Wstęp

Pisząc programy w językach takich jak C++ czy Java istnieje bardzo dużo możliwości popełnienia błędu. Można zapomnieć o inicjalizacji zmiennej, czy jak zdarzyło się to w przypadku autora, o tym że ; po instrukcji warunkowej to też instrukcja i uniemożliwi ona poprawny przepływ sterowania. Takie błędy są trudne do wykrycia, ale zazwyczaj bardzo łatwe do naprawienia. Rodzi się więc pytanie czy dałoby się stworzyć język, który uniemożliwi popełnianie tego typu błędów? Potencjalnym rozwiązaniem jest stworzenie wizualnego języka programowania. Konstrukcja takiego języka mogłaby narzucać taką formę budowania programu, która eliminowałaby możliwość popełnienia części błędów, jak chociażby przypadkowe stworzenie pustego bloku w instrukcji warunkowej.

Celem niniejszej pracy jest właśnie stworzenie takiego języka, a przynajmniej rozpoczęcie tego procesu. Dotychczas istniejące języki programowania, są silnie wyspecjalizowane i nie nadają się do ogólnych zastosowań. W ramach pracy powstał silnie ustrukturalizowany wizualny język programowania, bazujący na języku C++. Sama praca jest opisem procesu tworzenia języka programowania oraz powstałego rezultatu. Autor nie ma na celu opisania wszystkich języków programowania, czy też wskazywania wad w konkretnych językach, ale przedstawienie różnych możliwości spośród których można wybierać podczas tworzenia własnego języka.

Chcąc stworzyć nowy język programowania trzeba odpowiedzieć na wiele pytań dotyczących tego, jak ten język będzie wyglądał i do jakich zastosowań będzie mógł być użyty. Czy będzie to samodzielny język, czy też będzie bazował na istniejącym już innym języku? Jakie instrukcje będzie posiadał? Jak będzie wyglądać składnia? Pytań dotyczących kształtu języka jest wiele, a fakt istnienia olbrzymiej różnorodności na polu języków programowania pokazuje, że na wiele z nich nie ma jednej odpowiedzi.

Pierwszy rozdział stanowi wprowadzenie, w którym ustalamy co sprawia, że język można nazwać językiem programowania. Zaprezentowany jest także rozwój programowania, poprzez przedstawienie kluczowych języków programowania w kolejności ich powstawania. W tym rozdziale dowiemy się także jakie cele powinien realizować funkcjonalny język programowania.

W drugim rozdziale znajduje się dokładniejsza analiza języka programowania pod kątem wybranego paradygmatu. Realizacja każdego z paradygmatów wymaga od języka pewnego konkretnego zestawu instrukcji. Zobaczymy jakie

instrukcje są konieczne ,dla którego paradygmatu, a także jak wygląda budowanie programu w każdym z paradygmatów.

Ostatni rozdział stanowi prezentacje rozwiązań wybranych przez autora w stworzonym przez niego języku. Fakt graficznej natury języka został już wyjaśniony we wstępie, pozostaje jednak wiele szczegółów, które to zostaną wyjaśnione właśnie w ostatnim rozdziale. Zawiera on zarówno opis konstrukcji, które zostały zawarte w języku jak i przegląd interfejsu aplikacji umożliwiającej tworzenie w nim programów. Pojęcia wizualnego i graficznego języka używane są w niniejszej pracy wymiennie.

# Rozdział 1

# Historia i rozwój języków programowania

## 1.1 Czym jest programowanie i język programowania

Encyklopedia PWN definiuje programowanie jako "... konstruowanie programów i przygotowanie ich do eksploatacji (...), również zajęcie polegające na opracowaniu algorytmów". Jest to powszechnie przyjmowana i intuicyjna definicja. Szukając informacji o programowaniu znajdziemy najczęściej właśnie takie wyjaśnienie tego pojęcia. Definicja, którą możemy znaleźć w internetowym słowniku Merriam-Webster, składa się z kilka możliwych wyjaśnień tego pojęcia. Jedno z nich jest zgodne z tym z encyklopedii PWN, inny z podpunktów jest jednak różny od wspomnianego powyżej wyjaśnienia: "The process of instructing or learning by means of an instructional program"[1]. Podpunkt ten można by niezbyt dosłownie przetłumaczyć jako "Proces instruowania bądź uczenia się poprzez zdefiniowany zestaw poleceń". Definicja programowania przyjęta w niniejszej pracy będzie brzmiała następująco:

**Definicja 1** *Programowanie - proces nauczania czegoś lub kogoś poprzez przekazanie dokładnych, jasnych do zinterpretowania instrukcji zachowania w danej sytuacji.*

Programowanie jest zatem czymś więcej niż tylko pisaniem kodu, który jest zrozumiały dla komputera. Byłaby to sztuka przekazania informacji w sposób umożliwiający pokierowanie akcjami bytu, któremu te instrukcje są przekazywane.

Programowanie nawet rozumiane w tym szerszym sensie wymaga narzędzia, które umożliwi wypełnienie kluczowego elementu przyjętej definicji, tj. przekazanie dokładnych instrukcji. Narzędzie takie nazywamy językiem programowania. W programowaniu komputerów właśnie ta dokładność komunikatów jest kwestią kluczową. Przekazanie informacji tak, aby komputer był w stanie zinterpretować je i wykonać, wymaga aby instrukcje nie pozostawiały żadnego miejsca

na błędną interpretację. Język programowania komputera, który będzie nazywany w dalszej części tej pracy po prostu językiem programowania, powinien spełniać te założenia. Oznacza to, że zarówno od strony syntaktycznej jak i semantycznej, nie powinno być możliwości wieloznacznej interpretacji instrukcji, bądź braku jej interpretacji. Mówiąc o syntaktyce języka mówimy o sposobie budowania "zdań", czyli poleceń. Reguły syntaktyczne informują nas o tym jak możemy składać słowa danego języka. Zasady te są zazwyczaj zdefiniowane za pomocą gramatyk formalnych i zapisane za pomocą wyrażeń regularnych i notacji Bacus'a-Naur'a. Fakt poprawności syntaktycznej nie informuje nas o tym iż zdanie posiada sens. Podobnie jak w przypadku języków naturalnych, choć zdanie "Kamil jest żonatym kawalerem" jest poprawną konstrukcją według zasad polskiej gramatyki to jest ono wewnętrznie sprzeczne. Innym przykładem mogłoby być zdanie "Kolorowe pomysły pływają najszybciej", które choć nie zawiera sprzeczności, nie niesie ze sobą żadnego powszechnie rozumianego przesłania. Dlatego aby posługiwać się językiem wymagane jest także rozumienie znaczenia komunikatów oraz poszczególnych słów. Część języka, która za to odpowiada nazywamy semantyką. Dzięki zdefiniowanym regułom semantycznym i syntaktycznym jesteśmy w stanie tworzyć poprawne wyrażenia w danym języku. Na tej podstawie kompilator jest w stanie przetłumaczyć instrukcje zapisane w języku programowania do kodu maszynowego wykonywanego przez komputer. W przypadku języków programowania, reguły syntaktyczne sprawiają, że jeśli linia kodu zostanie skompilowana to już niesie za sobą pewną informację. Nie oznacza to jednak iż, informacja ta wnosi coś do programu. Rozpatrzmy chociażby fragment kodu w napisany w języku C++.

```
int a = 0;
a=2;
a=5;
a=10;
cout<<a<endl;
```

Powyższy fragment ma na celu zdefiniowanie zmiennej typu całkowitego i wypisanie jej wartości na ekran. Pomimo tego, że wszystkie linie są poprawne w kontekście języka C++, to ustawienie zmiennej 'a' wartości 2 i 5 nie wnosi niczego do programu. Wartości te nie są nigdzie wykorzystywane. Z perspektywy użytkownika poniższy program wykona dokładnie te same operacje.

```
int a = 10;
cout<<a<endl;
```

Dlatego też o ile łatwo można automatycznie stwierdzić poprawność syntaktyczną języka, o tyle celowość wszystkich użytych w programie instrukcji wymaga już głębokiej analizy.

W niniejszej pracy zostanie przyjęta następująca definicja języka programowania:

**Definicja 2** *Język programowania - język formalny zaprojektowany w celu przekazania instrukcji dla maszyny, posiadający jednoznacznie zdefiniowaną semantykę i syntaktykę.*

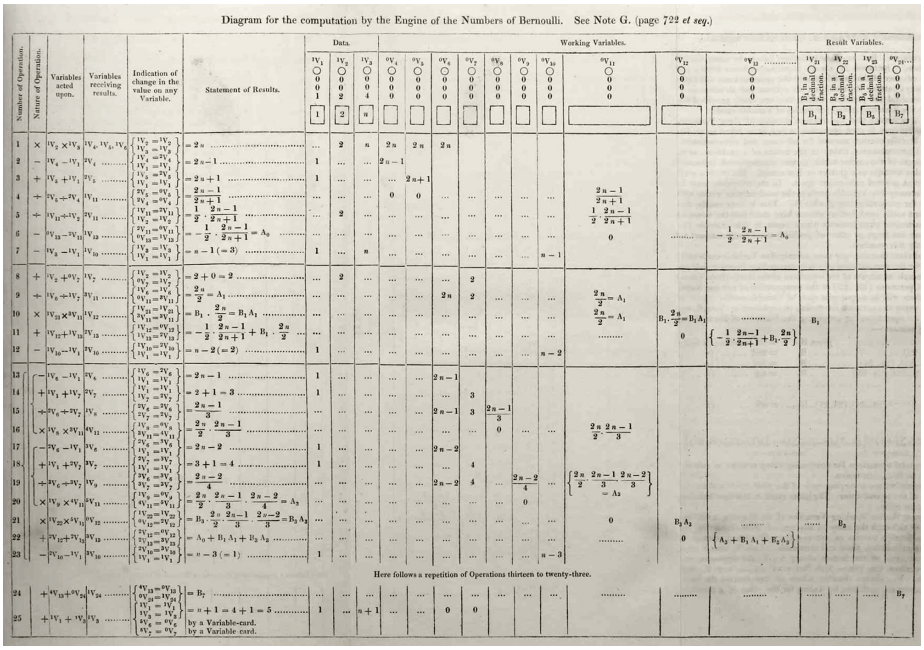
Pojęcie celu języka jest jasne i nie wymaga dalszego tłumaczenia, jeśli chodzi o pojęcie języka formalnego, wymaga ono nieco klaryfikacji. Język formalny to termin matematyczny, oznaczającym zbiór słów nad skończonym alfabetem. Języki formalne są z samej swej natury językami sztucznymi. Język sztuczny to język stworzony i zaprojektowany przez jakiś byt, czy to osobę czy grupę osób. Języki sztuczne nie zmieniają się dynamicznie w odróżnieniu od języków naturalnych, które powstały poprzez ewolucję komunikacji i cały czas rozwijają się i dopasowują do potrzeb, osób które ich używają.

## 1.2 Pierwsze języki programowania

Już od starożytności ludzie próbowali konstruować maszyny, które miały wykonać za nich pracę. Pierwszym przypadkiem, który można by uznać za krok w stronę programowania maszyny było pojawienie się krosna tkackiego projektu Josepha Jacquarda w 1805 roku[2]. Maszyna ta pozwalała tkać tkaninę o niemal dowolnym wzorze. Stworzenie wzoru polegało na przygotowaniu odpowiednich kart perforowanych. Schemat dziurek w karcie sterował podnoszeniem i opuszczaniem nitek osnowy, a co za tym idzie tkaniem wzoru, stworzonego przez "programistę". Choć Jacquard prawdopodobnie nie planował swojego dzieła w tym kontekście, to język kart perforowanych można wpisać w przyjętą w niniejszej pracy definicję języka programowania. Sztuczność języka Jacquarda nie podlega dyskusji, jest to niewątpliwie język stworzony przez jednostkę w konkretnym celu. Jako język formalny jest to, podzbiór zbioru słów nad binarnym alfabetem. Pierwszym znakiem alfabetu jest dziura w karcie, drugim jej brak. Celem języka jest przekazanie do krosna informacji o wzorze, który chcemy uzyskać. Od strony syntaktycznej ograniczeniami są kształt karty i sposób umieszczenia w niej dziurek. Semantyka jest jeszcze prostsza, dziurka oznacza opuszczenie nici, jej brak, nie opuszczanie. Jak widać język kart perforowanych spełnia wszystkie kryteria, które pozwalają zakwalifikować go jako język programowania. Dlatego to Jacquarda, a nie Adę Lovelace można uznać za pierwszego programistę w historii.

W roku 1837 Charles Babbage opublikował pracę w której opisał maszynę analityczną[3]. Maszyna ta według jego projektu byłaby w zasadzie prototypem komputera napędzanym silnikiem parowym. Budowa maszyny nie została ukończona przez Babbage'a, przed jego śmiercią w 1871r udało mu się zbudować tylko uproszczoną część maszyny. Pomimo pozornego braku sukcesu Charles Babbage wywarł wielki wpływ na kształt informatyki jaką znamy dzisiaj. Ada Lovelace matematyczka i córka poety Lorda Byrona, poznała Babbage'a, który podzielił się z nią pomysłem swojej maszyny. Wśród notatek Ady znaleźć można opis wykonania algorytmu znajdującego liczby Bernouliego z wykorzystaniem maszyny Babbage'a. Program jest opisem instrukcji które kolejno wykonywać ma maszyna i wykorzystuje podstawowe operacje matematyczne a także pętlę. Ze względu na ten właśnie program Ada Lovelace jest uznawana przez wielu za pierwszą programistkę na świecie[4].

Choć idea maszyny wykonującej skomplikowane operacje nie umarła, to od prób Babbage'a do momentu, w którym technologia pozwoliła na rozwój informatyki minęło dużo czasu. Dopiero w latach 30 i 40 XX wieku informatyka



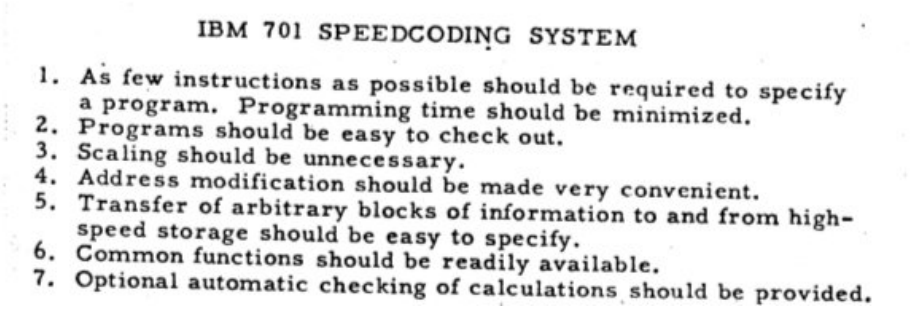

Rysunek 1.1: Program napisany przez Adę Lovelace [5]

zaczęła rozwijać się w szybszym tempie. Prace Alana Turinga czy Johna von Neumanna oraz fakt finansowania badań w związku z II wojną światową (łamanie szyfrów, doskonalenie broni) spowodowały powstanie maszyn, które przypominały już dzisiejsze komputery. Dostępność komputerów spowodowała rozwój programowania. Początkowo programowanie ograniczało się do pisania kodu maszynowego i wymagało dokładnej znajomości maszyny, na której program miał być wykonany. Pierwszy koncept języka wysokiego poziomu pojawił się w 1943 roku w postaci języka Plankalkül. Choć język podzielił los maszyny Babbage'a i doczekał się działającej implementacji dopiero w latach 90, to wprowadził wiele koncepcji używanych do dzisiaj, takich jak między innymi operator przypisania, instrukcje warunkowe, podprogramy, obsługa wyjątków. Pod koniec lat 40 i na początku 50 rozwinęła się idea języków niskiego poziomu. Języki assemblerowe nie oferowały wielkiego wzrostu komfortu programistów, ale były już krokiem w dobrą stronę. Lata 50 zaowocowały powstaniem kolejnych idei języków wysokiego poziomu, takich jak Short Code czy Autocode. Żaden z tych języków nie odniósł jednak spektakularnego sukcesu, aż do roku 1957, kiedy to na rynku pojawił się zaprojektowany przez Johna Backusa i stworzony przy współpracy z IBM, Fortran.

## 1.3 Rozwój języków

John Backus miał już za sobą pierwsze próby tworzenia języka programowania. W 1953 roku stworzył on system Speedcoding mający na celu ułatwienie pisanie programów na komputer IBM 701. Speedcoding miał na celu umożliwienie wykonywania obliczeń przy użyciu przyjaznej dla człowieka składni oraz obsługi

operacji wejścia/wyjścia. Raport przygotowany przez Johna Backusa i Harlana Herricka dla Biura Badawczego Marynarki Wojennej Stanów Zjednoczonych[7] pokazuje jasną wizję, którą Backus miał dla języków programowania, zwanych przez niego "automatic-programming systems". Celem języka miało być oferowanie operacji niebędących częścią bazowych możliwości maszyny oraz umożliwienie wykorzystania operacji oferowanych przez maszynę w sposób bardziej przystępny. Autorzy uważali, że dodatkowy czas pracy komputera, poświęcony na "tłumaczenia" instrukcji na kod maszynowy, jest mniej kosztowny od tego wymaganego od programisty w celu napisania od razu kodu maszynowego.



IBM 701 SPEEDCODING SYSTEM

1. As few instructions as possible should be required to specify a program. Programming time should be minimized.
2. Programs should be easy to check out.
3. Scaling should be unnecessary.
4. Address modification should be made very convenient.
5. Transfer of arbitrary blocks of information to and from high-speed storage should be easy to specify.
6. Common functions should be readily available.
7. Optional automatic checking of calculations should be provided.

Rysunek 1.2: Cele stawiane przed językami programowania przez Johna Backusa [7]

Zasady przedstawione na rysunku **??** miały według Backusa przyświecać językom programowania. Takie podejście miało zdecydowanie ułatwić tworzenie oprogramowania. Zyski jakie miałby dać prosty system programowania to między innymi:

- Możliwość re-używania kodu.
- Redukcja wiedzy koniecznej do programowania.
- Łatwiejsze obrazowanie problemu.
- Łatwiejsze odnajdywanie błędów w programie.

Backus zaprojektował język, który stał się pierwszym popularnym językiem wysokiego poziomu. Pierwsza proponowana wersja Fortrana zawierała niewiele instrukcji. Nazwy zmiennych składały się z dwóch znaków. Pierwszy oznaczał typ, litery od i do n liczbę całkowitą, pozostałe zmiennoprzecinkową. Język zawierał instrukcję podstawienia, instrukcję warunkową, pętlę i instrukcję skoku. Pozwalał na zdefiniowanie tablicy poprzez specjalną instrukcję DIMENSION, w której należało wskazać ilość elementów. Całość dopełniała instrukcja stopu oraz obsługa wejścia wyjścia.

Pierwsza wersja języka, dla którego został stworzony kompilator nie różniła się bardzo od propozycji przedstawionej w poprzednim akapicie, poza faktem uproszczenia niektórych instrukcji ze względu na trudność implementacji. Fortran I zawierał instrukcję warunkową umożliwiającą skok do jednej z trzech lokacji w zależności od znaku zmiennej przekazanej jako warunek. Instrukcja

pętli wymagała ustalenia liczby iteracji oraz wskazania linii kończącej pętlę. W trakcje implementacji powstała też potrzeba stworzenia instrukcji, pozwalającej ominąć dalszą część wykonania pętli bez omijania zwiększenia indeksu, spowodowało to powstanie instrukcji CONTINUE. Możliwa długość nazw zmiennych wzrosła z dwóch do sześciu znaków. Pomimo swoich braków Fortran już w swojej pierwszej wersji przypomina znane nam współcześnie języki programowania. Posiadając standardowe instrukcje kontroli przepływu, zmienne i tablice, Fortran był bez wątpienia rewolucją w świecie programowania. Raporty przygotowane w rok po stworzeniu pierwszego kompilatora wskazują, że w tak krótkim czasie, ponad połowa programów tworzonych na komputery IBM 704 napisana była właśnie w Fortranie. W przeciągu dwóch lat powstały kolejne wersje Fortrana wprowadzające obsługę funkcji, ciągów znakowych i typu boolowskiego. Język jest ciągle rozwijany, ostatnia gotowa wersja znana jako Fortran 2008 została zaakceptowana przez standard ISO w 2010 roku. Aktualnie trwają pracę nad kolejnym standardem zwanym Fortran 2015.

Koniec lat 50 i całe lata 60 zaowocowały prawdziwym wysypem nowych pomysłów na język programowania. Wtedy narodziła się koncepcja języków takich jak Lisp, ALGOL czy COBOL. Standardem dla języków programowania stały się instrukcje, które zaproponował Fortran. Dominującym paradygmatem był imperatywny. Pomimo dużej ilości powstających języków oraz powstawaniu kolejnych wersji już istniejących, żaden z nich nie stanowił przełomu w programowaniu.

Początek lat 70 to narodziny kluczowych języków. Już w roku 1970 powstał Pascal. Prawdziwie przełomowym okazał się jednak rok 1972. Powstał wtedy zarówno język C, jak i pierwszy prawdziwie rozpowszechniony język programowania w logice Prolog. W tym samym roku stworzony został także język zapytań SQL oraz jeden z pierwszych języków obiektowych Smalltalk. C jest językiem oferującym bardzo duże możliwości, doświadczony programista jest w stanie wykorzystać specyficzne elementy języka do uzyskania spektakularnych efektów, zaś laik, już po kilku godzinach jest w stanie napisać pierwszy nie banalny program. Ta kombinacja mocy i prostoty sprawiła, że C jest do dzisiaj jednym z najbardziej popularnych języków programowania na świecie, będąc praktycznie nie zastąpiony w miejscach, w których wydajne działanie stanowi priorytet. C jest także protoplastą kolejnego z powszechnie używanych aktualnie języków C++. Smalltalk przyczynił się do stworzenia czwartego w rankingu TIOBE w kwietniu 2015r[6] języka Objective-C. SQL jest z kolei monopolistą w obsłudze relacyjnych baz danych.

Kluczowym hasłem lat 80 w programowaniu jest obiektowość. Podobnie jak w poprzednim dziesięcioleciu, najważniejsze wydarzenia miały miejsce w jego początkach. Pierwsze podejście do obiektowego C z roku 1980, czyli C with classes, zostało w 1983r zastąpione językiem C++. W tym samym roku powstał także język Objective-C. Oba języki są obecnie niezwykle popularne i oba powstały na bazie języka C. Oba powstały z chęci wprowadzenia do C obiektowości, ale sposób jej implementacji był wzorowany na innych przodkach. C++ wyrósł z idei wywoływania metod na rzecz obiektów, zaczerpniętej z języka Simula, zaś Objective-C rozwinął pomysł wysyłania komunikatów do obiektów wzorowany

na Smalltalk'u. W tym samym roku powstała także obiektowa wersja Pascala - TurboPascal. Wartym wspomnienia jest pojawienie się w 1989r języka powłoki unixowej, Basha.

Lata 90, otworzyło pojawienie się jednego z najbardziej obecnie popularnych języków funkcyjnych Haskell'a. Rok później powstał, będący według badań z lipca 2014r najpopularniejszym językiem na kursach programowania na najwyżej sklasyfikowanych amerykańskich uczelniach Python [8], a także dość znany Visual Basic. 1993 to rok powstania popularnego, zwłaszcza w branży gier wideo, skryptowego języka LUA. Prawdziwa rewolucja nastąpiła jednak w roku 1995. Wtedy to pojawił się modny ostatnio, głównie dzięki popularności zapewnianej przez framework Ruby on Rails, język Ruby. W tym samym roku powstał obecny, król języków skryptowych w sektorze rozwiązań webowych, JavaScript. Rozwój stron internetowych, niewątpliwie przyspieszyło także pojawienie się języka PHP. Do tego zacnego grona dołącza język Java, który w XXI wieku, nie znalazł się w rankingu popularności języków programowania na niższej niż trzecia pozycji.

Najważniejsze wydarzenie XXI wieku, jeśli chodzi o języki programowania, to pojawienie się, stworzonego przez Microsoft, języka C#. Z bardziej popularnych języków powstałych od roku 2000 można wymienić Scalę, Groovy czy najświeższą próbę firmy Apple - Swift. Poza wspomnianymi językami, na przestrzeni mniej więcej siedemdziesięciu lat, powstało ich oczywiście o wiele więcej. Istnieje co najmniej kilka setek języków programowania. Celem tej pracy nie jest jednak opisanie ich wszystkich, a raczej spojrzenie na te najbardziej popularne, znalezienie cech, które powodują, że to właśnie te języki odnoszą sukces, podczas gdy tak wiele innych znika bez śladu.

## 1.4 Stan obecny

Aktualnie dominującym paradygmatem programowania, jest paradygmat obiektowy. Ciężko jednak znaleźć popularny język, który nie wykorzystywałby co najmniej kilku paradygmatów. Zalety języków funkcyjnych są tak, duże że wplatane są one w języki, które miały być po prostu obiektowe (C#, Java, Python). Wartym zauważenia jest fakt, że pięć najpopularniejszych w rankingu TIOBE języków, ma bardzo silny związek z C są to: Java, C, C++, Objective-C i C#. Można nawet zaryzykować stwierdzenie, że pierwsza piątka to C i pochodne. C++ nie ukrywa swojego rodowodu. Cel języka Objective-C był taki sam jak C++ tylko rozwiązanie okazało się inne. Java bardzo mocno przypomina C++, a C# bardzo mocno przypomina Javę. Języki te zdecydowanie dominują na rynku informatycznym, a znajomość chociaż jednego z nich, wydaje się być konieczna dla dobrego programisty. Tuż za wielką piątką plasują się języki używane głównie do tworzenia aplikacji internetowych, JavaScript, PHP, Python.

Podsumowując obecnie na polu języków programowania, prym wiodą języki wywodzące się z C. Nowe języki traktowane są raczej jako ciekawostki niż realni konkurenci. Wiele ciekawych rozwiązań znajduje swoje nisze np. Scala czy

Lua ale nie są w stanie zagrozić monopolowi C-podobnych gigantów. Rozwój języków programowania polega teraz bardziej na łączeniu najlepszych rozwiązań poszczególnych paradygmatów, niż próbie stworzenia nowego paradygmatu.

# Rozdział 2

# Paradygmaty i budowa języków programowania

## 2.1 Tworzenie języka programowania

Przed przystąpieniem do próby stworzenia nowego języka programowania, należy odpowiedzieć na kilka pytań, które wyznaczą ramy i strukturę języka. Najważniejszym wyborem jaki musi podjąć twórca języka programowania, jest wybór paradygmatu. Wybór paradygmatu nie oznacza ograniczenia się do jednego z nich. Można łączyć najbardziej odpowiadające twórcy rozwiązania wielu paradygmatów, a każde z nich będzie miało silny wpływ na charakter powstającego języka. Wybranie paradygmatu określi ogólny kształt i naturę programów pisanych w danym języku. W obrębie paradygmatu należy wybrać, jakie instrukcje zostaną zapewnione przez standard języka. Każdy z paradygmatów, wymaga pewnego minimalnego zbioru operacji w celu zapewniania pełnej funkcjonalności języka. Niezależnie od paradygmatu, język programowania, będzie musiał zapewnić obsługę operacji wejścia i wyjścia, a także obsługę podstawowych operacji matematycznych. Twórca języka musi podjąć decyzję, jakie dodatkowe instrukcje, poza tymi, które są niezbędne, zechce umieścić w jego standardzie. Spójrzmy zatem na niektóre z dostępnych paradygmatów, jakie są ich główne cechy oraz jakie konstrukcje języka są wymagane, aby zapewnić funkcjonalność języka. Zacznijmy jednak od dokładniejszego wyjaśnienia pojęcia paradygmatu.

## 2.2 Czym jest paradygmat programowania

Czym jest paradygmat? Jak podaje Słownik Języka Polskiego PWN [9], paradygmat to "przyjęty sposób widzenia rzeczywistości w danej dziedzinie, doktrynie itp." lub "zespół form fleksyjnych danego wyrazu lub końcówek właściwych danej grupie wyrazów". O ile pierwszą definicję można wykorzystać do wytłumaczenia pojęcia paradygmatu programowania, o tyle druga jest specyficzna, raczej dla lingwistów niż informatyków. Słowo paradygmat wywodzi się z greckiego παράδειγμα, oznaczającego wzorzec bądź przykład. W kontekście paradygmatu programowania oznacza to zbiór mechanizmów, używanych przez

programistę w danym języku i sposób późniejszego wykonywania tego programu na maszynie na której ma działać. Ray Toal, profesor w Loyola Marymount University w Kalifornii wyróżnia w artykule na swojej stronie 14 paradygmatów [10], które określa jako "bardziej popularne'. Jak zostało wspomniane w poprzednim akapicie, poszczególne paradygmaty nie są wzajemnie wykluczające się, dlatego można łączyć użycie większej liczby z nich w jednym języku programowania. W niniejszej pracy przedstawione zostaną, dwie rodziny paradygmatów, prezentujące skrajnie różne podejście do sposobu tworzenia programów.

## 2.3 Paradygmat imperatywny

Paradygmat imperatywny, stanowiący korzeń imperatywnej rodziny paradygmatów, zakłada poddanie precyzyjnych instrukcji i manualną kontrolę przebiegu działania programu. Program imperatywny to lista instrukcji, które mają zostać wykonane. Każda z instrukcji może modyfikować obecny stan maszyny. Jeśli chcemy wykonać przeniesienie sterowania do innego miejsca programu, należy zastosować instrukcję skoku. Można powiedzieć, że programowanie imperatywne istnieje od momentu napisania pierwszego przepisu kulinarnego. Produkty wymagane do przygotowania potrawy, garnki i kuchenkę, traktujemy jako stan maszyny, zaś opisy poszczególnych czynności, np. "dolej mleka" są instrukcjami modyfikującymi ten stan. Najbliższe temu paradygmatowi jest programowanie w językach assemblerowych, gdzie operujemy bezpośrednio na rejestrach komputera. Języki wysokiego poziomu, jak na przykład C, także oferują możliwość pisania programów w paradygmacie imperatywnym. Różnicą jest tutaj fakt, że instrukcje nie operują bezpośrednio na stanie maszyny, ale na pewnej abstrakcji obrazowanej przez zmienne. Języki imperatywne nie wymagają dużego zestawu instrukcji aby być funkcjonalne. Język imperatywny wymaga do bycia funkcjonalnym, następującego zestawu instrukcji:

- Instrukcja przypisania - umożliwiają zapisanie stanu w pamięci maszyny.
- Instrukcja warunkowa - w zależności od spełnienia zadanego warunku wykonuje, bądź nie, wskazany fragment kodu
- Instrukcja skoku - instrukcja pozwala na przeniesienie sterowania w inne miejsce programu

### 2.3.1 Paradygmat strukturalny i proceduralny

Paradygmat strukturalny jest rozwinięciem paradygmatu imperatywnego, o dodatkowe struktury sterujące. Używanie instrukcji skoku (często w połączeniu z instrukcją warunkową) może prowadzić do trudności w zrozumieniu programu. Języki strukturalne rozszerzają podstawowy zakres instrukcji w celu zapewnienia łatwiejszej kontroli nad działaniem programu. Na przykład instrukcja pętli while z języka c, oferuje prostą składnię, która może zastąpić użycie połączenia instrukcji warunkowej i skoku. Główną ideą tego paradygmatu jest wykorzystywanie struktur kontrolnych do dzielenia kodu na łatwiejsze do zrozumienia bloki. Paradygmat proceduralny jest kolejnym krokiem w stronę tworzenia programów łatwych do zrozumienia przez programistę. Języki proceduralne pozwalają na

wydzielenie fragmentów kodu do specjalnych bloków zwanych procedurami. Procedury mogą być następnie wywoływane z innego miejsca w programie. Rozwiązanie to eliminuje konieczność powtarzania tego samego kodu w wielu miejscach. Aby zrealizować wywołanie procedury w czysto imperatywnym języku, należałoby wykonywać instrukcję skoku, przechowując w jakiejś globalnej zmiennej miejsce do którego należy wykonać skok, po wykonaniu instrukcji traktowanych jako procedura. Pomimo, iż jest to wykonalne, to jest to rozwiązanie nieczytelne, które bardzo utrudnia znalezienie ewentualnych błędów w kodzie. W teorii procedury powinny używać do obliczeń, tylko danych, które zostaną przekazane jako parametry wywołania i być niezależne od innych danych w programie. Nie jest to jednak zasada, której złamanie dyskwalifikowałoby język jako proceduralny. Podsumowując języki proceduralne i strukturalne rozwijają imperatywny zestaw instrukcji, na przykład o następujące konstrukcje:

- Instrukcja pętli - dostępna często w różnych wariantach (for, while, do-while, foreach)
- Instrukcja wyboru - rozwinięcie instrukcji warunkowej, w strukturę switch-case
- Deklaracja i wywołanie procedury - dotyczy tylko języków proceduralnych

Struktury zapewniane przez języki strukturalne i proceduralne, powodują znaczny spadek znaczenia instrukcji skoku.

### 2.3.2 Paradygmat obiektowy

W paradygmacie programowania obiektowego programy definiuje się za pomocą obiektów. Obiekt posiada swój stan, czyli dane oraz zachowanie, czyli możliwe do wykonania procedury. Program jest więc zbiorem obiektów komunikujących się między sobą w celu wykonania zadania. Zwolennicy tego paradygmatu uznają, iż jest on bliski temu jak ludzie postrzegają rzeczywistość. Nawet jeśli uzna się ten pogląd za przesadzony to nie można nie zauważyć, olbrzymiej popularności tego właśnie paradygmatu. Spójrzmy na pojedynczy obiekt. Jest to w zasadzie program. Posiada dane i instrukcje, które mogą zostać wykonane, pogrupowane w procedury. Można zatem uznać, że obiektowy paradygmat to w zasadzie sposób definiowania relacji pomiędzy programami proceduralnymi. Bądź w odwrotnym kierunku, programy proceduralne, to programy obiektowe zawierające tylko jeden obiekt. Programy obiektowe są z natury imperatywne, obiekt jest tak na prawdę kolejną strukturą ułatwiającą dzielenie kodu na łatwe do zrozumienia bloki. Język obiektowy poza wymaganymi instrukcjami z paradygmatu proceduralnego, musi udostępniać możliwość wykonania następujących operacji:

- Stworzenie obiektu - może być realizowane na różne sposoby, w zależności od podtypu języka może wymagać oddzielnego zdefiniowania klasy obiektu, bądź prototypu
- Interakcja z obiektem - czyli możliwość wywołania metod na rzecz obiektu, czy dostępu do jego stanu wewnętrznego

## 2.4 Paradygmat deklaratywny

Paradygmat deklaratywny jest skrajnie różny od paradygmatu imperatywnego. Programista tworząc deklaratywny program, nie pisze instrukcji, które mają się wykonać, lecz opisuje warunki jakie ma spełniać rozwiązanie. Paradygmat deklaratywny stanowi w zasadzie rodzinę paradygmatów, do której zaliczyć można programowanie funkcyjne oraz programowanie w logice. Znanym językiem wpisującym się w ramy tego paradygmatu jest SQL. Program napisany w SQL-u (zwany często zapytaniem) nie mówi o tym w jaki sposób silnik bazy danych powinien dostać się do danych. Zapytanie SELECT zawiera tylko opis tego jakie dane powinny być zwrócone, sposób w jaki silnik bazodanowy uzyska do nich dostęp nie jest dla programisty istotny. Aby można było uznać program za w pełni deklaratywny musi spełniać on trzy warunki. Program musi być niezależny, czyli wynik nie zależy od, żadnego zewnętrznego stanu. Bezstanowy, sam program nie posiada stanu, który byłby zachowany pomiędzy wywołaniami. Deterministyczny, czyli dla tych samych danych wejściowych wynik powinien być zawsze taki sam. Realizacje deklaratywnego paradygmatu nie są ze sobą tak silnie związane, jak wspomniane wcześniej warianty paradygmatu imperatywnego. Jako wspólną cechę wszystkich deklaratywnych języków programowania podać można w zasadzie tylko konieczność zapewnienia sposobu na opisanie oczekiwanego rozwiązania.

### 2.4.1 Paradygmat funkcyjny

Programowanie funkcyjne jest odmianą programowania deklaratywnego. Wartościami podstawowymi w tym paradygmacie są funkcje. Programy nie mają na celu wykonywania instrukcji, lecz obliczanie wartości funkcji. Języki czysto funkcyjne spełniają wszystkie warunki języka deklaratywnego. Język funkcyjny do działania wymaga w zasadzie tylko możliwości zdefiniowania funkcji. Funkcja posiada typ zwracany oraz argumenty. Pętle są zastąpione przez rekurencyjne wywoływanie funkcji, a instrukcja warunkowa nie jest niezbędna, gdyż może zostać zastąpiona przez odpowiednie definiowanie funkcji. Mimo tego nawet języki czysto funkcyjne na przykład Haskell, używają konstrukcji określanej jako wyrażenie warunkowe. Jest to konstrukcja przypominająca znany z c operator trójargumentowy. Jeśli pierwszy argument jest prawdą jako wynik wyrażenia zwracany jest drugi, a jeśli pierwszy argument jest fałszem to wynikiem jest trzeci. Poniżej prezentowane są dwa sposoby na zdefiniowanie funkcji liczącej silnie w języku Haskell. Pierwszy z wykorzystaniem wyrażenia warunkowego, drugi poprzez odpowiednią definicję parametrów funkcji.

```
factorial n = if n == 0
              then 1
              else n * factorial (n - 1)

factorial 0 = 1
factorial n = n * factorial (n-1)
```

Podsumowując, język funkcyjny do zapewnienia funkcjonalności potrzebuje tylko dwóch konstrukcji.

- Deklaracji funkcji

- Wywołania funkcji

Całość działania programu oparta jest na odpowiednim składaniu funkcji. W językach funkcyjnych ze względu na założenie niepozwalające na generowanie efektów ubocznych, w porównaniu z programami imperatywnymi utrudnione jest realizowanie obsługi operacji wejścia i wyjścia.

### 2.4.2 Paradygmat programowania w logice

Programowanie w logice tak jak i programowanie funkcyjne, jest odmianą programowania deklaratywnego. Programem jest tutaj zestaw zależności. Zależności te opisują interesujące nas własności. Obliczenia polegają na próbie dowiedzenia, bądź obalenia jakiegoś twierdzenia. Weźmy za przykład program napisany w języku Prolog. Celem programu będzie sprawdzenie, czy w danym grafie skierowanym istnieje ścieżka pomiędzy zadanymi wierzchołkami.

```
krawedz(a,b).
krawedz(b,c).
krawedz(d,e).

polaczone(X, Y) :- X = Y.
polaczone(X, Y) :- krawedz(X,Z), polaczone(Z, Y).
```

Wyrażenia nieposiadające ciała, występującego po znaku `:-`nazywamy faktami. Zaś wyrażenia w postaci `A:-B` regułami. Fakty w tym przypadku stanowią reprezentację grafu. Jest to graf posiadający wierzchołki `a,b,c,d` i `e` oraz krawędzie `(a,b),(b,c)`oraz `(d,e)`. Reguły pokazują jak sprawdzić czy dwa wierzchołki są połączone. Pierwsza reguła określa istnienie ścieżki pomiędzy wierzchołkiem, a samym sobą. Druga reprezentuje rekurencyjną zależność, pozwalającą na obliczenie pozostałych krawędzi. Program ten zgodnie z założeniem pozwala sprawdzić czy dwa wierzchołki są połączone.

```
?- polaczone(a,b).
true
?- polaczone(a,d).
false
```

Program ten oferuje jednak większe możliwości. W języku Prolog zmienne muszą rozpoczynać się od wielkiej litery, zaś stałe od małej. Dotychczas wykonywaliśmy nasz program tylko dla stałych. Oto co uzyskamy jeśli użyjemy zmiennych.

```
?- polaczone(X,Y).
X = Y
X = a, Y = b
X = a, Y = c
X = b, Y = c
X = d, Y = e
false
```

```
?- polaczone(a,Y).
Y = a
Y = b
Y = c
false
```

Wykorzystując ten sam program jesteśmy w stanie obliczyć wszystkie ścieżki pomiędzy wierzchołkami w grafie. Pierwsze z wykonań programu pokazuje w wyniku, że istnieje ścieżka pomiędzy każdym wierzchołkiem, a nim samym. Kolejne linie pokazują takie wartościowania dla `X` i `Y`, dla których reguła `polaczone` jest spełniona. Widać zatem, że w grafie istnieją ścieżki pomiędzy następującymi parami wierzchołków `(a,b),(a,c),(b,c),(d,e)` oraz oczywiście ścieżki `(a,a),(b,b),(c,c),(d,d),(e,e).` Drugie wykonanie jest przykładem połączenia użycia stałej i zmiennej. Program odnajdzie wszystkie wartościowania dla `Y,`które spełnią regułę `polaczone`. Zatem widać, że z wierzchołka a istnieje ścieżka do wierzchołków: `a,b,c`. Komunikat `false,`wypisany na końcu każdego z wykonań oznacza, iż Prolog zakończył obliczenia i nie ma już więcej możliwych rozwiązań.

Języki programowania w logice, do bycia funkcjonalnymi wymagają instrukcji pozwalającej na dodanie faktu do bazy wiedzy oraz instrukcji opisującej wyrażenie, które próbujemy wartościować. Łatwo zauważyć analogię pomiędzy instrukcjami programowania funkcyjnego (deklaracja funkcji, wywołanie funkcji) i programowania w logice (dodanie faktu do bazy wiedzy, wykonanie wartościowania z wykorzystaniem faktów). Związane jest to oczywiście z wymaganiami dotyczącymi języka deklaratywnego.

# Rozdział 3

# Opis założeń i działania aplikacji

## 3.1 Pomysł na graficzny język programowania

Pomysł na programowanie w formie graficznej nie jest nowy. Schematy blokowe, są bardzo dobrym narzędziem, pozwalającym rozpocząć naukę programowania. Pozwalają one przedstawić, algorytm w postaci kroków połączonych strzałkami obrazującymi przepływ sterowania. Innym przykładem może być zaprojektowany na MIT, Scratch. Scratch jest językiem mającym ułatwić naukę programowania najmłodszym. Scratch jest kolorowy i łatwy do przyswojenia. Instrukcje wydaje się poprzez układanie w kolejności bloków opisanych w języku angielskim. Construct 2 jest językiem umożliwiającym programowanie gier. Jest on bardziej skomplikowany od Scratch-a, jednak pozwala w dość prosty sposób stworzyć prawie dowolną grę w 2D.

Aplikacja, która powstała podczas pisania niniejszej pracy, ma na celu dostarczenie języka prostego i łatwego do nauczenia się, jednak łatwo rozszerzalnego. Głównym paradygmatem, stanowiącym o kształcie języka jest paradygmat strukturalny. Za bazę języka przyjęty został C++. W celu zaoferowania maksymalnej ilości funkcjonalności umożliwione zostało pisanie kodu w natywnym C++. Cechą szczególną języka, którego główną inspiracją były schematy blokowe, jest możliwość definiowania bloku i eksportowania go w celu re-użycia w innych miejscach programu, bądź nawet opublikowania zbioru bloków na zasadzie biblioteki. Możliwość stworzenia biblioteki bloków, ma pozwolić na rozszerzanie języka i dodawanie do niego nowych funkcjonalności, bez konieczności modyfikacji kodu źródłowego. Ze względu na ścisłą strukturę, narzucaną przez konieczność organizowania wszystkich operacji w bloki, każde rozszerzenie automatycznie będzie wpisywać się w istniejącą strukturę języka. Wspomniane wcześniej wizualne języki, są skierowane na konkretne zadanie powoduje to znaczne zawężenie możliwości języka. W przypadku języka tej aplikacji ograniczeniem są tylko możliwości języka C++, dodatkowo bazowe funkcjonalności tego języka opakowano w łatwą do zrozumienia graficzną formę. Celem projektowym języka było położenie dużego nacisku na ustrukturalizowanie przepływu sterowania. Pozostałe operacje realizowane są przy użyciu języka C++. Możliwe jest jednak

przysłonięcie operacji wykonywanych w C++ poprzez definiowanie odpowiednich struktur wykorzystujących bloki.

## 3.2 Budowa języka

Standardowe funkcjonalności dostępne w aplikacji obejmują podstawowe struktury niezbędne do działania języka, są to:

- Pętla - występująca w jednym z trzech wariantów

  Warunkowa - odpowiada pętli while znanej z C++, wykonuje się tak długo jak spełniony jest warunek

  Ilościowa - odpowiada pętli for z C++, wykona się wskazaną ilość razy

  Po kolekcji - odpowiada pętli foreach dostępnej w standardzie C++ od wersji 11, iteruje po wszystkich elementach wskazanej kolekcji

- Instrukcja warunkowa - blok wykonujący się w przypadku spełnienia podanego przy jego definicji warunku

- Blok tekstowy - jest to blok pozwalający na tworzenie kodu w natywnym C++. Zapewnia funkcjonalność języka poprzez udostępnienie instrukcji podstawienia, operacji wejścia/wyjścia oraz innych możliwości oferowanych przez język C++.

Poza wymaganymi w celu zapewnienia funkcjonalności języka strukturami, język udostępnia także, struktury mające ułatwić tworzenie w nim programów oraz wsparcie funkcjonalności eksportowania bloków. Struktury te to:

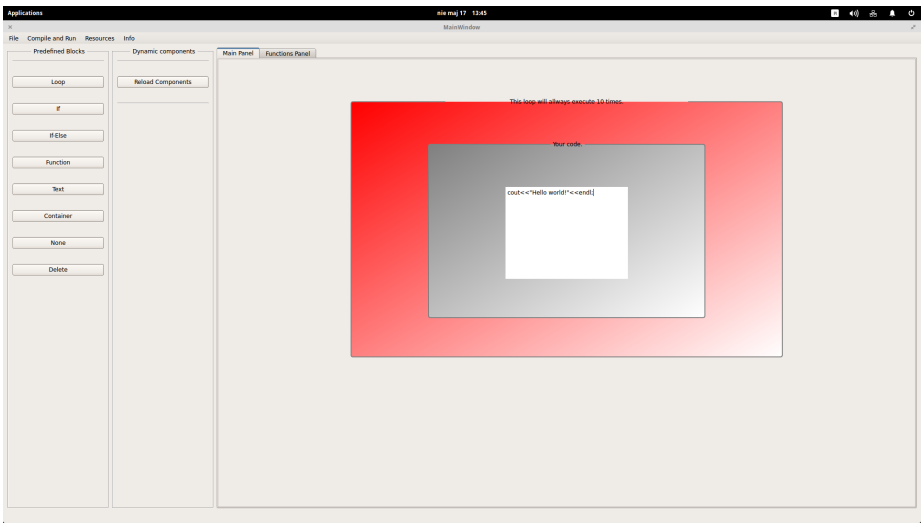
- Instrukcja warunkowa if-else - blok zawierający parę bloków z których pierwszy wykona się w przypadku spełnienia podanego przy definicji warunku, drugi w przeciwnym przypadku. Instrukcja warunkowa, bardzo często bywa używana w ten właśnie sposób. Ze względu na ten fakt, oferowanie struktury automatycznie definiującej powszechny przypadek użycia, powinno zwiększyć czytelność programów.

- Blok funkcyjny - pozwalający zdefiniować funkcję analogiczną do tej z C++, która może być wywołana poprzez użycie jej nazwy w bloku tekstowym.

- Kontener - blok grupujący pozostałe bloki, z punktu widzenia pojedynczego programu pełni role { ... } z języka C++, czyli pozwala sterować zakresem widoczności zmiennych, stanowi także ważny element w przypadku tworzenia dynamicznych komponentów. Ponieważ eksportu dokonujemy na pojedynczym bloku, jeśli chcemy stworzyć komponent zawierający w sobie więcej bloków musimy umieścić je w bloku-kontenerze i dokonać eksportu tego właśnie bloku.

Aplikacja zawiera także opcje pozwalającą na tworzenie zmiennych globalnych, a także zarządzanie importowanymi bibliotekami i przestrzeniami nazw. Najważniejszą cechą języka wspomnianą już we wstępie do tego rozdziału, jest możliwość eksportowania bloków, a w zasadzie całych drzew bloków, ponieważ blok jest eksportowany z całą wewnętrzną zawartością. Rozdział 3.3.4 opisuje sposób użycia tej funkcjonalności. Znaleźć można w nim opis definiowania prostego komponentu, będącego w stanie zastąpić standardową obsługę operacji wyjścia.

## 3.3 Interfejs aplikacji



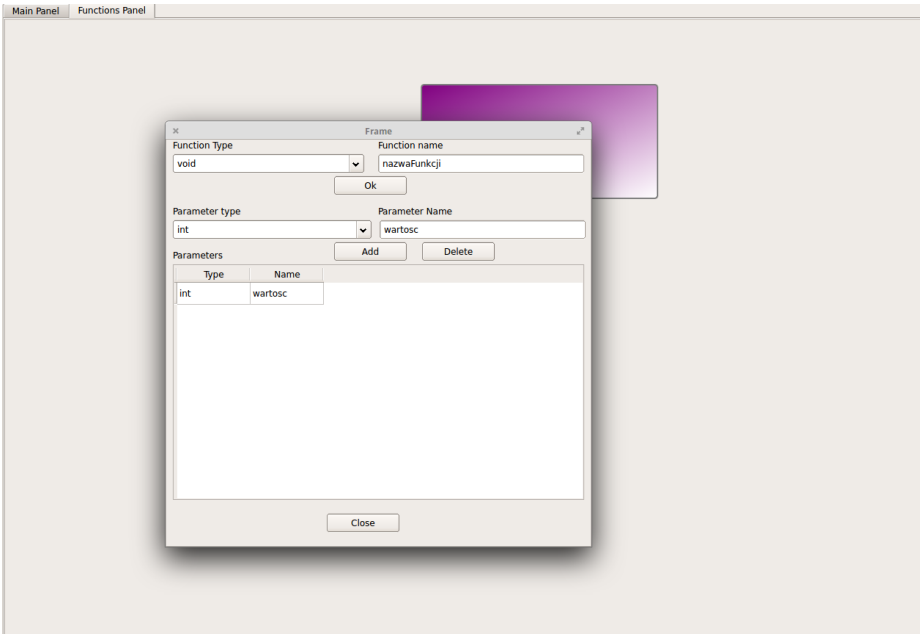

Rysunek 3.1: Widok głównego okna aplikacji z programem wypisującym “Hello world” 10 razy.

Po uruchomieniu aplikacji zobaczymy listę predefiniowanych komponentów, opisanych w poprzednim rozdziale oraz przestrzeń roboczą z dwoma zakładkami, główną i funkcyjną. Obsługa aplikacji jest prosta, wybieramy blok, którego chcemy użyć i klikamy w przestrzeni roboczej w miejsce w którym chcemy umieścić dany komponent. Działanie programu to przejście z góry na dół przestrzeni roboczej i wykonanie instrukcji wynikających z poszczególnych bloków.

### 3.3.1 Zakładka funkcyjna

Zakładka funkcyjna umożliwia tworzenie funkcji. Funkcja jest bytem analogicznym do tego z C++. Może zostać wywołana poprzez użycie jej nazwy w kodzie C++. Możliwe jest wywoływanie jednej funkcji przez drugą, niezależnie od kolejności ich deklaracji. Ciało funkcji jest blokiem takim samym jak główna zakładka programu, dlatego w funkcji możemy umieszczać wszystkie pozostałe bloki oprócz innego bloku funkcji. Blok funkcyjny może zostać umieszczony tylko jako podstawa w zakładce funkcyjnej. Możliwość rozwoju funkcjonalności komponentu dynamicznego, może spowodować utratę znaczenia przez funkcje. W

przypadku dodania własności typu zwracanego do dynamicznego komponentu, funkcje znane z C++ straciłby rację bytu.




Rysunek 3.2: Widok tworzenia funkcji.

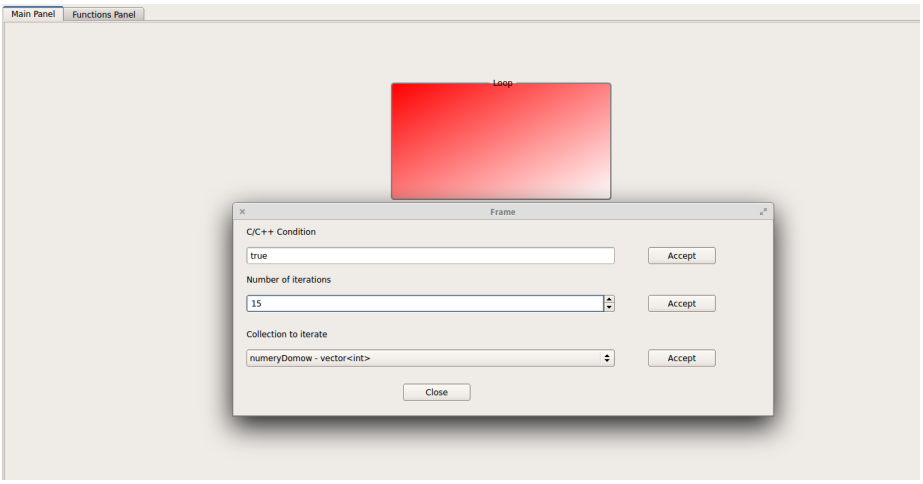
### 3.3.2 Predefiniowane bloki

Bloki predefiniowane zostały opisane w rozdziale 3.2. Zmienne zadeklarowane w bloku będą widoczne we wszystkich blokach znajdujących się wewnątrz niego, ale nie na zewnątrz bloku w którym były zdefiniowane. Funkcje none i delete, pozwalają odpowiednio zwolnić aktualnie wybrany blok, aby uniknąć przypadkowego dodania go do programu oraz usunięcie kliknięgo kursorem bloku. Na rysunku 3.3 możemy zobaczyć jak wygląda tworzenie nowego bloku pętli.

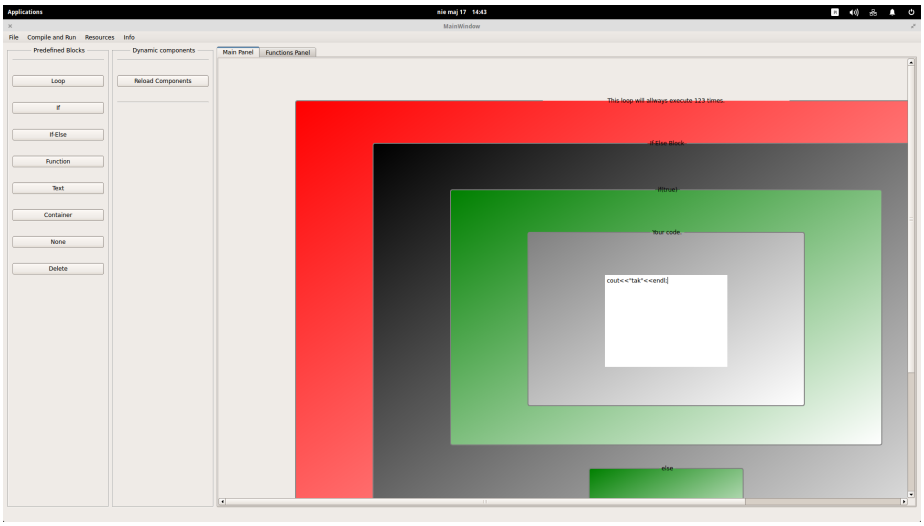
Jak widać możemy stworzyć pętlę typu while, wykonującą obroty dopóki spełniony jest zadany warunek. Pętlę wykonującą się określoną ilość razy, tak na przykład zrealizowany jest program z rysunku 3.1. Trzecią możliwością jest pętla foreach iterująca po zdefiniowanej wcześniej kolekcji. W przypadku pętli foreach otrzymujemy identyfikator, będący referencją do elementu, który aktualnie rozpatrujemy. Interfejs informuje użytkownika o tym jak działają poszczególne warianty pętli, dlatego nie musi on znać założeń języka, aby wiedzieć jakiego efektu powinien się spodziewać.

### 3.3.3 Zwijanie bloków

Bloki są dość duże w celu wyraźnego pokazania granic i zwiększenia łatwości odczytu. Jednak przy zagnieżdżaniu większej ilości bloków może być to problematyczne. Bloki przestaną mieścić się w szerokości ekranu i program będzie trudny do odczytania.

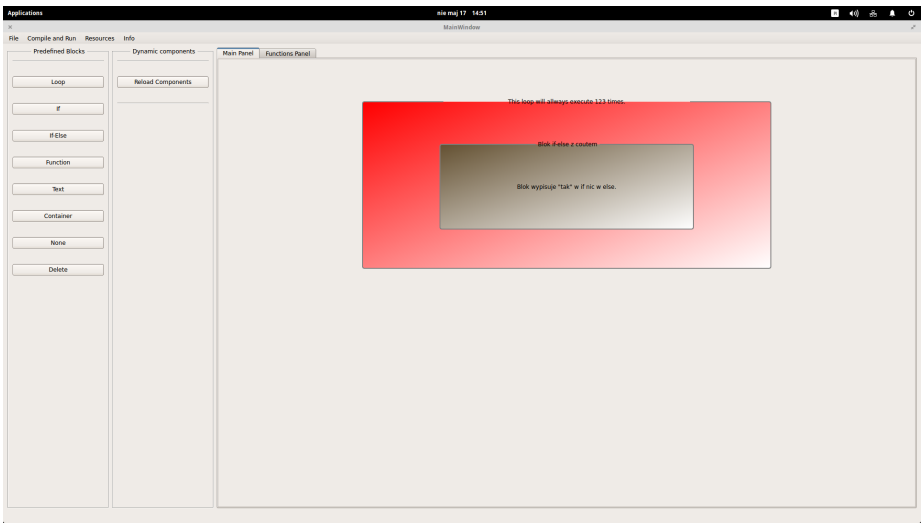



Rysunek 3.3: Tworzenie bloku pętli.




Rysunek 3.4: Bloki przed zwinięciem.

Po kliknięciu na blok prawym przyciskiem myszy i wybraniu opcji "switch form" zostanie on zwinięty do pojedynczego kwadracika. Cała wewnętrzna zawartość zostanie ukryta. Możemy wtedy użyć opcji rename, również dostępnej z poziomu menu kontekstowego bloku. Blok można nazwać i nadać mu opis, który opisze co dzieje się w środku. Do zwiniętego bloku nie można dodawać elementów. Przyczyna jest prosta. Nie wiadomo, w którym miejscu bloku miałby się pojawić dodawany element, dlatego jeśli chcemy zmodyfikować działanie zwiniętego bloku, należy go rozwinąć i zwinąć ponownie po zakończeniu modyfikacji. Można zwijać bloki wielokrotnie. W zobrazowanym przypadku można by było zwinąć także blok pętli, otrzymując tylko jeden blok zawierający cały program.



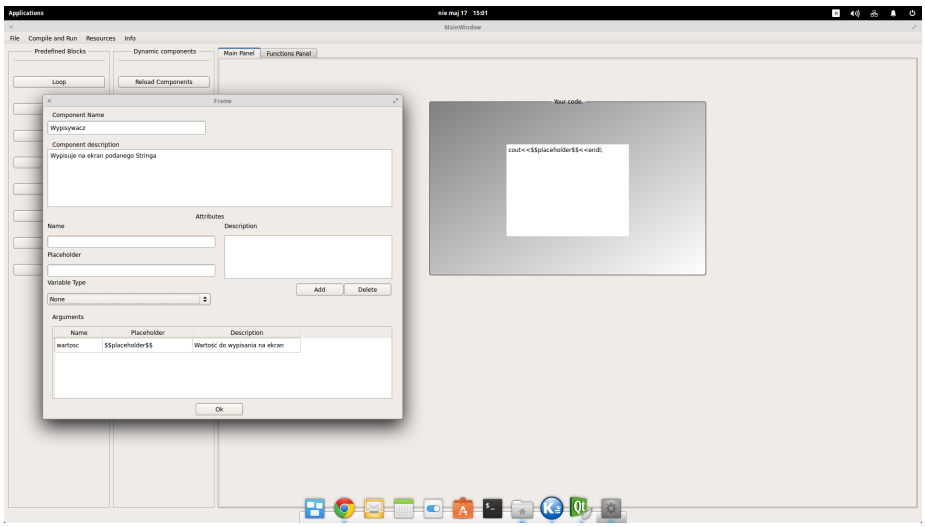

Rysunek 3.5: Po zwinięciu.

### 3.3.4 Dynamiczne komponenty

Biblioteki, czyli zbiory funkcji są główną przyczyną wzrostu łatwości programowania. Mając do dyspozycji tysiące gotowych rozwiązań, programować jest dużo łatwiej. Operacje sortowania, czy struktury danych takie jak, choćby kolejka priorytetowa ze standardowej biblioteki szablonów znacznie przyspieszają tworzenie kodu. Język tej aplikacji również posiada takie możliwości. Oprócz oczywistej opcji zapisania całego programu i otworzenia go później, język udostępnia możliwość eksportowania dowolnej jego części jako komponent. Komponent jest pojedynczym plikiem z rozszerzeniem .cmp zawierającym opis eksportowanych bloków w języku xml. Pisanie komponentów jest trudniejsze niż pisanie gotowego kodu, ponieważ wymaga, zrozumienia pewnej abstrakcji reprezentowanej przez "placeholdery" czyli elementy, które podczas wykonania programu zostaną zastąpione przez oczekiwane wartości. Spójrzmy na prosty komponent, którego celem jest wypisanie podanego napisu na ekran. Poprzednio widzieliśmy, że aby wypisać coś na ekran należy napisać fragment kodu w C++. Dzięki posiadaniu gotowego komponentu nie będzie to konieczne. Najpierw jednak trzeba ten komponent stworzyć.

Tworzymy blok tekstowy, umieszczamy w nim instrukcje cout, zamiast wartości wpisujemy jednak placeholder. W naszym przypadku będzie to $$placeholder$$. Naszym zadaniem jest zapewnić, aby placeholder nie był użyty w żadnym nieoczekiwanym miejscu, dlatego dobrze jest opakować go w symbole, które uniemożliwią przypadkowe pojawienie się takiego wyrażenia. W naszym przypadku są to $$ na początku i na końcu. Po uzupełnieniu bloku tekstowego, klikamy na niego prawym przyciskiem myszy i z menu kontekstowego wybieramy "Export as component". Pojawi się okno widoczne na rysunku 3.6.

Jest to okno definicji komponentu. Możemy wpisać jego nazwę i opis. Zostaną one wyświetlone w sposób podobny do tego w "zwiniętym" bloku. Następną część okna stanowi definicja atrybutów. Atrybut posiada nazwę, opis i



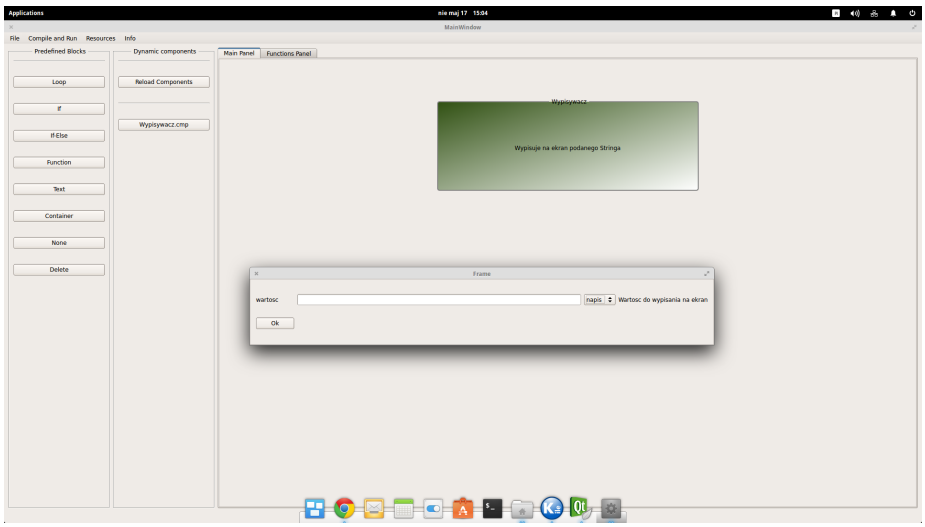

Rysunek 3.6: Definiowanie komponentu.
Celem niniejszej pracy jest stworzenie nowego graficznego języka programowania.

właśnie placeholder, czyli ciąg znaków, który zostanie zastąpiony właściwą wartością przy użyciu komponentu. Atrybut może także posiadać typ zmiennej. Jeśli wybierzemy typ zmiennej, to przy użyciu komponentu otrzymamy listę zmiennych tego typu, które możemy użyć w naszym komponencie. Po kliknięciu "ok", komponent zostanie wyeksportowany do katalogu "components", który zostanie założony w miejscu, w którym znajduje się plik wykonywalny aplikacji. Po kliknięciu przycisku "Reload Components" z menu "Dynamic components" nastąpi wczytanie komponentów ze wspomnianego katalogu. Ponieważ właśnie dodaliśmy komponent "Wypisywacz.cmp" pojawi się on na liście. Możemy go teraz użyć, tak samo jak używamy predefiniowanego bloku. Po kliknięciu w przestrzeni roboczej zobaczymy okno widoczne na rysunku 3.7.

Jest to lista atrybutów, z opisem i miejscem na wpisanie wartości. Jeśli wybraliśmy typ zmiennej, możemy wybrać wartość z listy. W tym przypadku wybrana została zmienna `napis` i to jej wartość zostanie wypisana na ekran. Mamy zatem komponent, który pozwala na wypisywanie napisów na standardowe wyjście. Do tego programista używający tego komponentu nie musi pisać nawet linii kodu w C++, aby go użyć.

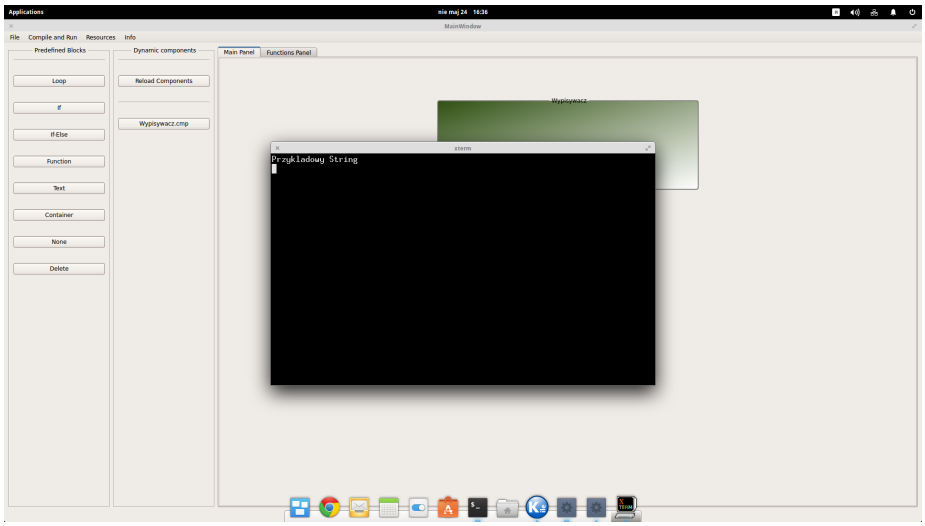
### 3.3.5 Kompilacja i uruchomienie programu

Program kompilowany jest kompilatorem `g++` z opcja `std=C++11`, co oznacza, że programy mogą korzystać z funkcji języka C++11, takich jak na przykład wyrażenia lambda. Kompilacja odbywa się poprzez wybranie opcji "Compile" z paska menu. Po skompilowaniu powstaje plik wykonywalny `result` w katalogu "sourcecode", który znaleźć można w miejscu, w którym znajduje się plik wykonywalny aplikacji. We wspomnianym katalogu znajdują się także pliki źródłowe wszystkich programów, które były kompilowane. Aplikacja posiada także opcje "Compile and run", opcja ta, do poprawnego działania wymaga aby na kom-




Rysunek 3.7: Użycie komponentu.

puterze zainstalowany był program "xterm". Jeśli ten program jest obecny na komputerze to po kliknięciu "Compile and run" w naszym programie, w którym użyliśmy komponentu "Wypisywacz.cmp" otrzymamy wynik widoczny na rysunku 3.8.




Rysunek 3.8: Wynik działania programu "wypisywacz", z parametrem wziętym, ze zmiennej "napis".

# Bibliografia


[1] Merriam-Webster, An Encyclopædia Britannica Company, http://www.merriam-webster.com/dictionary/programming

[2] History of Computers, http://history-computer.com/Dreamers/Jacquard.html

[3] New Scientist magazine issue number 2791, http://www.newscientist.com/article/mg20827915.500-lets-build-babbages-ultimate-mechanical-computer.html

[4] Computer History Museum, http://www.computerhistory.org/babbage/adalovelace/

[5] Computer History Museum, http://www.computerhistory.org/atchm/wp-content/uploads/2015/01/Diagram_for_the_computation_of_Bernoulli_numbers.jpg

[6] TIOBE Software, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[7] The IBM 701 Speedcoding System, John Backus, http://www.computerhistory.org/collections/catalog/102653983

[8] Communications ot the ACM, http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext

[9] Słownik Języka Polskiego PWN, http://sjp.pwn.pl/sjp/paradygmat;2570598.html

[10] Programming Paradigms, Ray Toal, http://cs.lmu.edu/ ray/notes/paradigms/

[11] Ważniak MIM UW, http://wazniak.mimuw.edu.pl/index.php?title=Paradygmaty_programowania/Wyk%C5%82ad_1:_Co_to_jest_paradygmat_programowania%3F, Małgorzata Moczurad — Uniwersytet Jagielloński, Włodzimierz Moczurad — Uniwersytet Jagielloński