

POLITECHNIKA KRAKOWSKA
IM. TADEUSZA KOŚCIUSZKI
WYDZIAŁ INŻYNIERII ELEKTRYCZNEJ I KOMPUTEROWEJ
KIERUNEK INFORMATYKA

Mateusz BUCZEK
Wojciech KWIECIEN

**Środowisko do tworzenia schematów blokowych i
generacji kodu**

DYPLOMOWY PROJEKT INŻYNIERSKI
STUDIA NIESTACJONARNE

Promotor:
dr inż. Joanna STRUG

Kraków 2012

SPIS TREŚCI

SPIS TREŚCI.....	2
1. WSTĘP.....	5
1.1. CEL I ZAKRES PRACY	5
1.2. PODZIAŁ ZADAŃ	6
1.3. ZAWARTOŚĆ PRACY	6
2. ŚRODOWISKA WSPOMAGAJĄCE TWORZENIE SCHEMATÓW BLOKOWYCH.....	7
2.1. JAVABLOCK	7
2.2. ELBOX - LABORATORIUM INFORMATYKI (ELI)	8
2.3. MICROSOFT VISUAL PROGRAMMING LANGUAGE (VPL)	10
3. SPECYFIKACJA WYMAGAŃ.....	12
3.1. WYMAGANIA FUNKCJONALNE.....	12
3.1.1. Tworzenie i edycja schematu	12
3.1.2. Uruchomienie (symulacja) schematu	14
3.1.3. Wygenerowanie kodu źródłowego.....	14
3.2. WYMAGANIA NIEFUNKCJONALNE	15
4. WYKORZYSTANE TECHNOLOGIE.....	16
4.1. JĘZYK PROGRAMOWANIA I ŚRODOWISKO WYKONAWCZE	16
4.2. BIBLIOTEKI.....	16
4.3. NARZĘDZIA PROGRAMISTYCZNE	17
5. ARCHITEKTURA I IMPLEMENTACJA.....	18
5.1. STRUKTURA WYSOKOPOZIOMOWA	18
5.2. KLASA <i>APPLICATION</i>	19
5.3. PAKIET <i>DIAGRAM</i>	20
5.3.1. Klasa <i>BaseNode</i>	21
5.3.2. Klasa <i>ConditionalNode</i>	22
5.3.3. Klasa <i>EndNode</i>	23
5.3.4. Klasa <i>InputOutputNode</i>	24
5.3.5. Klasa <i>InputNode</i>	25
5.3.6. Klasa <i>OutputNode</i>	26
5.3.7. Klasa <i>ProcessingNode</i>	26
5.3.8. Klasa <i>StartNode</i>	27
5.3.9. Klasy <i>ThenLink</i> i <i>ElseLink</i>	28
5.4. PAKIET <i>EDITOR</i>	28
5.4.1. Klasa <i>DiagramEditor</i>	29
5.4.2. Pakiet <i>editor.action</i>	32

5.4.3.	<i>Pakiet editor.export</i>	34
5.4.4.	<i>Pakiet editor.form</i>	35
5.5.	PAKIET RUNTIME.....	37
5.5.1.	<i>Klasa DiagramRunner</i>	38
5.5.2.	<i>Klasa RunEngine</i>	39
5.5.3.	<i>Klasa ErrorHandler</i>	41
5.5.4.	<i>Klasa VariablesTableModel</i>	41
5.6.	PAKIET CODEGEN	42
5.6.1.	<i>Klasa CodeGeneration</i>	43
5.6.2.	<i>Interfejs ICodeGenerator</i>	44
5.6.3.	<i>Klasa CCodeGenerator</i>	46
5.6.4.	<i>Klasy JavaScriptCodeGenerator i WebPageCodeGenerator</i>	49
5.7.	PAKIET EVAL.....	50
5.7.1.	<i>Typy zmiennych</i>	50
5.7.2.	<i>Klasa EvalEngine</i>	53
5.7.3.	<i>Klasa Environment</i>	53
5.7.4.	<i>Pakiet eval.error</i>	54
5.7.5.	<i>Klasa SyntaxValidator</i>	55
5.8.	PAKIET EVENT.....	56
5.8.1.	<i>Klasa EventManager i interfejs EventHandler</i>	56
5.8.2.	<i>Typy zdarzeń</i>	58
5.9.	PAKIET LOG	58
5.10.	PAKIET UI	59
5.11.	POZOSTAŁE KLASY: <i>CONFIG I I18N</i>	59
6.	TESTOWANIE	61
6.1.	TESTY AUTOMATYCZNE	61
6.2.	TESTY RĘCZNE	63
7.	INSTRUKCJA UŻYTKOWNIKA	64
7.1.	OPIS PROGRAMU.....	64
7.1.1.	<i>Wymagania</i>	64
7.1.2.	<i>Uruchomienie</i>	64
7.2.	TWORZENIE I EDYCJA DIAGRAMÓW	64
7.2.1.	<i>Tworzenie diagramu</i>	64
7.2.2.	<i>Dodawanie i edycja bloków</i>	64
7.2.3.	<i>Cofanie/ponawianie operacji</i>	73
7.2.4.	<i>Zapisywanie i zamykanie diagramu</i>	73
7.3.	WYKONYWANIE DIAGRAMU	74
7.3.1.	<i>Uruchomienie</i>	74
7.3.2.	<i>Zatrzymanie wykonywania</i>	75

7.4.	GENERACJA KODU	75
7.4.1.	<i>Generowanie kodu</i>	75
7.4.2.	<i>Zapisywanie wygenerowanego kodu</i>	76
7.5.	KONFIGURACJA	76
8.	PODSUMOWANIE	78
9.	BIBLIOGRAFIA	80
9.1.	LITERATURA	80
9.2.	ŹRÓDŁA INTERNETOWE	80
10.	SPIS ILUSTRACJI	82

1. WSTĘP

Jedną z metod reprezentacji algorytmu jest schemat blokowy, będący układem figur geometrycznych (bloków) połączonych ze sobą liniami, przy czym każdy z bloków reprezentuje określony rodzaj operacji, natomiast linie wskazują przepływ sterowania między nimi.

W porównaniu do innych sposobów reprezentacji algorytmów, takich jak pseudokod czy implementacja w języku programowania, schematy blokowe oferują większy stopień abstrakcji, pozwalając na graficzne przedstawienie podstawowych konstrukcji programistycznych (sekwencja, instrukcja warunkowa, pętla) w sposób niezależny od konkretnej składni. Z tego względu są one często wykorzystywane przy projektowaniu i weryfikacji algorytmów oraz dydaktyce – w nauczaniu algorytmiki oraz podstaw programowania [9, 13].

Efektywne wykorzystanie schematów blokowych w wyżej wymienionych obszarach (w szczególności dydaktyce) wymaga odpowiednich narzędzi. Ręczne konstruowanie czy analiza jest w tym przypadku procesem czasochłonnym i mało podatnym na zmiany - konieczność modyfikacji struktury schematu jest obciążona stosunkowo dużym kosztem. Znacznie efektywniejsze wydaje się wykorzystanie w tym celu narzędzi komputerowych – aplikacji pozwalających na budowanie schematów z predefiniowanych bloków oraz ich interaktywne testowanie.

1.1. Cel i zakres pracy

Celem pracy jest zaprojektowanie i implementacja aplikacji typu *desktop* umożliwiającej tworzenie schematów blokowych oraz generowanie kodu źródłowego na ich podstawie. Aplikacja będzie w szczególności pozwalała na:

- Tworzenie oraz edycję podstawowych elementów strukturalnych spotykanych w schematach blokowych [9, 13] za pomocą graficznego interfejsu użytkownika (GUI)
- Przechowywanie utworzonych schematów w formie plików
- Interaktywne testowanie i uruchamianie schematów
- Generowanie kodu w języku C++ będącego kompletnym programem odpowiadającym danemu schematowi blokowemu.

1.2. Podział zadań

- Mateusz Buczek:
 - komponenty do testowania i uruchamiania schematów;
 - generowanie kodu na podstawie schematów, w tym kodu w języku C++;
- Wojciech Kwiecień:
 - edytor schematów blokowych;
 - zapisywanie/odczytywanie schematów do/z plików;

1.3. Zawartość pracy

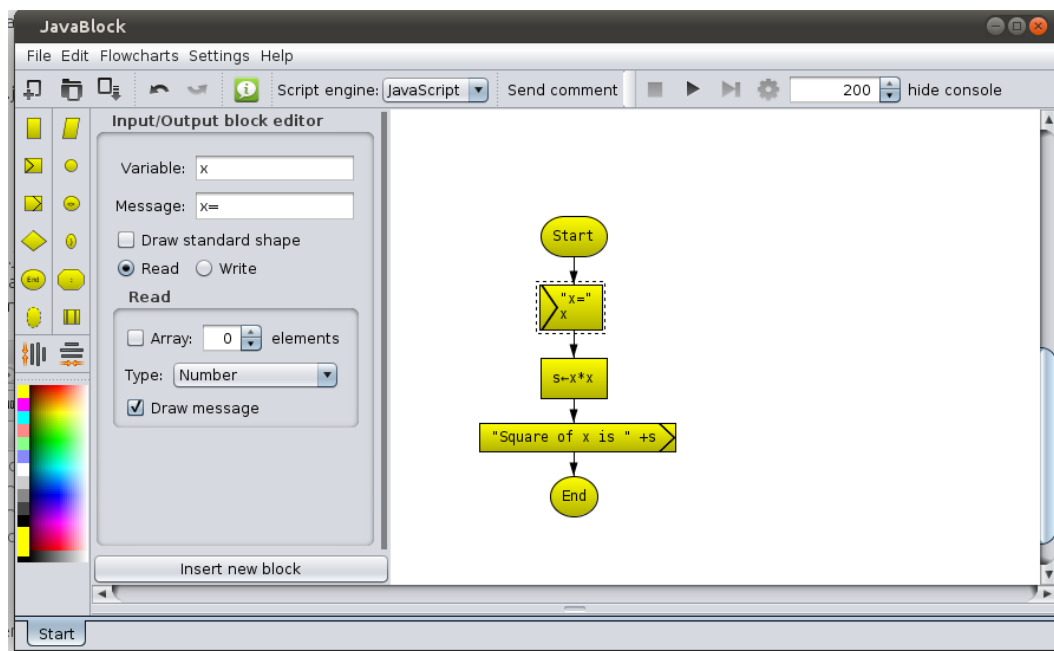
W kolejnym rozdziale opisano trzy istniejące środowiska/aplikacje o podobnym zakresie funkcjonalnym z uwzględnieniem różnic w stosunku do rozwiązania proponowanego w niniejszej pracy. Rozdział 3. zawiera opis wymagań funkcjonalnych i niefunkcjonalnych projektowanego systemu; w rozdziale 4. przedstawiono technologie wykorzystane przy realizacji projektu. Rozdział 5. opisuje architekturę i implementację aplikacji. W rozdziale 6. omówiono krótko proces testowania stworzonego systemu. Rozdział 7. zawiera instrukcję użytkownika. Ostatni rozdział - 8. poświęcono podsumowaniu najważniejszych kwestii związanych z realizacją projektu.

2. ŚRODOWISKA WSPOMAGAJĄCE TWORZENIE SCHEMATÓW BLOKOWYCH

2.1. JavaBlock

JavaBlock [20] jest darmową (udostępnioną na licencji LGPL¹) aplikacją autorstwa Jakuba Koniecznego. Posiada następujące funkcjonalności:

- Tworzenie i uruchamianie schematów blokowych za pomocą graficznego interfejsu użytkownika
- Generowanie kodu w językach JavaScript i Python
- Możliwość publikacji wygenerowanego kodu w serwisie Pastebin²
- Możliwość generowania liniowej grafiki w oparciu o prostą implementację języka LOGO (tzw. „grafika żółwia”)



Rysunek 1. Interfejs aplikacji JavaBlock

(Źródło: <http://javablock.sourceforge.net/img/JB06.png>)

Spośród omawianych w tym rozdziale rozwiązań, *JavaBlock* jest z punktu widzenia modelowania interfejsu użytkownika oraz wykorzystanej technologii (Java) środowiskiem stosunkowo najbardziej zbliżonym do prezentowanego w niniejszej pracy. Do podstawowych różnic należą natomiast:

¹ GNU Lesser General Public License - <http://www.gnu.org/licenses/lgpl.html>

² <http://pastebin.com/>

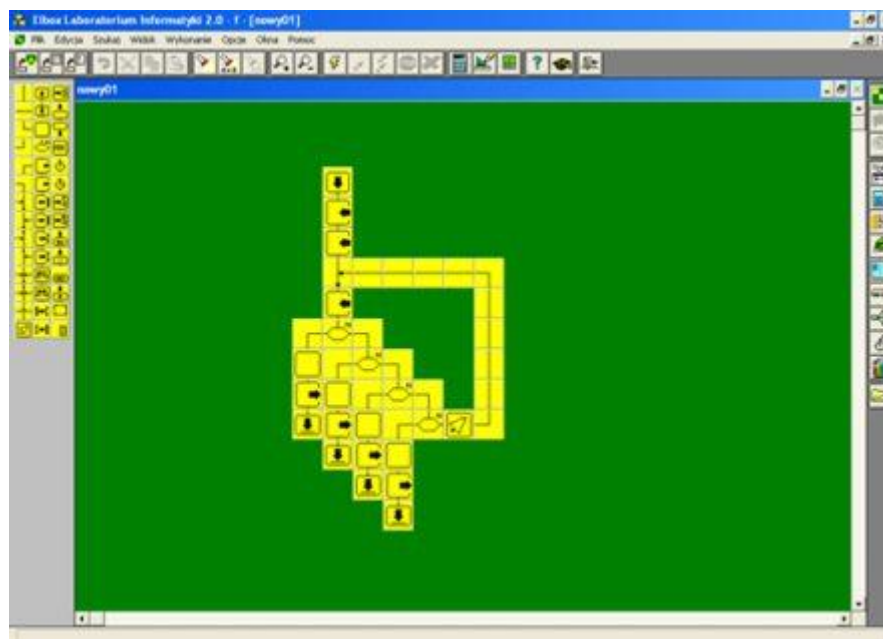
- brak możliwości generowania kodu w języku C++
- brak możliwości bezpośredniego zapisu wygenerowanego kodu do pliku
- brak możliwości definiowania kilku bloków startowych w obrębie jednego diagramu (jako podprogramy/procedury)

2.2. Elbox - Laboratorium Informatyki (ELI)

Elbox - Laboratorium Informatyki to komercyjny pakiet programów autorstwa firmy ELBOX Pomoce Dydaktyczne Sp. z o.o. przeznaczony do wspomagania nauczania elementów informatyki na poziomie szkoły podstawowej, gimnazjum oraz średniej. Dostępny jest w kilku wersjach: 2.0, 2.0 Pro, 2.0 Plus; wersja 2.1 (ostatnia) została wydana w 1997 roku [9].

Pakiet *Laboratorium* składa się z oprogramowania działającego w środowisku Windows, pozwalającego na budowanie i uruchamianie schematów blokowych (nazywanych algorytmami). Algorytmy składane są z gotowych elementów (klocków) na jednej lub wielu planszach. Podstawowa paleta oferuje następujące klocki:

- Początek algorytmu - wyświetla podany komunikat, rozpoczyna wykonywanie algorytmu (może wystąpić tylko raz)
- Koniec algorytmu - wyświetla podany komunikat, kończy wykonanie algorytmu
- Obliczenia - wykonuje kolejno podane obliczenia
- Sprawdzenie warunku - rozgałęzienie przepływu sterowania w zależności od podanego warunku
- Wprowadzenie danej - przypisuje zmiennej wartość podaną przez użytkownika
- Wyprowadzenie wyniku - wyświetla komunikat i obliczoną wartość podanego wyrażenia
- Odczyt z tablicy - przypisuje zmiennej wartość odczytaną z pola tablicy
- Zapis do tablicy - wpisuje podaną wartość do pola tablicy
- Wywołanie procedury - przejście do wykonania procedury
- Początek procedury - rozpoczyna wykonanie procedury
- Koniec procedury - kończy procedurę i powoduje powrót do miejsca wywołania



Rysunek 2. Interfejs pakietu Elbox - Laboratorium Informatyki
(Źródło: <http://cacheee.ovh.org/Szablony/eliobszarrpacy.jpg>)

Połączenia między poszczególnymi blokami również tworzone są z gotowych klocków (segmentów), co znacząco różni się od rozwiązania proponowanego w niniejszej pracy, gdzie bloki łączą się za pomocą dowolnie edytowalnych linii (dzięki czemu użytkownik ma większą swobodę w układaniu elementów schematu).

Składnia instrukcji dostępnych w ramach bloku obliczeń inspirowana jest językiem Pascal (stosowanie `:=` jako operatora przypisania) przy jednoczesnym wprowadzeniu znacznych ograniczeń: nie oferuje możliwości dostępu do tablic oraz wywoływania procedur, które to operacje muszą być realizowane za pomocą innych, dedykowanych bloków. W proponowanym podejściu natomiast obydwie konstrukcje są dozwolone zarówno w obrębie bloku przetwarzania jak i warunkowego, wykorzystując składnię znaną z języka C.

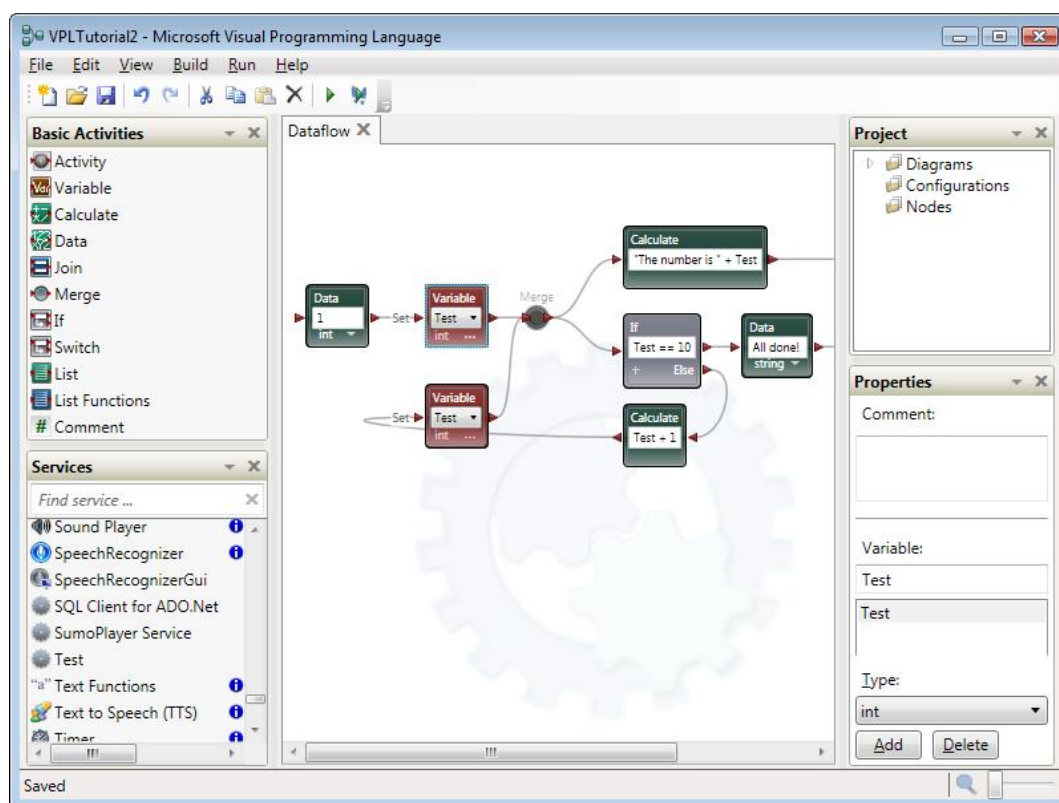
W ELI każdy algorytm (schemat) musi posiadać dokładnie jeden punkt startu (klock START), a procedury są definiowane w obrębie specjalnych bloków (początek/koniec procedury), co istotnie różni się od modelu proponowanego w niniejszej pracy, gdzie każdy blok startowy interpretowany jest jako procedura/podprogram identyfikowany nazwą oraz (opcjonalną) listą parametrów formalnych. Pojedynczy schemat może więc zawierać kilka takich bloków, przy czym jeden z nich (o nazwie *Main*, nie posiadający parametrów formalnych) jest traktowany jako główny - rozpoczynający przepływ sterowania w całym schemacie. Model ten odpowiada konstrukcji programów

w językach proceduralnych, zwłaszcza w języku C, gdzie wymagane jest zdefiniowanie co najmniej jednej procedury/funkcji (o nazwie *main*), od której rozpoczyna się wykonywanie programu [7].

2.3. Microsoft Visual Programming Language (VPL)

Microsoft Visual Programming Language (VPL lub MVPL) jest środowiskiem wchodzącym w skład pakietu Microsoft Robotics Studio przeznaczonym do programowania graficznego, opartego na modelu przepływu danych (*dataflow-based programming*).

Schemat VPL składa się z sekwencji połączonych ze sobą aktywności/procesów (*activities*) reprezentowanych przez bloki o zdefiniowanych zestawach wejść i wyjść. Pojedynczy blok może odpowiadać elementarnej konstrukcji sterowania (instrukcja warunkowa, przypisanie zmiennej), wbudowanemu modułowi (zamiana tekstu na głos, rozpoznawanie mowy, zliczanie itp.) lub nowej aktywności zdefiniowanej przez użytkownika.



Rysunek 3. Interfejs środowiska Microsoft VPL

(Źródło: <http://i.msdn.microsoft.com/dynimg/IC234746.png>)

Spośród omawianych środowisk, MVPL wykazuje najbardziej znaczące różnice w stosunku do projektu prezentowanego w niniejszej pracy, co wynika z przyjętego mode-

lu programowania - opartego na przepływie danych. W modelu tym, połączenia między blokami schematu odpowiadają kierunkom przepływu danych, nie zaś - jak w przypadku modelu imperatywnego - kierunkowi przepływu sterowania. Poszczególne bloki reprezentują operacje wykonywane w momencie pojawienia się na ich wejściu odpowiednich danych, przetwarzanie zależy więc bezpośrednio od przekazywanych komunikatów (i może odbywać się w sposób równoległy).

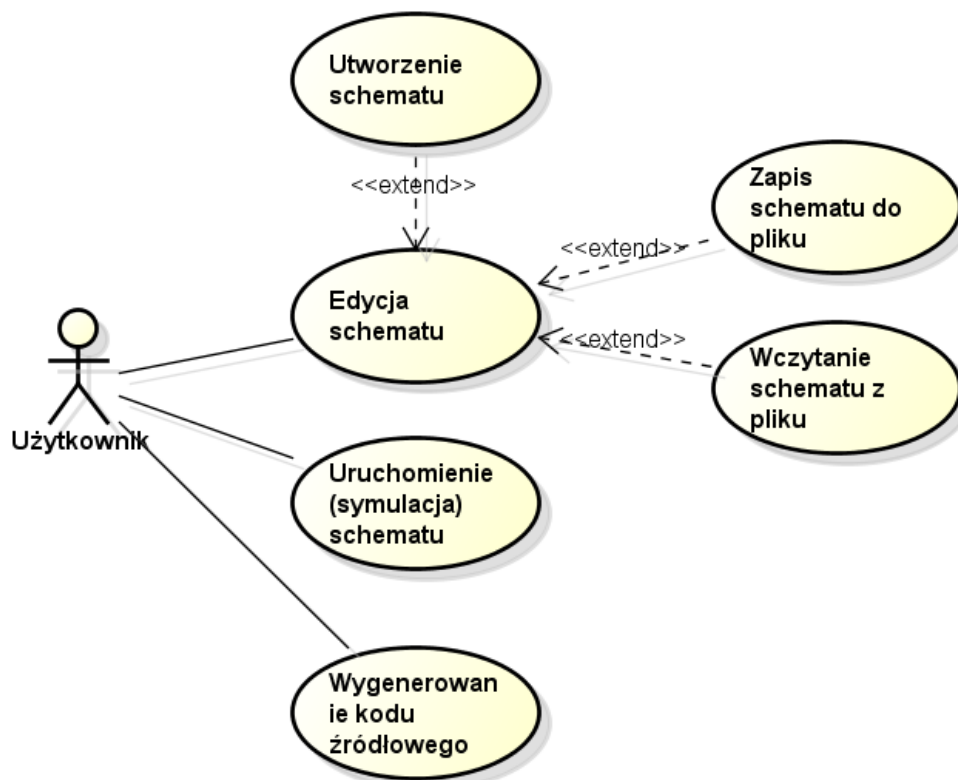
Ze względu na model sterowany danymi, diagramy MVPL wymagają reprezentacji wszystkich typowych konstrukcji (nawet tak elementarnych jak przypisanie/pobranie wartości zmiennej) jako procesów przekształcających komunikaty wejściowe na wyjściowe, co może powodować znaczne zwiększenie stopnia złożoności w stosunku do odpowiadających im schematów w modelu imperatywnym.

Każda z przedstawionych tu pokrótce aplikacji prezentuje nieco odmienne podejście do zagadnienia tworzenia schematów blokowych, determinowane specyficznym gronem użytkowników, do których jest adresowana (pakiet dydaktyczny ELI) czy też specjalistyczną dziedziną zastosowań (robotyka w przypadku Microsoft VPL). Największym stopniem uniwersalności wydaje się charakteryzować program JavaBlock (również pod względem wymagań systemowych - jako jedyny może być uruchamiany niezależnie od systemu operacyjnego dzięki wykorzystaniu technologii Java), nie oferuje jednak możliwości generowania kodu w języku C++. Projekt prezentowany w niniejszej pracy ma na celu realizację tej funkcjonalności przy jednoczesnym zachowaniu podobnego stopnia uniwersalności.

3. SPECYFIKACJA WYMAGAŃ

3.1. Wymagania funkcjonalne

Wymagania funkcjonalne stawiane wobec realizowanej aplikacji można podzielić na trzy grupy: tworzenie i edycja schematów blokowych, uruchamianie (symulowanie) schematów oraz generowanie kodu źródłowego. Rysunek 4. przedstawia odpowiadający im diagram przypadków użycia.



Rysunek 4. Diagram przypadków użycia
(Źródło: opracowanie własne)

Poszczególne grupy funkcjonalności zostały opisane w kolejnych podrozdziałach.

3.1.1. Tworzenie i edycja schematu

Ta grupa przypadków użycia pozwala Użytkownikowi na budowanie schematu w oparciu o typowe elementy strukturalne [9]: blok startu, blok zakończenia, blok obliczeniowy (przetwarzania), blok warunkowy (decyzyjny), blok wejścia (wprowadzenia danych), blok wyjścia (wyprowadzenia danych), łącznik (połączenie między blokami).

Edycja schematu – scenariusz główny:

1. Użytkownik tworzy nowy schemat lub wczytuje istniejący schemat z pliku.

2. Zostaje utworzony nowy obszar roboczy dla schematu.
3. Użytkownik dodaje, usuwa lub modyfikuje elementy schematu:
 - i. dodaje nowy blok
 - ii. usuwa istniejący blok
 - iii. zmienia atrybuty wybranego bloku
 - iv. tworzy połączenie między dwoma blokami
 - v. usuwa połączenie między dwoma blokami
4. Użytkownik może cofnąć lub powtórzyć ostatnio wykonaną operację, zapisać schemat do pliku lub wyeksportować schemat do wybranego formatu.

Edycja schematu – możliwe warianty i rozszerzenia:

- 1a. Użytkownik tworzy nowy schemat wybierając opcję otwarcia nowego schematu.
- 1b. Użytkownik wczytuje istniejący schemat z pliku:
 - i. Użytkownik wybiera opcję otwarcia istniejącego schematu.
 - ii. System udostępnia listę istniejących schematów.
 - iii. Użytkownik wybiera żądany schemat blokowy.
- 3a. Użytkownik wprowadza niepoprawne wartości atrybutów dla edytowanego bloku – system informuje Użytkownika o błędzie, zmiany nie są zapisywane.
- 4a. Użytkownik zapisuje schemat do pliku:
 - i. Użytkownik wybiera opcję zapisania schematu do pliku.
 - ii. Użytkownik wybiera docelową lokalizację oraz podaje nazwę pliku.
 - iii. Schemat zostaje zapisany do wybranego pliku.
- 4b. Użytkownik eksportuje schemat do wybranego formatu:
 - i. Użytkownik wybiera opcję eksportu schematu.
 - ii. Użytkownik wybiera żądany format, docelową lokalizację oraz podaje nazwę pliku.
 - iii. Schemat zostaje zapisany do pliku w wybranym formacie.
- 4c. Użytkownik cofa lub ponawia ostatnio wykonaną operację – ostatnia operacja zostaje odpowiednio – wycofana lub wykonana ponownie.

3.1.2. Uruchomienie (symulacja) schematu

Ten przypadek użycia pozwala Użytkownikowi na przetestowanie działania danego schematu poprzez wykonanie sekwencji bloków zaczynającej się od bloku startowego zgodnie ze zdefiniowanym przepływem sterowania.

Scenariusz główny:

1. Użytkownik wybiera blok startowy, od którego ma być rozpoczęte wykonywanie schematu. Opcjonalnie Użytkownik może podać wartości argumentów, z którymi ma zostać wywołany blok startowy.
2. Symulacja schematu zostaje rozpoczęta począwszy od wybranego bloku startowego.
3. W trakcie symulacji aktualnie wykonywany blok jest wyróżniany kolorem a po jego wykonaniu aktualizowany jest panel do śledzenia wartości zmiennych.
4. Symulacja zostaje zakończona po osiągnięciu ostatniego bloku schematu lub w wyniku zatrzymania przez Użytkownika.

Możliwe warianty i rozszerzenia:

- 2a. Nie znaleziono bloku startowego o podanej nazwie – wyświetlony zostaje komunikat o błędzie, symulacja nie zostaje rozpoczęta.
- 3a. Wystąpił błąd czasu wykonania – symulacja zostaje zatrzymana z odpowiednim komunikatem o błędzie, ostatnio wykonany blok jest wyróżniony kolorem.

3.1.3. Wygenerowanie kodu źródłowego

Ten przypadek użycia umożliwia Użytkownikowi wygenerowanie kodu źródłowego odpowiadającego danemu schematowi.

Scenariusz główny:

1. Użytkownik wybiera język programowania, w którym ma być wygenerowany kod. W szczególności, Użytkownik ma możliwość wyboru jednego z następujących generatorów:
 - i. C++ - generuje kod w języku C++ programu konsolowego, który po skompilowaniu można uruchomić pod kontrolą wiersza poleceń
 - ii. JavaScript – generuje kod w języku JavaScript

- iii. Strona internetowa – generuje statyczną stronę internetową w formacie HTML zawierającą wykonywalny kod JavaScript oraz obrazek przedstawiający dany schemat
 - 2. Wygenerowany kod zostaje wyświetlony w edytowalnym polu tekstowym.
- Możliwe warianty i rozszerzenia:

- 3. Użytkownik zapisuje wygenerowany kod do pliku.

3.2. Wymagania niefunkcjonalne

- Interfejs aplikacji
 - Aplikacja powinna posiadać graficzny interfejs użytkownika (GUI - *Graphical User Interface*) oferujący dostęp do wszystkich wymaganych funkcjonalności
 - Interfejs powinien w sposób przejrzysty odpowiadać trzem głównym grupom funkcjonalnym: tworzeniu i edycji schematów, uruchamianiu schematów oraz generowaniu kodu; przy czym część edycyjna interfejsu powinna posiadać obszar roboczy zapewniający komfort pracy
 - Interfejs powinien być intuicyjny z punktu widzenia użytkownika i stosować ogólnie przyjęte standardy i konwencje spotykane w aplikacjach typu *desktop* (lokalizacja i układ głównego menu, sposób zamykania aplikacji, otwieranie i zapisywanie plików, obsługa myszy i skrótów klawiaturowych itp.)
- Przenośność (wieloplatformowość)
 - Aplikacja powinna prawidłowo działać z każdym graficznym systemem operacyjnym mającym zainstalowane standardowe środowisko JRE (*Java Runtime Environment*) w wersji 6 lub nowszej
- Łatwość instalacji i uruchamiania
 - Aplikacja nie powinna wymagać instalacji
 - Aplikacja powinna oferować jeden plik wykonywalny, dający się uruchomić w sposób standardowy dla danego systemu operacyjnego (zarówno za pomocą myszy jak i wiersza poleceń).

4. WYKORZYSTANE TECHNOLOGIE

Technologie użyte przy realizacji projektu zostały dobrane pod kątem odpowiedniości w stosunku do stawianych wymagań funkcjonalnych i нефunkcjonalnych oraz pod kątem łatwości ich wykorzystania.

4.1. Język programowania i środowisko wykonawcze

Jako podstawowy język implementacyjny i zarazem środowisko wykonawcze wykorzystano platformę Java (*Java Standard Edition*, wersja 6). Decydujące znaczenie miała w tym przypadku przenośność między różnymi systemami operacyjnymi jaką zapewnia ta platforma (dzięki wykorzystaniu maszyny wirtualnej JVM abstrahującej szczegóły konkretnej architektury systemowej [1]). W ten sposób stworzona aplikacja, bez potrzeby modyfikowania kodu źródłowego, może być uruchamiana zarówno pod kontrolą systemów z rodziny Microsoft Windows jak i w środowiskach UNIX-owych (np. GNU Linux, Mac OS X) z zainstalowanym środowiskiem JRE (*Java Runtime Environment*).

Równie ważnym kryterium wyboru była szeroka dostępność wysokiej jakości bibliotek (np. Swing, JUnit) i narzędzi programistycznych (np. IntelliJ IDEA, Apache Ant) wspierających proces tworzenia, debugowania i testowania aplikacji.

4.2. Biblioteki

Przy implementacji projektu, oprócz standardowej biblioteki klas dostępnej w ramach platformy Java, zostały wykorzystane następujące biblioteki:

- Swing [3] – biblioteka umożliwiająca tworzenie graficznych interfejsów użytkownika w sposób niezależny od konkretnego systemu operacyjnego; oferuje bogaty zestaw standardowych komponentów i kontrolek (przyciski, palety menu, okna dialogowe, panele itp.); biblioteka została wykorzystana do stworzenia całości interfejsu użytkownika.
- JDiagram [1] - biblioteka wspomagająca konstruowanie grafów, diagramów i wykresów w oparciu o komponent Swing; oferuje bogaty interfejs programistyczny pozwalający na modyfikowanie poszczególnych elementów diagramu oraz modelowanie interakcji z użytkownikiem; biblioteka została wykorzystana w edytorze schematów blokowych.
- Rhino [18] – biblioteka implementująca silnik języka JavaScript; pozwala na uruchamianie skryptów JavaScript z poziomu kodu napisanego w Javie; w

omawianej aplikacji biblioteka została wykorzystana przy symulacji działania schematów blokowych.

- JUnit [4] - biblioteka do tworzenia testów jednostkowych (*unit tests*) dla programów napisanych w Javie; biblioteka została wykorzystana do stworzenia zestawu testów pozwalających na weryfikację poprawności kluczowych komponentów i funkcjonalności aplikacji (generator kodu źródłowego, walidator składni, uruchamianie schematów).
- Apache Commons Codec [19] - biblioteka wspomagająca proces kodowania/dekodowania danych tekstowych i binarnych; w projekcie została wykorzystana przy osadzaniu graficznej reprezentacji schematów blokowych w ramach strony internetowej (kodowanie Base64) .

4.3. Narzędzia programistyczne

W trakcie prac nad projektem użyto następujących narzędzi programistycznych:

- IntelliJ IDEA [12] - zintegrowane środowisko programistyczne (IDE) posiadające zaawansowane mechanizmy do analizy i refaktoryzacji kodu źródłowego oraz wbudowaną integrację z innymi narzędziami deweloperskimi, takimi jak Subversion, Apache Ant i JUnit.
- Apache Ant [14] - narzędzie służące do automatyzacji procesu budowy oprogramowania napisanego w Javie; w projekcie zostało wykorzystane w celu zautomatyzowania kompilacji kodu do pojedynczego wynikowego pliku wykonywalnego (JAR).
- Subversion [15] - system kontroli wersji; narzędzie to zostało wykorzystane do synchronizacji pracy nad kodem źródłowym ze względu na zespołowy charakter projektu.

5. ARCHITEKTURA I IMPLEMENTACJA

Rozdział ten opisuje architekturę i implementację aplikacji - najważniejsze moduły i komponenty wraz z realizowanymi przez nie funkcjonalnościami i wzajemnymi powiązaniami - zarówno na poziomie koncepcyjnym jak i kodu źródłowego³.

5.1. Struktura wysokopoziomowa

Aplikacja została zaprojektowana w sposób modułowy, z wykorzystaniem mechanizmu pakietów języka Java. Moduły realizujące kluczowe funkcjonalności opisane w Rozdziale 3.1. to:

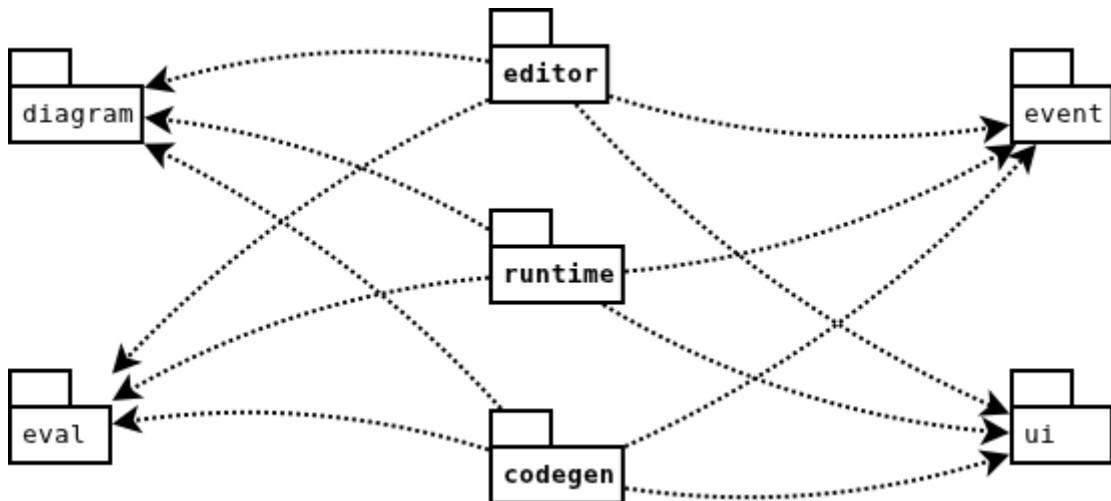
- *editor* - tworzenie i edycja schematów blokowych
- *runtime* - uruchamianie (symulowanie) schematów
- *codegen* - generowanie kodu źródłowego

Pozostałe pakiety mają charakter pomocniczy i grupują funkcje wykorzystywane przez trzon aplikacji:

- *diagram* - definicje elementów schematu (bloki i połączenia)
- *eval* - ewaluacja instrukcji umieszczanych w blokach schematu, kompilacja do kodu wykonywalnego (JavaScript), typy zmiennych i błędów
- *event* - mechanizm do komunikacji między głównymi modułami oparty na zdarzeniach
- *ui* - komponenty interfejsu użytkownika (panele, formularze, okna dialogowe), ikony

Rysunek 5. przedstawia wymienione pakiety oraz zależności między nimi. Zostaną one szerzej omówione w kolejnych podrozdziałach.

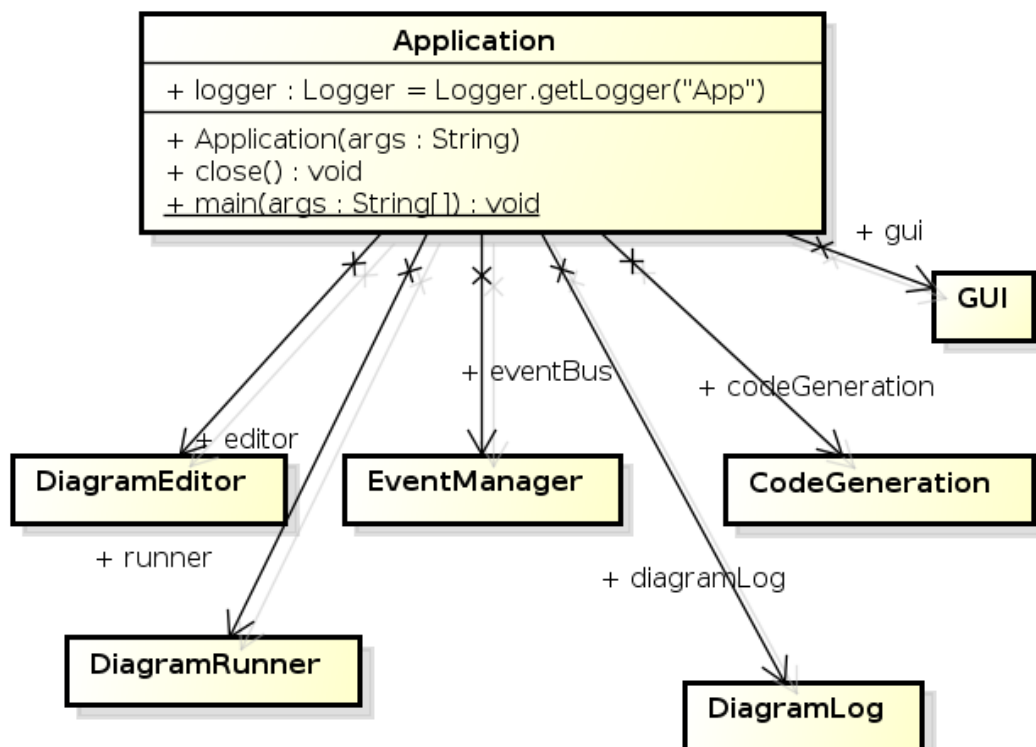
³ Diagramy klas finalnych (nie posiadających klas pochodnych w hierarchii dziedziczenia), dla zachowania przejrzystości, uwzględniają tylko metody i atrybuty publiczne. Diagramy klas bazowych zawierają ponadto metody abstrakcyjne oraz kluczowe atrybuty i metody chronione.



Rysunek 5. Modułarna struktura aplikacji
(Źródło: opracowanie własne)

5.2. Klasa *Application*

Klasa *Application* zawiera wejściową metodę *main* i reprezentuje uruchomioną instancję programu. Jej zadaniem jest inicjalizacja interfejsu użytkownika oraz głównych komponentów, a także zapewnienie poprawnego zamknięcia aplikacji.



Rysunek 6. Diagram klasy *Application*
(Źródło: opracowanie własne)

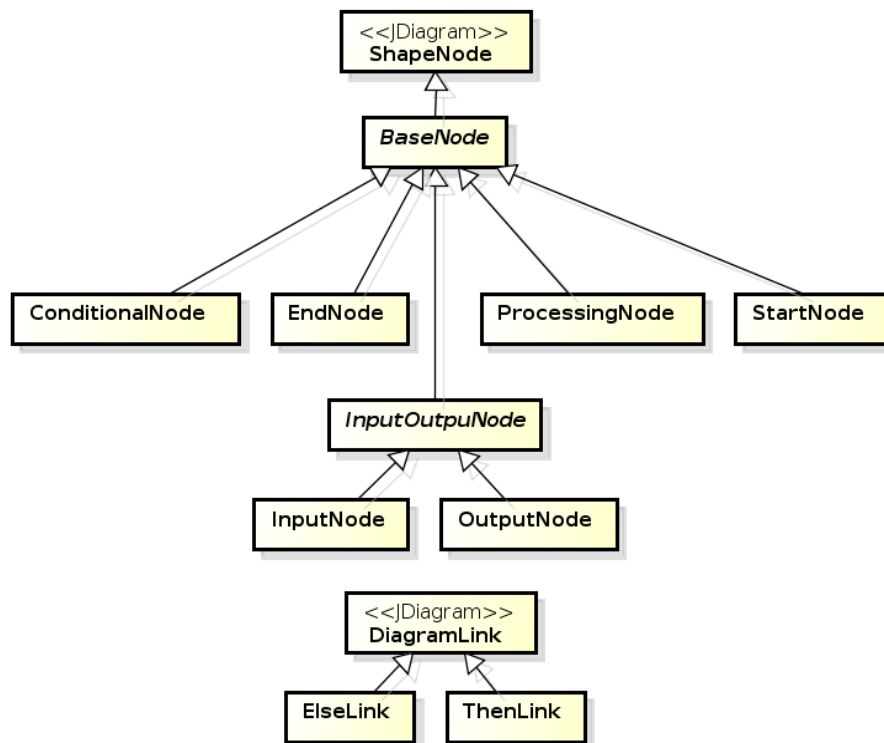
Najważniejsze metody i atrybuty

- *main(args : String[]) : void*
Rozpoczyna wykonywanie programu z tablicą parametrów wiersza poleceń oraz tworzy instancję klasy *Application*.
- *Application(args : String[])*
Konstruktor; przyjmuje argumenty *args* przekazane do *main()*.
- *close() : void*
Zamyka aplikację.

5.3. Pakiet *diagram*

Pakiet *diagram* grupuje klasy reprezentujące poszczególne elementy schematu - bloki i połączenia między nimi. Każda z klas definiuje typ elementu: jego wygląd, sposób działania i łączenia z innymi elementami, specyficzne dla niego pola danych oraz ich reprezentację w formacie XML (zapis/odczyt do/z pliku).

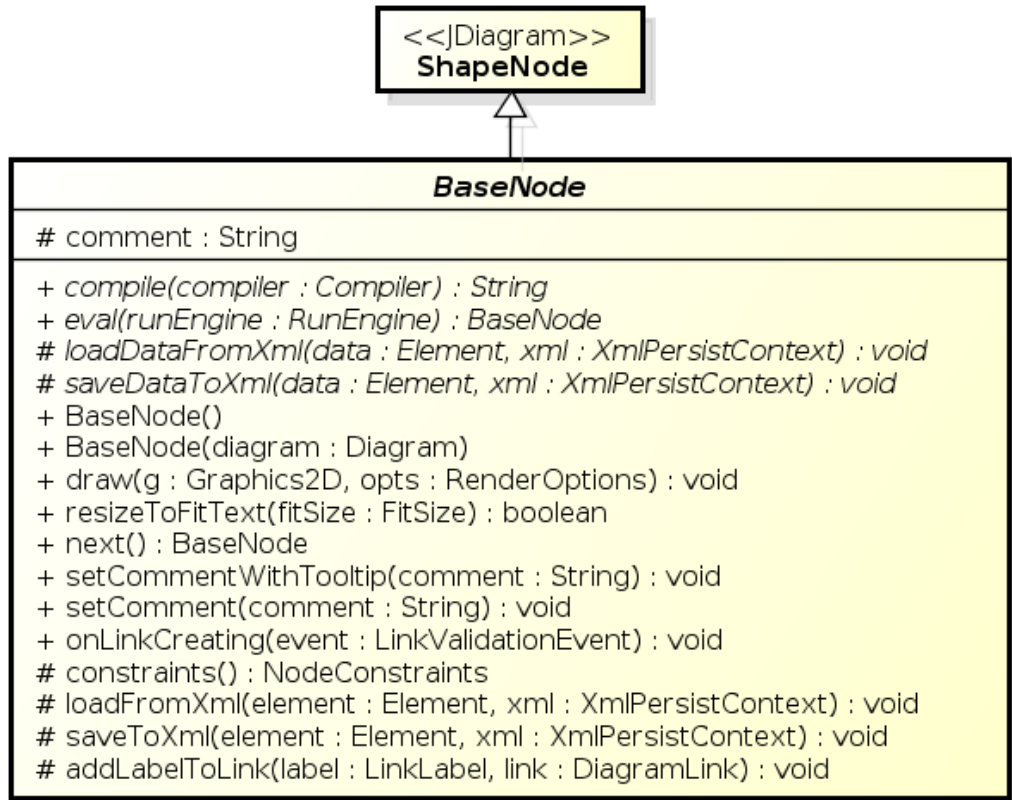
Struktura klas ma charakter hierarchiczny - wszystkie bloki dziedziczą z klasy abstrakcyjnej *BaseNode*, natomiast połączenia - z klasy *DiagramLink* (wchodzącej w skład biblioteki *JDiagram*).



Rysunek 7. Hierarchia klas reprezentujących elementy schematu
(Źródło: opracowanie własne)

5.3.1. Klasa *BaseNode*

Abstrakcyjna klasa *BaseNode* jest bazową klasą dla pozostałych klas reprezentujących bloki diagramu. Definiuje dane (np. komentarz) i funkcjonalności (np. ogólny mechanizm serializacji i deserializacji XML) wspólne dla całej hierarchii oraz specyfikuje metody abstrakcyjne wymagające implementacji w klasach pochodnych. Dzięki temu bloki posiadają jednolity interfejs (API), który może być w sposób polimorficzny wykorzystywany przez pozostałe komponenty aplikacji.



Rysunek 8. Diagram klasy *BaseNode*

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *comment : String*

Komentarz opisujący operacje wykonywane przez dany blok.

- *compile(compiler : Compiler) : String*

Kompiluje blok do postaci wykonywalnej przez ewaluator (*RunEngine*); zwraca kod JavaScript odpowiadający konkretnemu blokowi - konkretna implementacja wymagana w klasach pochodnych. Metoda ta pozwala na wykorzystanie polimorfi-

zmu [1] - kompilator zamieniający diagram na postać wykonywalną może generować kod dla każdego bloku bez potrzeby sprawdzania jego typu (klasy).

- *eval(runEngine : RunEngine) : BaseNode*

Ewaluuje (wykonuje) dany blok wykorzystując ewaluator podany jako argument; zwraca kolejny blok do wykonania w zależności od rezultatu; wymaga konkretnej implementacji w klasach pochodnych; metoda używana jedynie w testach jednostkowych.

- *saveToXml(element : Element, xml : XmlPersistContext) : void*

Serializuje blok do formatu XML wraz informacjami o wyglądzie, położeniu i połączeniach w obrębie schematu. Dodaje specjalny znacznik *Data*, w którym, poprzez nadpisanie metody *saveDataToXml* w klasach pochodnych, mogą być umieszczane dane specyficzne dla konkretnego typu bloku.

- *saveDataToXml(data : Element, xml : XmlPersistContext) : void*

Metoda abstrakcyjna pozwalająca na implementację serializacji do XML danych specyficznych dla konkretnego typu bloku (bez potrzeby kaskadowego wywoływania metody *saveToXml* z klasy bazowej).

- *loadFromXml(element : Element, xml : XmlPersistContext) : void*

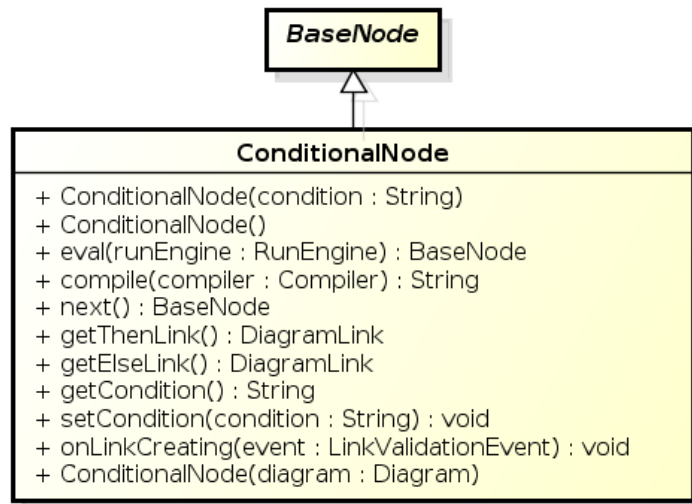
Metoda komplementarna do *saveToXml*. Deserializuje blok zapisany w formacie XML. Inicjuje, za pomocą wywołania *loadDataFromXml*, atrybuty specyficzne dla danego typu bloku.

- *loadDataFromXml(data : Element, xml : XmlPersistContext) : void*

Abstrakcyjna metoda komplementarna do *saveDataToXml*. Pozwala na implementację deserializacji danych XML dla konkretnego bloku (bez konieczności kaskadowego wywoływania metody *loadFromXml* z klasy bazowej).

5.3.2. Klasa *ConditionalNode*

Klasa *ConditionalNode* reprezentuje blok decyzyjny pozwalający uzależnić ścieżkę wykonania od zadanego wyrażenia warunkowego.



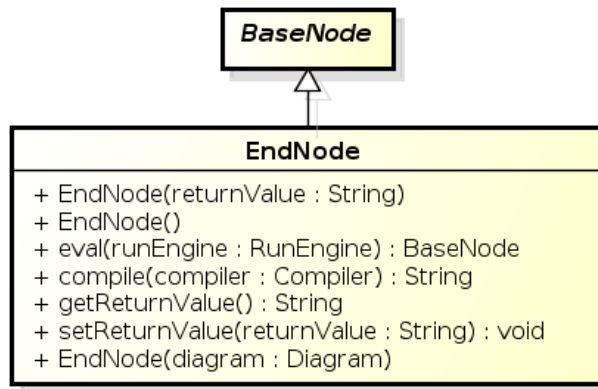
Rysunek 9. Diagram klasy *ConditionalNode*
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *condition : String*
Wyrażenie warunkowe.
- *compile(compiler : Compiler) : String*
Implementacja metody z klasy bazowej; zwraca kod postaci *if (condition) blok-po-ThenLink else blok-po-ElseLink*.
- *eval(runEngine : RunEngine) : BaseNode*
Implementacja metody z klasy bazowej; jeśli dane wyrażenie (*condition*) ewaluuje się jako prawdziwe, zwracany jest jako następny blok przyłączony linkiem *Then-Link* - w przeciwnym przypadku zwracany jest blok przyłączony poprzez *ElseLink*.

5.3.3. Klasa *EndNode*

Klasa *EndNode* reprezentuje końcowy blok schematu, oznaczający zakończenie ścieżki wykonania lub powrót do miejsca wywołania (w przypadku funkcji).



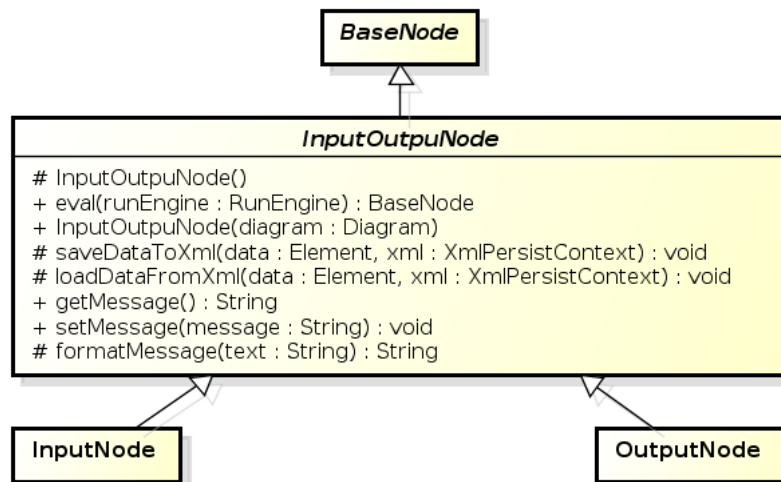
Rysunek 10. Diagram klasy *EndNode*
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *returnValue : String*
Nazwa zmiennej zwracanej jako rezultat działania.
- *compile(compiler : Compiler) : String*
Implementacja metody z klasy bazowej; zwraca instrukcję *return returnValue*, gdzie *returnValue* jest atrybutem klasy przechowującym zwracaną wartość.

5.3.4. Klasa *InputOutputNode*

Klasa *InputOutputNode* jest abstrakcyjną klasą bazową dla bloków wejścia (*InputNode*) i wyjścia (*OutputNode*). Definiuje atrybuty (*message* - wiadomość wyświetlana zarówno przy wprowadzaniu jak i wyprowadzaniu danych) oraz operacje (zapis do/odczyt z XML, interpolacja zmiennych) wspólne dla obydwu bloków.



Rysunek 11. Diagram klasy *InputOutputNode*
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *message : String*

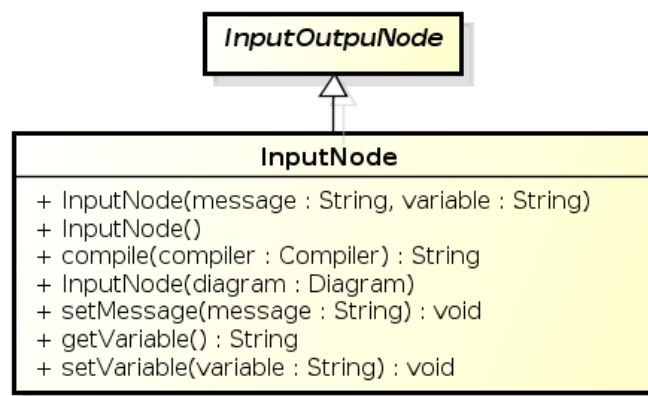
Komunikat wyświetlany użytkownikowi przy wprowadzaniu/wyprowadzaniu danych.

- *formatMessage(text : String) : String*

Formatuje komunikat podany jako argument przy pomocy znaczników HTML - zwracany tekst oznaczony jest kursywą, zawarte w nim odwołania do zmiennych (postaci *\$nazwa_zmiennej*) oznaczane są pogrubieniem (interpolacja).

5.3.5. Klasa *InputNode*

Klasa *InputNode* reprezentuje blok wejścia, poprzez który użytkownik może w czasie wykonania wprowadzić dane tekstowe lub liczbowe. Wprowadzona wartość przypisywana jest do zmiennej o podanej nazwie.



Rysunek 12. Diagram klasy *InputNode*

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *variable : String*

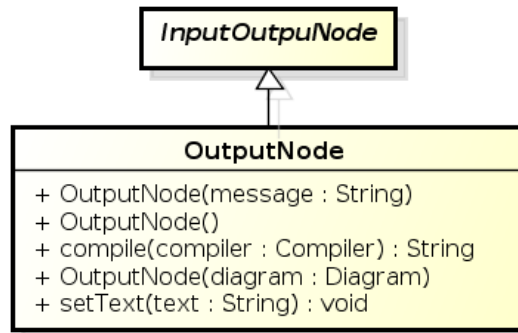
Zmienna, do której ma być przypisana wprowadzona wartość.

- *compile(compiler : Compiler) : String*

Implementacja abstrakcyjnej metody klasy *BaseNode*. Zwracany kod powoduje wyświetlenie w czasie wykonania okna dialogowego (*javax.swing.JOptionPane.showInputDialog*) zawierającego podany komunikat (*message*) oraz pole tekstowe do wprowadzenia wartości dla zmiennej *variable*.

5.3.6. Klasa *OutputNode*

Klasa *OutputNode* reprezentuje blok wyjściowy (wyprowadzenia danych), pozwalający na wyświetlanie komunikatów i wyników w czasie wykonania.



Rysunek 13. Diagram klasy *OutputNode*

(Źródło: opracowanie własne)

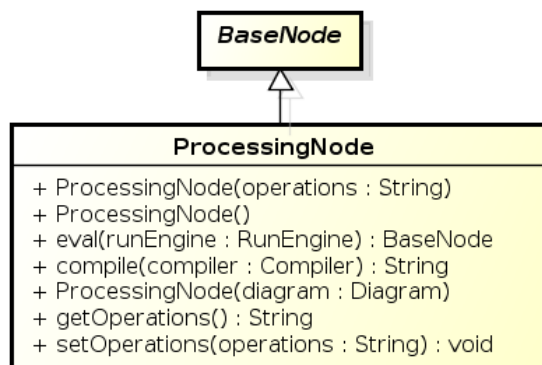
Najważniejsze metody i atrybuty

- *compile(compiler : Compiler) : String*

Implementacja abstrakcyjnej metody klasy *BaseNode*. Generuje kod wyświetlający w czasie wykonania okno dialogowe (*javax.swing.JOptionPane.showMessageDialog*) zawierające podany komunikat (*message*). Odwołania do zmiennych (postaci *\$nazwa_zmiennej*) są interpolowane - zamieniane na ich wartości (za pomocą metody *eval.Compiler.interpolate*).

5.3.7. Klasa *ProcessingNode*

Klasa *ProcessingNode* reprezentuje blok obliczeń (przetwarzania).



Rysunek 14. Diagram klasy *ProcessingNode*

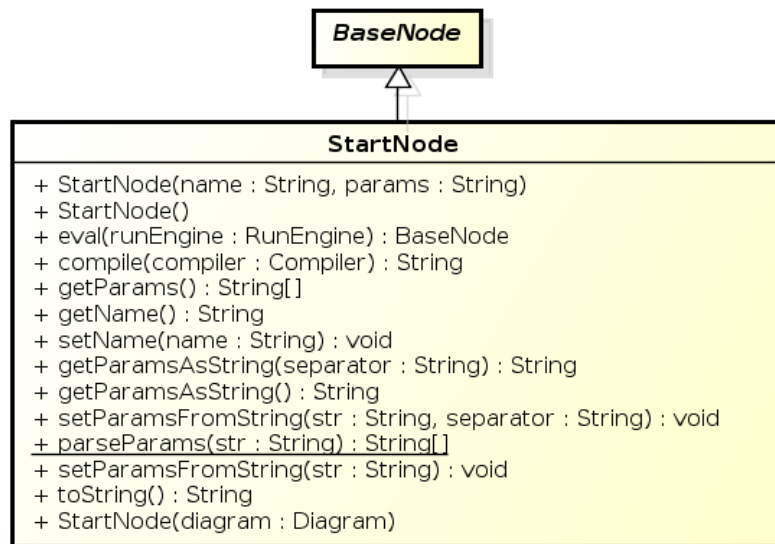
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *operations : String*
Operacje do wykonania w ramach bloku (instrukcje rozdzielone przecinkami).
- *compile(compiler : Compiler) : String*
Implementacja abstrakcyjnej metody klasy *BaseNode*. Zwraca kod podany w atrybucie *operations*.

5.3.8. Klasa *StartNode*

Klasa *StartNode* reprezentuje blok startowy, od którego zaczyna się wykonywanie następującej po nim, połączonej sekwencji bloków (zakończonej blokiem *EndNode*). Dany schemat może zawierać kilka bloków tego typu – każdy jest identyfikowany przez swoją nazwę i listę parametrów formalnych (w czasie wykonania sekwencja bloków między *StartNode* i *EndNode* traktowana jest jak funkcja).



Rysunek 15. Diagram klasy *StartNode*

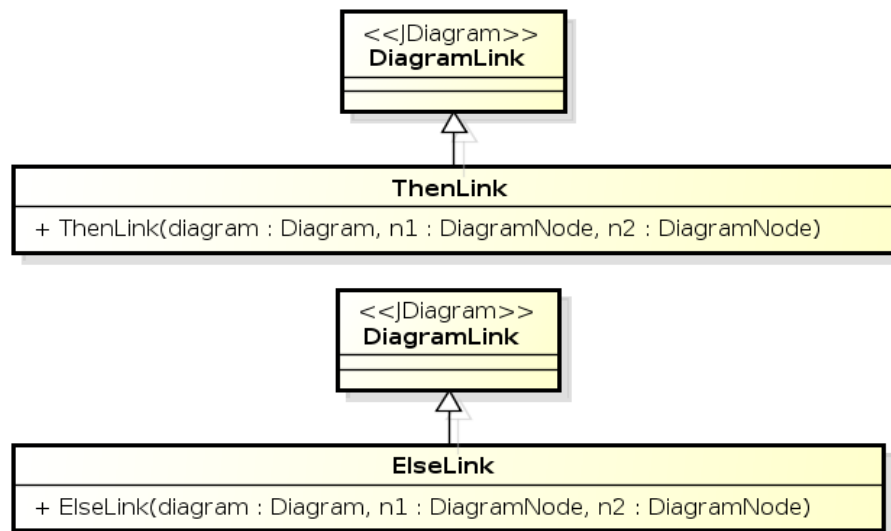
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *name : String*
Nazwa bloku startowego (identyfikuje go w czasie wykonania).
- *params : String[]*
Tablica parametrów formalnych, które w czasie wykonania są zamieniane na odpowiednie parametry aktualne (argumenty wywołania).

5.3.9. Klasy *ThenLink* i *ElseLink*

Klasy *ThenLink* i *ElseLink* reprezentują połączenia wychodzące z bloku *ConditionalNode*. Wyróżnienie w tym przypadku dwu odrębnych klas (wszystkie pozostałe połączenia reprezentowane są przez klasę *DiagramLink* z biblioteki *JDiagram*) wynika ze specyfiki bloku warunkowego – jest to jedyny typ bloku, który posiada dwa wyjścia. Jeśli podany warunek jest spełniony, zostaje wybrane wyjście „TAK” (*ThenLink*), w przeciwnym razie wybierane jest połączenie „NIE” (*ElseLink*). W przypadku braku którejkolwiek z połączeń, wykonanie zostaje wstrzymane.



Rysunek 16. Diagram klas *ThenLink* i *ElseLink*

(Źródło: opracowanie własne)

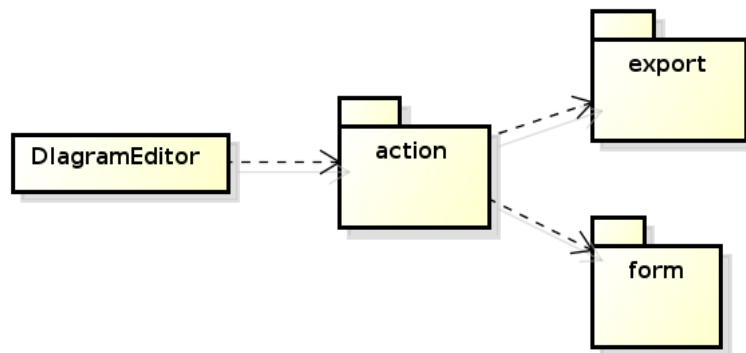
Najważniejsze metody i atrybuty

- *ThenLink(diagram : Diagram, n1 : DiagramNode, n2 : DiagramNode)*
- *ElseLink(diagram : Diagram, n1 : DiagramNode, n2 : DiagramNode)*

Konstruktory. Przyjmują jako argument instancję schematu (*diagram*), w ramach którego ma zostać utworzone połączenie, oraz bloki do połączenia: *n1* – blok źródłowy, *n2* – blok docelowy.

5.4. Pakiet *editor*

Pakiet ten zawiera komponenty składające się na edytor schematów, realizujące takie funkcjonalności jak: zarządzanie schematami, operacje na blokach i połączeniach (dodawanie, usuwanie, edycja), zapisywanie i wczytywanie plików XML oraz eksport do innych formatów (GIF/JPEG/PNG, PDF), obsługa historii operacji (cofanie, powtarzanie).



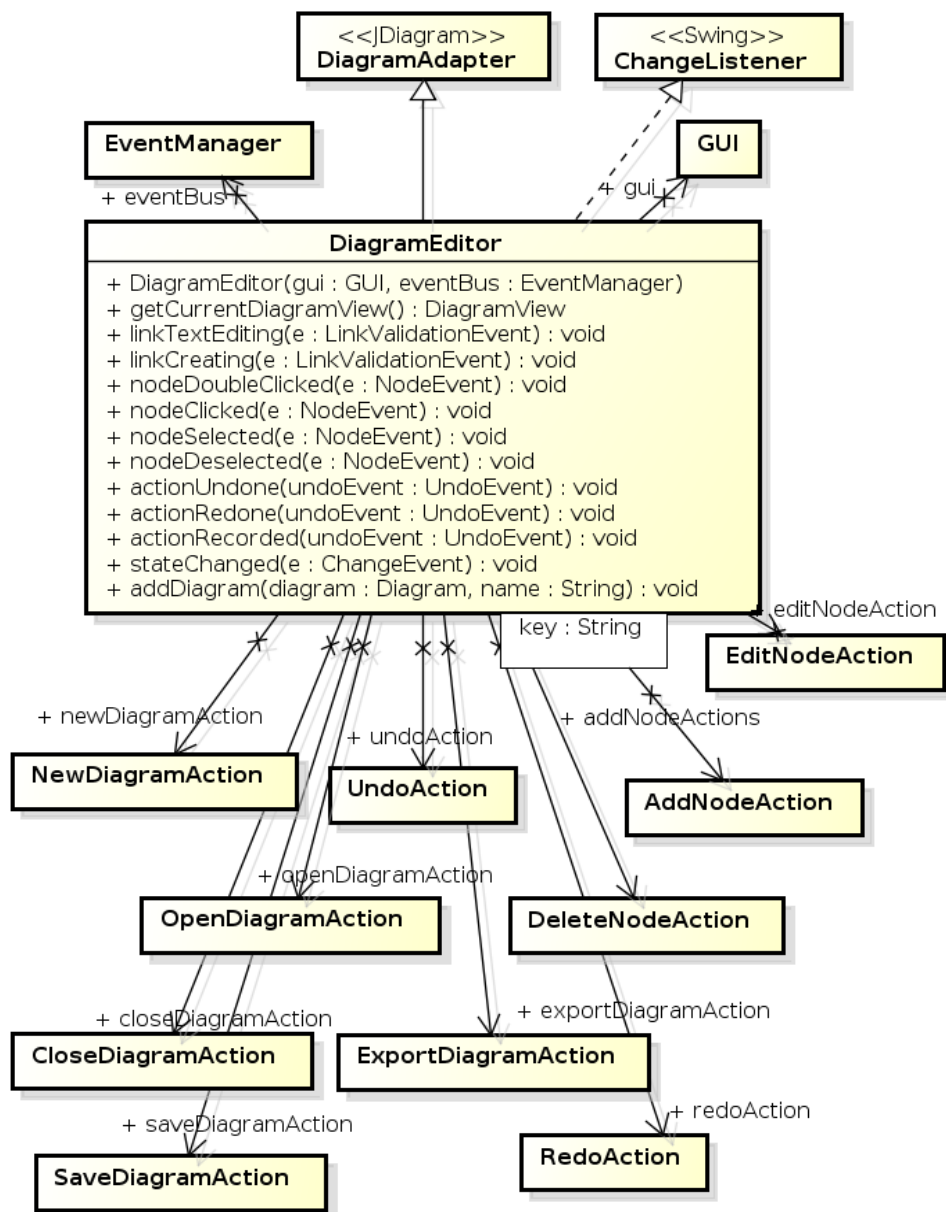
Rysunek 17. Wysokopoziomowa struktura pakietu editor
(Źródło: opracowanie własne)

DiagramEditor jest główną klasą integrującą pozostałe części składowe pakietu w jeden komponent gotowy do osadzenia w aplikacji. Funkcjonalności tego komponentu zostały zdefiniowane poprzez oddzielne klasy (np. *DeleteNodeAction* – usunięcie bloku, *UndoAction* – cofnięcie operacji) zgrupowane w pakiecie *action*. Pozostałe pakiety - *form* i *export* - zawierają komponenty wykorzystywane przy – odpowiednio: edycji bloków (formularze dla poszczególnych typów) i eksporcie schematu (eksportery dla poszczególnych formatów).

5.4.1. Klasa *DiagramEditor*

Jak już wspomniano, klasa *DiagramEditor* reprezentuje komponent edytora schematów blokowych. Jej zadaniem jest przypisanie swoich akcji do odpowiednich elementów interfejsu użytkownika i zarządzanie ich aktywnością, rejestracja klas reprezentujących składowe schematu (wymagana przez bibliotekę *JDiagram*), oraz obsługa zdarzeń związanych z edycją.

DiagramEditor dziedziczy z klasy *DiagramAdapter* biblioteki *JDiagram* nadpisując niektóre z metod odpowiadających za obsługę zdarzeń wynikających z działań użytkownika (zastosowanie wzorca projektowego *adapter* [10]). Implementuje ponadto interfejs *ChangeListener* (Swing), aby wykryć zmianę aktywnego obszaru roboczego.



Rysunek 18. Diagram klasy DiagramEditor

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty

- *gui* : GUI
- *eventBus* : EventManager

Referencje do wspólnych dla całej aplikacji komponentów obsługujących interfejs graficzny oraz zdarzenia (Rozdział 5.8.1).

- *newDiagramAction* : NewDiagramAction
- *openDiagramAction* : OpenDiagramAction
- *closeDiagramAction* : CloseDiagramAction

- *saveDiagramAction : SaveDiagramAction*

- *exportDiagramAction : ExportDiagramAction*

Instancje zdarzeń dotyczące schematu jako całości, przypisywane do odpowiednich elementów interfejsu (menu *Plik* i paska narzędziowego).

- *addNodeActions : Map<String, AddNodeAction>*

Kolekcja akcji modelujących dodanie bloku do schematu, po jednej dla każdego typu. Na podstawie tej kolekcji automatycznie tworzone jest lewe menu z paletą dostępnych bloków oraz menu *Edycja/Wstaw*.

- *editNodeAction : EditNodeAction*

- *deleteNodeAction : DeleteNodeAction*

Instancje akcji reprezentujących operacje na pojedynczym bloku – edycja i usunięcie, przypisywane do menu kontekstowego oraz menu *Edycja*.

- *undoAction : UndoAction*

- *redoAction : RedoAction*

Instancje akcji umożliwiających odpowiednio: cofnięcie i powtórzenie ostatnio wykonanej operacji.

- *getCurrentDiagramView() : DiagramView*

Zwraca aktywny obszar roboczy (aktualnie widoczna zakładka). Każdy schemat umieszczany jest w ramach obszaru roboczego reprezentowanego przez klasę *DiagramView* (*JDDiagram*) będącą jednocześnie komponentem Swing. Z kolei każdy obszar roboczy jest zagnieżdżony w osobnym panelu *JTabbedPane*, aby umożliwić pracę nad kilkoma schematami jednocześnie.

- *addDiagram(diagram : Diagram, name : String) : void*

Dodaje zakładkę (o podanej nazwie *name*) z obszarem roboczym dla schematu *diagram* - nowego lub wczytanego z pliku; metoda pomocnicza wykorzystywana przez akcje *NewDiagramAction* oraz *OpenDiagramAction*.

- *nodeDoubleClicked(e : NodeEvent) : void*

- *nodeClicked(e : NodeEvent) : void*

Nadpisane metody z klasy *DiagramAdapter* pozwalające na obsługę zdarzeń - odpowiednio: podwójnego kliknięcia na bloku (otwarcie formularza edycji) i kliknięcia prawym przyciskiem myszy (wyświetlenie menu kontekstowego z operacjami na danym bloku).

- *nodeSelected(e : NodeEvent) : void*

- *nodeDeselected(e : NodeEvent) : void*

Metody klasy *DiagramAdapter* wywoływane przy zaznaczeniu/odznaczeniu danego bloku; napisane w celu aktywacji/dezaktywacji akcji dot. danego bloku (edycja, usunięcie).

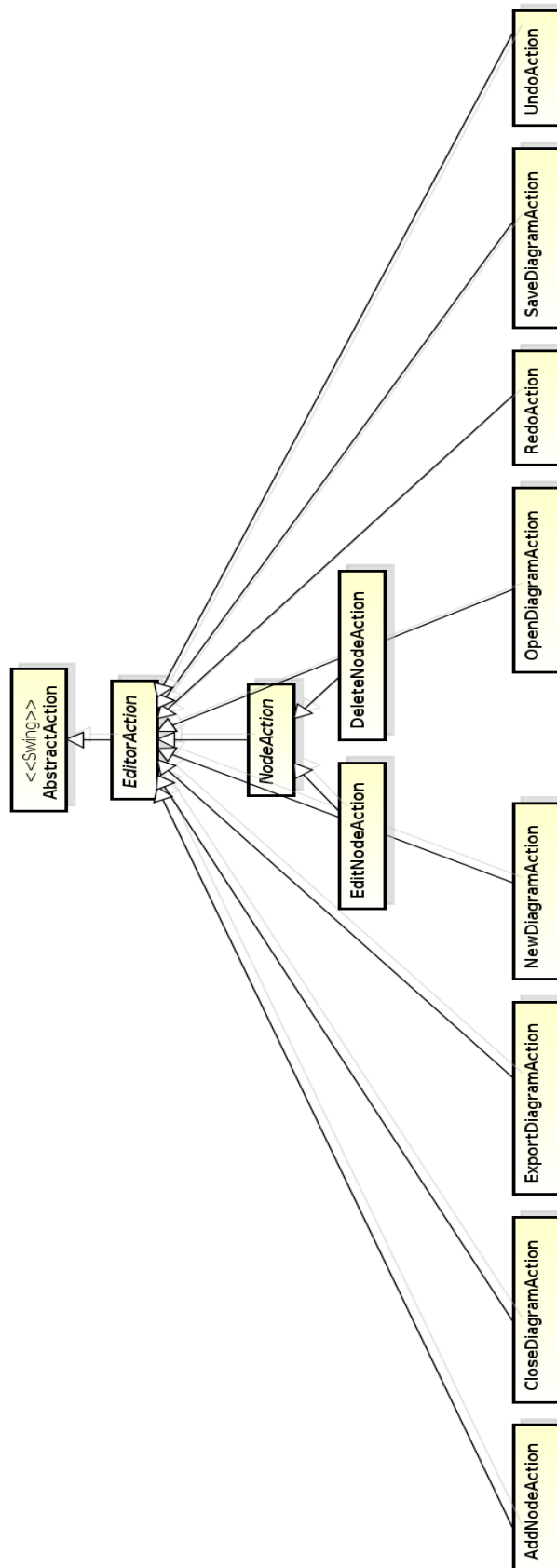
- *actionUndone(undoEvent : UndoEvent) : void*
- *actionRedone(undoEvent : UndoEvent) : void*
- *actionRecorded(undoEvent : UndoEvent) : void*

Metody klasy *DiagramAdapter* związane z obsługą zdarzeń dot. historii operacji, wywoływane odpowiednio po: cofnięciu ostatniej operacji, ponownym wykonaniu ostatniej operacji, zapisaniu operacji w historii wykonania; napisane w celu aktywacji/dezaktywacji akcji *UndoAction* i *RedoAction*.

5.4.2. Pakiet *editor.action*

Pakiet *action* zawiera klasy reprezentujące akcje edytora, które odpowiadają jego poszczególnym funkcjonalnościom. Zorganizowano je na zasadzie hierarchicznej – klasą bazową jest dla nich klasa abstrakcyjna *EditorAction* dziedzicząca z *AbstractAction* biblioteki Swing [3]. Dla akcji operujących na pojedynczych blokach została wydzielona dodatkowa klasa nadrzędna – *NodeAction*.

Enkapsulacja funkcjonalności w osobnych klasach pozwala na przypisanie tej samej akcji (za pomocą metody *setAction*) do wielu komponentów interfejsu użytkownika, umożliwiając łatwe kontrolowanie ich aktywności, definiowanie nazewnictwa oraz ikon (np. aby dezaktywować wszystkie komponenty powiązane z daną akcją, wystarczy wywołać jej metodę *setEnabled(false)*).



Rysunek 19. Hierarchia klas pakietu editor.action
(Źródło: opracowanie własne)

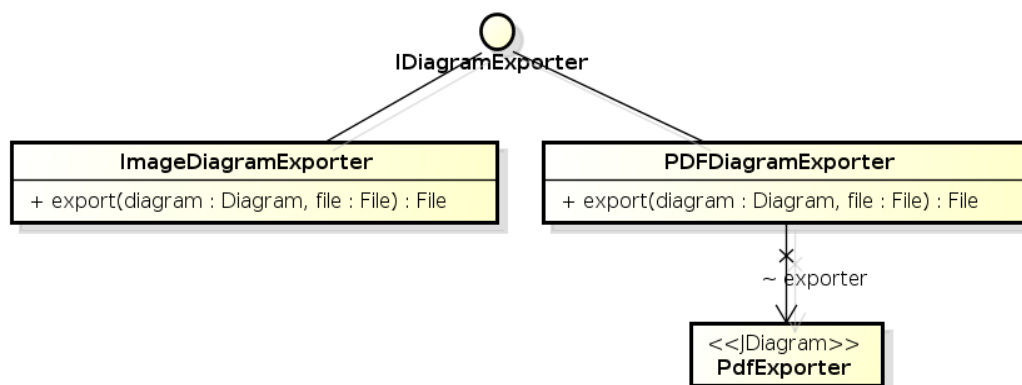
Akcje posiadają stosunkowo prosty interfejs – poza konstruktorem, implementują jedynie metodę *actionPerformed* definiującą faktyczne operacje (zgodnie z wzorcem projektowym *polecenie - command pattern* [3, 10]). Poniżej znajduje się ich krótki opis:

- *AddNodeAction* – dodanie nowego bloku do schematu, akcja parametryzowana typem (klasą) bloku;
- *CloseDiagramAction* – zamknięcie aktywnego obszaru roboczego;
- *ExportDiagramAction* – eksport aktywnego schematu do wybranego formatu za pomocą jednego z dostępnych eksporterów;
- *NewDiagramAction* – dodanie nowego schematu do edytora;
- *OpenDiagramAction* – wczytanie schematu z pliku;
- *SaveDiagramAction* – zapisanie aktywnego schematu do pliku;
- *UndoAction* – cofnięcie ostatnio wykonanej operacji;
- *RedoAction* – powtórzenie ostatnio wykonanej operacji;
- *EditNodeAction* – edycja zaznaczonego bloku; otwiera okno dialogowe z odpowiednim formularzem (Rozdział 5.4.4);
- *DeleteNodeAction* – usunięcie zaznaczonego bloku ze schematu;

5.4.3. Pakiet *editor.export*

Pakiet ten zawiera komponenty pomocnicze wykorzystywane przez akcję *ExportDiagramAction*: eksportery do różnych formatów oraz związane z nimi filtry wyboru plików.

Eksportery implementują wspólny interfejs *IDiagramExporter* definiujący jedną metodę – *export*, której zadaniem jest wykonanie konwersji diagramu do wymaganego formatu oraz zapisanie go w podanym pliku (zastosowano tu wzorec projektowy *strategia* [10]).

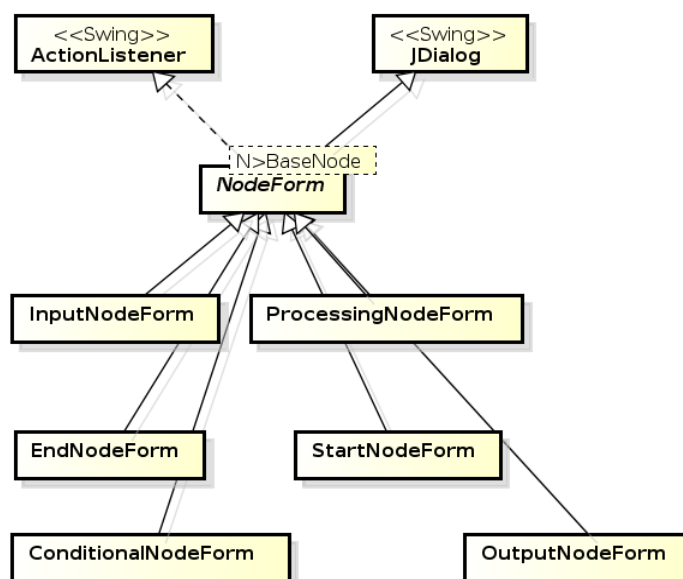


Rysunek 20. Diagram klas reprezentujących eksportery
(Źródło: opracowanie własne)

Klasa *ImageDiagramExporter* umożliwia zapisanie schematu jako plik graficzny w formacie PNG, JPEG lub GIF (format wybierany jest na podstawie rozszerzenia pliku). Drugim dostępnym eksporterem jest *PDFDiagramExporter* udostępniający konwersję diagramu i zapis jako dokument PDF. Wykorzystuje w tym celu klasę *PdfExporter* z biblioteki JDiagram. Obydwa eksportery starają się optymalizować rozmiar wynikowego pliku poprzez usuwanie zbędnych marginesów przed zapisem.

5.4.4. Pakiet *editor.form*

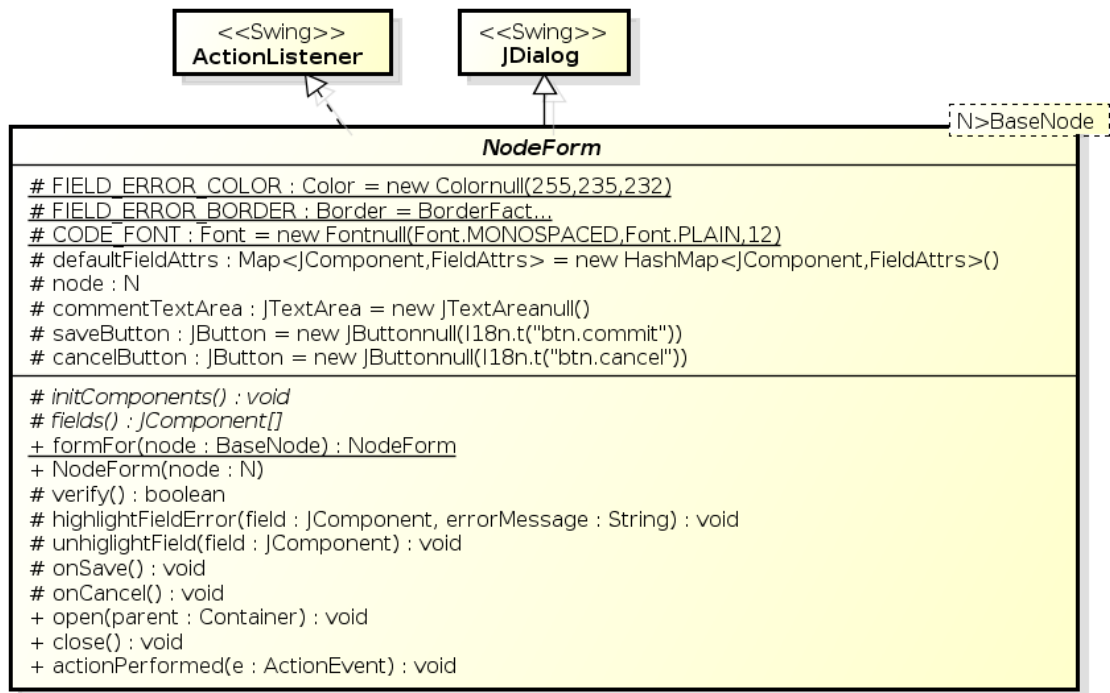
W pakiecie tym zgrupowano kod generujący formularze edycji bloków schematu wykorzystywane przez akcję *EditNodeAction*. Każdy typ bloku posiada swój formularz z polami odpowiadającymi poszczególnym atrybutom oraz regułami ich walidacji.



Rysunek 21. Diagram klas reprezentujących formularze edycji bloków
(Źródło: opracowanie własne)

5.4.4.1. Klasa *NodeForm*

Klasy reprezentujące formularze tworzą hierarchię dziedziczenia ze wspólną klasą bazową *NodeForm* zdefiniowaną jako typ generyczny parametryzowany typem bloku. Implementuje ona podstawowe mechanizmy dot. wyświetlania okna dialogowego i obsługi zdarzeń oraz udostępnia metody pozwalające na dostosowanie formularza do potrzeb danego bloku (*verify()*, *initComponents()*, *onSave()*, *onCancel()*).



Rysunek 22. Diagram klasy *NodeForm*

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

- *node : N*
Instancja bloku, którego dotyczy formularz; *N* jest typem parametryzującym, dziedziczącym z klasy *BaseNode*.
- *commentTextArea : JTextArea*
Pole tekstowe dla atrybutu komentarz (wspólne dla wszystkich formularzy).
- *initComponents() : void*
- *fields() : JComponent[]*
Metody abstrakcyjne pozwalające na zdefiniowanie pól wymaganych przez dany blok.
- *verify() : boolean*

Metoda pozwalająca na zdefiniowanie reguł walidacji dla pól formularza; jej implementacje w klasach pochodnych wykorzystują w tym celu komponent *eval.SyntaxValidator*.

- *onSave() : void*
- *onCancel() : void*

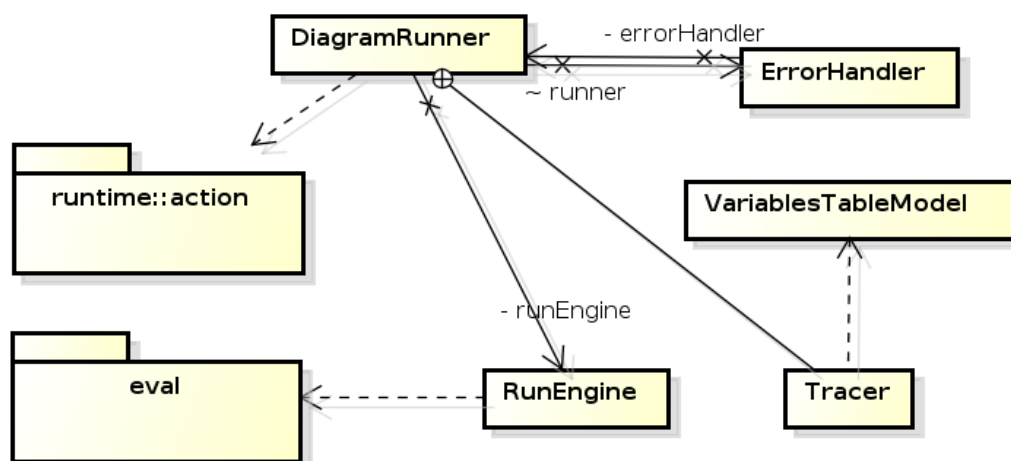
Metody wywoływane przed zamknięciem okna dialogowego z formularzem, po kliknięciu przycisków – odpowiednio: *OK* i *Anuluj*

- *formFor(node : BaseNode) : NodeForm*

Statyczna metoda wykorzystywana przez akcję *EditNodeAction* do stworzenia odpowiedniego typu formularza dla podanego w argumencie bloku; zwraca jedną z klas pochodnych *NodeForm* (metoda wytwórcza, *factory method* [10, 2]).

5.5. Pakiet *runtime*

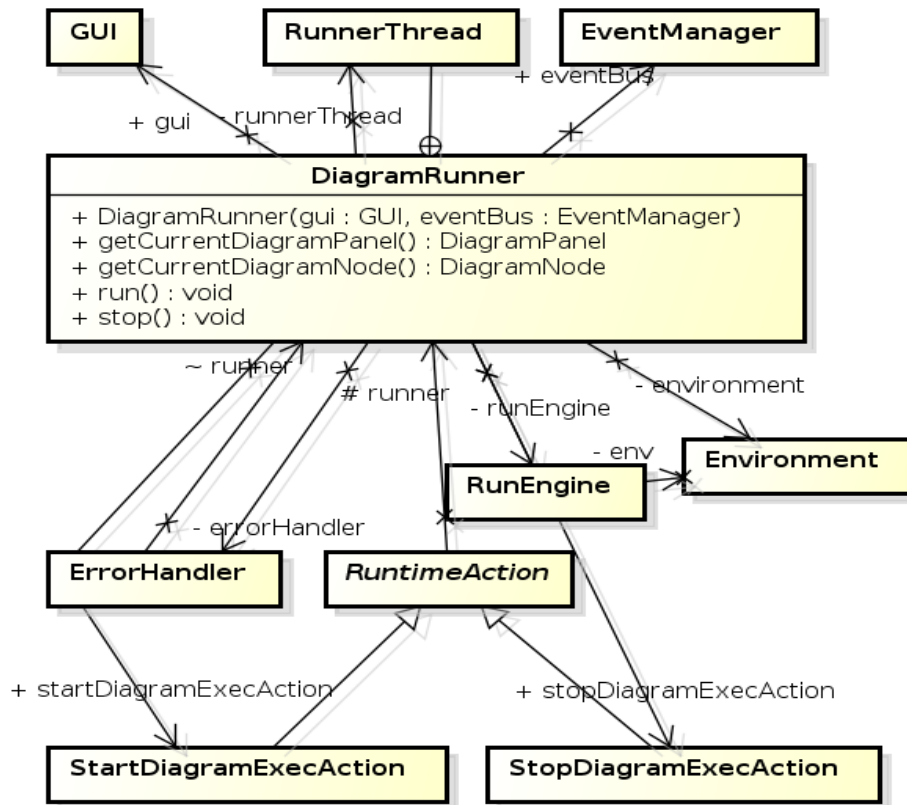
Pakiet *runtime* zawiera kod wykorzystywany przy symulowaniu działania (uruchamianiu) schematów. Zorganizowano go podobnie jak w pakiecie *editor* – z jednym komponentem nadrzędnym (*DiagramRunner*) integrującym pozostałe i osadzonym w głównej klasie aplikacji. Ponieważ symulowanie działania schematu wymaga interpretacji i ewaluacji instrukcji zawartych w blokach, moduł ten jest w znacznym stopniu powiązany z realizującym te funkcje pakietem *eval* (Rozdział 5.7).



Rysunek 23. Diagram klas pakietu *runtime*
(Źródło: opracowanie własne)

5.5.1. Klasa *DiagramRunner*

Zadaniem klasy *DiagramRunner* jest przypisanie akcji z pakietu *action* do odpowiednich elementów interfejsu użytkownika i zarządzanie ich aktywnością oraz koordynacja pracy pozostałych komponentów odpowiedzialnych za symulowanie działania schematów.



Rysunek 24. Diagram klasy *DiagramRunner*

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

- *runnerThread* : *RunnerThread*

Wątek, w ramach którego uruchamiana jest symulacja. Wykonywanie schematu odbywa się w osobnym wątku, aby nie blokować głównego programu oraz interfejsu użytkownika [3] – dzięki temu można przerwać symulację w dowolnym momencie, bez potrzeby oczekiwania na jej zakończenie.

- *runEngine* : *RunEngine*

Instancja komponentu, którego zadaniem jest ewaluacja i wykonanie poszczególnych bloków schematu.

- *environment* : *Environment*

Instancja środowiska (kontekstu) wykonawczego, zawierającego wszystkie zdefiniowane w czasie wykonania zmienne i funkcje.

- *errorHandler : ErrorHandler*

Obiekt odpowiedzialny za przechwycenie i obsługę wyjątków zgłaszanych w czasie wykonania schematu przez wątek *runnerThread*.

- *startDiagramExecAction : StartDiagramExecAction*

- *stopDiagramExecAction : StopDiagramExecAction*

Instancje akcji odpowiednio – rozpoczynających i kończących symulację.

- *run() : void*

Metoda wywoływana przez *startDiagramExecAction* uruchamiająca aktywny schemat blokowy; inicjuje nowe środowisko wykonawcze (*environment*) i uruchamia nowy wątek *RunnerThread*.

- *stop() : void*

Zatrzymuje wątek symulacji (*runnerThread*).

- *getCurrentDiagramNode() : DiagramNode*

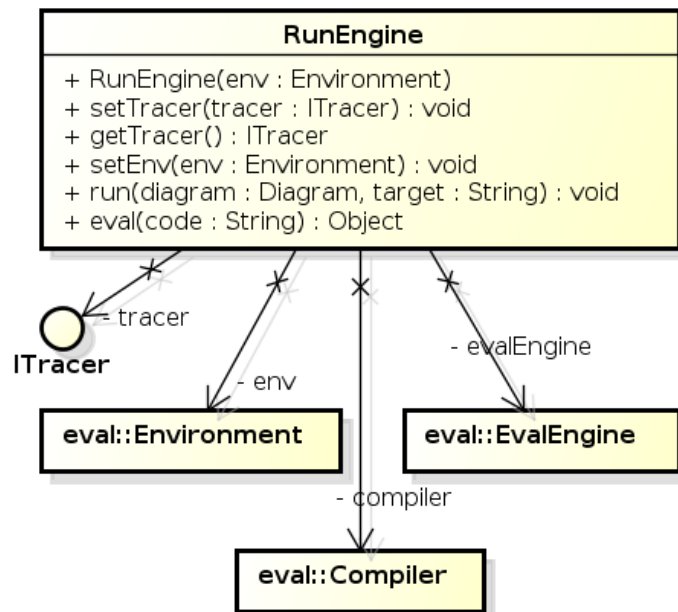
Zwraca aktualnie wykonywany blok schematu.

- *getCurrentDiagramPanel() : DiagramPanel*

Zwraca aktywny obszar roboczy.

5.5.2. Klasa *RunEngine*

RunEngine jest kluczowym komponentem odpowiedzialnym za ewaluację i wykonanie poszczególnych bloków schematu - wykorzystuje w tym celu klasy pakietu *eval*. Umożliwia dynamiczne definiowanie kodu mającego się wykonać przed i po ewaluacji każdego bloku (za pomocą interfejsu *ITracer*), co pozwala na śledzenie zmiennych i wyróżnianie aktualnego bloku.



Rysunek 25. Diagram klasy *RunEngine*
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

- *evalEngine : EvalEngine*

Ewaluator kodu (JavaScript) wygenerowanego przez *Compiler* (część pakietu *eval*).

- *compiler : Compiler*

Komponent przekształcający schemat do postaci wykonywalnej przez ewaluator (część pakietu *eval*).

- *env : Environment*

Środowisko (kontekst), w ramach którego ma zostać uruchomiony diagram.

- *tracer : ITracer*

Implementacja interfejsu umożliwiającego dynamiczne „wstrzyknięcie” kodu przed i po wykonaniu aktualnego bloku. Jego instancja jest dodawana do aktualnego środowiska (*env*) i w ten sposób dostępna dla kompilatora (*compiler*), który przed i po kodzie odpowiadającym danemu blokowi generuje wywołania metod interfejsu *ITracer*, odpowiednio – *before()* i *after()* - przekazując im numer bloku. Implementacja wykorzystana w *DiagramRunner* (zdefiniowana jako wewnętrzna klasa *Tracer*) pozwala na wprowadzenie opóźnienia symulacji, wyróżnienie aktualnie wykonywanego bloku oraz dynamiczną aktualizację tabeli zmiennych (*VariablesTableModel*).

- *run(diagram : Diagram, target : String) : void*

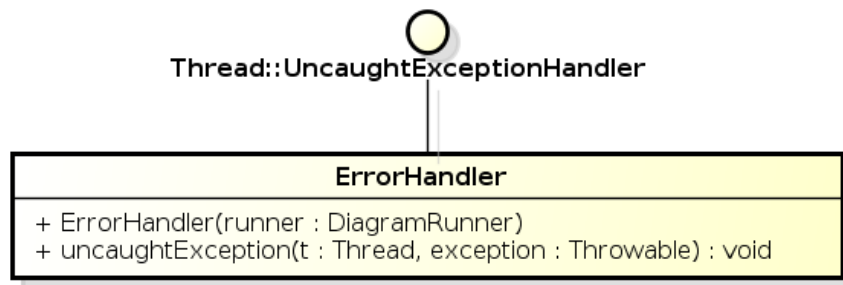
Metoda wykonująca podany schemat (*diagram*) zaczynając od bloku startowego o podanej nazwie i argumentach (*target*). Jeśli nie znaleziono pasującego bloku startowego, zgłaszany jest błąd referencji (*ReferenceError*); w przeciwnym wypadku, kompilator (*compiler*) generuje kod JavaScript dla schematu, po czym dokonuje się ewaluacji danego bloku startowego (*target*), co rozpoczyna jego wykonywanie.

- *eval(code : String) : Object*

Wykonuje kod (JavaScript) podany w argumencie *code* wykorzystując komponent *evalEngine*.

5.5.3. Klasa *ErrorHandler*

Zadaniem klasy *ErrorHandler* jest przechwytywanie błędów i wyjątków czasu wykonania wyrzucanych przez wątek *RunnerThread*. W tym celu implementuje ona standardowy interfejs *Thread, UncaughtExceptionHandler*.



Rysunek 26. Diagram klasy *ErrorHandler*

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

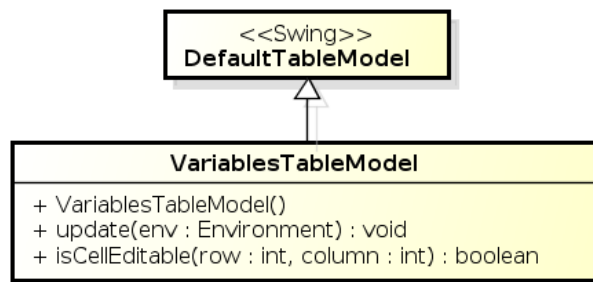
- *uncaughtException(t : Thread, exception : Throwable) : void*

Metoda wymagana przez interfejs *Thread, UncaughtExceptionHandler* [1]; obsługuje błędy czasu wykonania: zatrzymuje symulację schematu, oznacza kolorem ostatnio wykonany blok oraz wyświetla okno dialogowe z odpowiednim komunikatem.

5.5.4. Klasa *VariablesTableModel*

Klasa ta implementuje model dla komponentu *JTable*, aktualizujący podgląd zmiennych w czasie symulacji schematu (zgodnie z wzorcem *Model-View-Controller* [10]). Aktualizacja odbywa się za pośrednictwem wspomnianego interfejsu *ITracer*, po każ-

dym wykonaniu bloku. Źródłem informacji o kontekście, nazwach i aktualnych wartościach zmiennych jest instancja środowiska wykonawczego (*environment*).



Rysunek 27. Diagram klasy *VariablesTableModel*
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

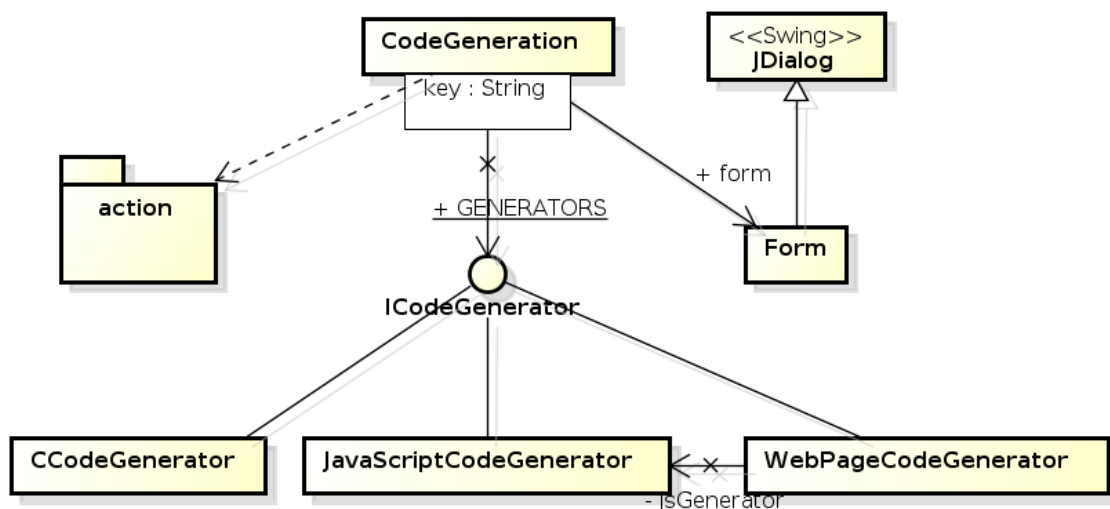
- *update(env : Environment) : void*

Metoda wywoływana po wykonaniu każdego bloku, odświeżająca podgląd zmiennych na podstawie aktualnej zawartości środowiska *env*; wyświetla kontekst zmiennej (nazwę najbliższego bloku startowego), jej nazwę oraz wartość.

5.6. Pakiet *codegen*

Pakiet *codegen* skupia komponenty odpowiedzialne za generowanie kodu źródłowego na podstawie danego schematu. Podobnie jak w przypadku omówionych już modułów *editor* i *runtime*, komponenty składowe integrowane są tu w pojedynczej klasie (*CodeGeneration*), której instancja osadzana jest w aplikacji (*Application*).

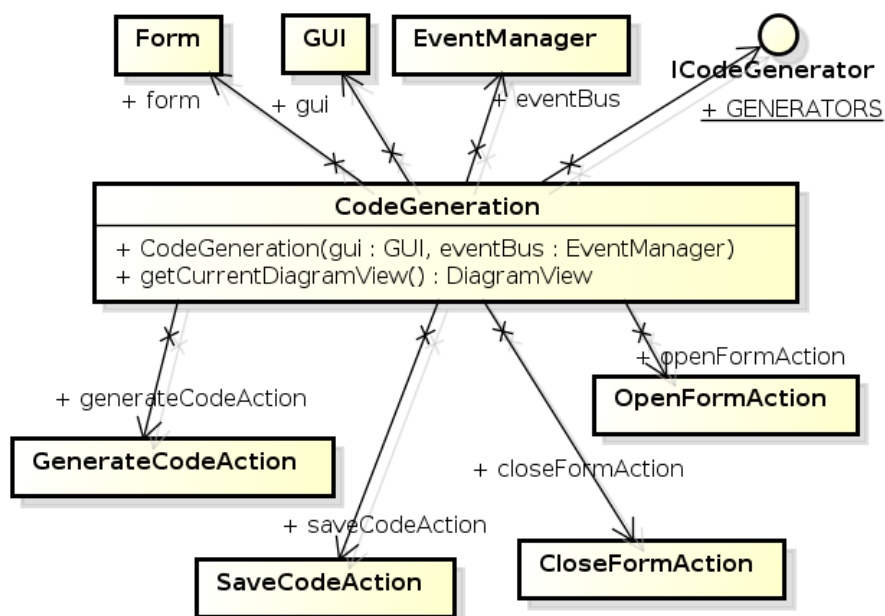
Kod źródłowy generowany jest za pomocą posiadających jednolity interfejs generatorów, z których każdy dokonuje transformacji schematu do swojego języka programowania/formatu.



Rysunek 28. Struktura pakietu codegen
(Źródło: opracowanie własne)

5.6.1. Klasa CodeGeneration

Klasa *CodeGeneration* udostępnia funkcjonalność pakietu *codegen* w postaci komponentu przystosowanego do osadzenia w głównej klasie aplikacji. Jej zadaniem jest więc podpięcie akcji do odpowiednich elementów interfejsu użytkownika i zarządzanie ich aktywnością oraz poprawna inicjalizacja dostępnych generatorów kodu.



Rysunek 29. Diagram klasy CodeGeneration
(Źródło: opracowanie własne)

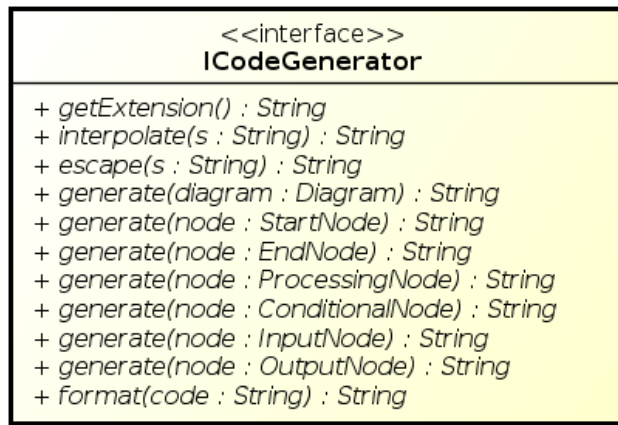
Najważniejsze metody i atrybuty:

- *GENERATORS : Map<String, ICodeGenerator>*
Kolekcja zawierająca dostępne generatory, indeksowane swoją nazwą. Na jej podstawie tworzona jest dynamicznie lista wyboru w oknie dialogowym z formularzem dot. generowania kodu.
- *form : Form*
Komponent interfejsu użytkownika reprezentujący formularz otwierany w oknie dialogowym, pozwalający na wybranieżądanego generatora oraz podgląd wygenerowanego kodu i zapisanie go do pliku.
- *openFormAction : OpenFormAction*
- *closeFormAction : CloseFormAction*
Akcje odpowiadające kolejno za: otworzenie i zamknięcie formularza (*form*) z opcjami generowania kodu.
- *generateCodeAction : GenerateCodeAction*
Akcja generująca kod źródłowy za pomocą generatora wybranego z listy wyboru. Kod wpisywany jest w pole tekstowe formularza.
- *saveCodeAction : SaveCodeAction*
Akcja wywoływana w celu zapisania wygenerowanego kodu do pliku.
- *getCurrentDiagramView() : DiagramView*
Metoda zwracająca aktywny widok schematu (obszar roboczy).

5.6.2. Interfejs *ICodeGenerator*

ICodeGenerator jest wspólnym interfejsem dla wszystkich generatorów kodu. Dzięki niemu komponenty korzystające z generatorów mogą polegać na ich jednolitym API, niezależnym od konkretnej implementacji.

Oprócz metody generującej kod dla całego schematu, interfejs specyfikuje również metody przeznaczone dla poszczególnych bloków (przeciążona metoda *generate()*), aby zapewnić poprawną transformację dla wszystkich elementów (lub wymusić świadome jej pominięcie).



Rysunek 30. Diagram interfejsu ICodeGenerator
(Źródło: opracowanie własne)

Metody publiczne:

- *generate(diagram : Diagram) : String*
Metoda ta powinna zwrócić kod wygenerowany dla całego schematu, potencjalnie wykorzystując do tego pozostałe metody interfejsu.
- *generate(node : StartNode) : String*
Metoda powinna zwrócić kod reprezentujący blok startowy. Generatory zaimplementowane w aplikacji reprezentują blok startowy jako funkcję o nagłówku zgodnym ze zdefiniowanymi w bloku: nazwą, argumentami i zwracaną zmienną, oraz ciele składającym się z kodu wygenerowanego dla kolejnych bloków. W ten sposób każda sekwencja połączonych bloków zaczynająca się od bloku startowego jest przekształcana do postaci funkcji (zgodnie z składnią danego generatora), którą można wywołać z poziomu każdego z bloków schematu.
- *generate(node : EndNode) : String*
Metoda powinna zwrócić kod odpowiadający blokowi końcowemu. W implementacji bloku startowego jako funkcji będzie to konstrukcja typu *return*.
- *generate(node : ProcessingNode) : String*
Metoda powinna zwrócić kod odpowiadający blokowi przetwarzania, a więc zawierający jego instrukcje przekształcone zgodnie z regułami składni określonymi dla danego generatora.
- *generate(node : ConditionalNode) : String*

Metoda powinna wygenerować kod reprezentujący blok decyzyjny – dla imperatywnych języków programowania będzie to forma instrukcji wyboru *if-then* lub *if-then-else*.

- *generate(node : InputNode) : String*

Metoda generująca kod dla bloku wejścia, a więc wykorzystujący odpowiednie wywołanie systemowe lub wbudowaną funkcję pozwalającą na wczytanie danych ze standardowego wejścia (np. klawiatury).

- *generate(node : OutputNode) : String*

Metoda generująca kod dla bloku wyjścia – powinna wykorzystać dostępne wywołania systemowe lub funkcje w celu wyprowadzenia danych na standardowe wyjście (np. ekran monitora).

- *interpolate(s : String) : String*

Metoda oferująca mechanizm interpolacji zmiennych – zastępowanie nazw zmiennych użytych w danym literale łańcuchowym *s* ich wartościami. W implementacjach poszczególnych generatorów funkcjonalność ta jest wykorzystywana w blokach wejścia/wyjścia.

- *format(code : String) : String*

Metoda oferująca możliwość sformatowania podanego kodu zgodnie z konwencjami przyjętymi przez dany generator.

- *getExtension() : String*

Metoda powinna zwrócić rozszerzenie zwyczajowo stosowane dla plików zawierających kod w wygenerowanym języku programowania/formacie.

5.6.3. Klasa *CCodeGenerator*

Klasa *CCodeGenerator* jest implementacją interfejsu *ICodeGenerator* generującą dla danego schematu kod w języku C++. Spośród generatorów dostępnych w aplikacji, ten cechuje się największą złożonością, głównie ze względu na statyczne typowanie języka C++ [7] wymuszające konieczność przekształcania deklaracji zmiennych, parametrów formalnych oraz typów zwracanych funkcji.

Najważniejsze metody i atrybuty:

- *INDENT : String = ' '*

Wielkość pojedynczego wcięcia logicznego bloku kodu (2 spacje). W tej implementacji kod jest formatowany podczas generowania, a nie za pomocą metody *format()*. Wykorzystanie tej metody wiązałoby się bowiem z koniecznością analizy

całości wygenerowanego kodu C++, co znacznie skomplikowałoby implementację oraz spowolniło działanie aplikacji.

- *compiler : Compiler*

Kompilator z pakietu *eval*, przekształcający schemat do kodu JavaScript na potrzeby wykonania (symulacji). Jest tutaj wykorzystywany jako analizator deklaracji typów oraz do przeprowadzania przekształceń na instrukcjach przypisanych blokom (np. w metodzie *convertArrayLevelInitializations()*)

- *generate(diagram : Diagram) : String*

Metoda generująca kod dla całego schematu *diagram*. Kod umieszczany jest w ramach szablonu zdefiniowanego w pliku *template/c.html* zgodnie z następującym układem [7]:

- Dyrektywy *#include* - umieszczone w szablonie
- Nagłówki (prototypy) funkcji - wygenerowane przez *emitFunctionHeader()* dla kolejnych bloków startowych
- Definicje funkcji - wygenerowane przez *generate(StartNode)* dla kolejnych bloków startowych
- Funkcja *main()* - jeśli schemat posiada bezargumentowy blok startowy o nazwie *Main*, to jego wywołanie zostanie automatycznie umieszczone w tej funkcji.

- *emitFunctionHeader(startNode : StartNode) : String*

Generuje nagłówek funkcji odpowiadającej danemu blokowi startowemu, na który składają się:

- komentarz (o ile istnieje)
- deklaracja typu zwracanego – *emitFunctionReturnType(startNode)*
- lista parametrów formalnych – *emitFunctionParams(startNode)*

- *generate(node : StartNode) : String*

Generuje kod dla bloku startowego: nagłówek - wykorzystując metodę *emitFunctionHeader* - oraz dla kolejno połączonych z nim bloków za pomocą metody *emitFunctionBody*.

- *emitFunctionBody(startNode : StartNode) : String*

Metoda generująca ciało funkcji reprezentującej dany blok startowy *startNode*. Emitowany kod składa się z dwóch części: deklaracji zmiennych lokalnych (*emitVariableDeclarations*) oraz instrukcji *switch* kontrolującej przepływ sterowania

między poszczególnymi blokami – dla każdego bloku generowana jest instrukcja *case* z jego indeksem oraz odpowiadającym mu kodem (*emitNodeCases*).

- *emitNodeCases(node : BaseNode, body : StringBuilder, visited : Set<DiagramNode>, startNode : StartNode) : void*
Generuje kod dla podanego bloku *node* i dopisuje go do dotychczas wygenerowanego kodu (*body*). Metoda ta interpretuje schemat jako graf skierowany i rekurencyjnie odwiedza jego kolejne wierzchołki (bloki) zgodnie z algorytmem DFS (*depth-first search*) [5], generując kod za pomocą przeciążonych dla poszczególnych typów bloków wersji metody *generate* (z interfejsu *ICodeGenerator*). Ponieważ graf reprezentujący schemat może zawierać cykle, metoda ta utrzymuje zbiór dotychczas odwiedzonych wierzchołków (*visited : Set<DiagramNode>*), aby każdy z nich został przetworzony dokładnie raz.
- *generate(node : ConditionalNode) : String*
Emituje kod dla bloku warunkowego – instrukcję *if (node.condition) {...} else {...}*.
- *generate(node : EndNode) : String*
Metoda generuje instrukcję *return node.returnValue* dla podanego bloku końcowego.
- *generate(node : InputNode) : String*
Generuje kod dla bloku wejścia wykorzystując standardowy strumień wejściowy *std::cin*. Dane przypisywane są do zmiennej zdefiniowanej w bloku.
- *generate(node : OutputNode) : String*
Generuje kod dla bloku wyjściowego wykorzystując standardowy strumień wyjściowy *std::cout*.
- *generate(node : ProcessingNode) : String*
Emituje kod dla instrukcji zawartych w bloku przetwarzania. Deklaracje zmiennych lokalnych są przenoszone na początek funkcji reprezentującej blok startowy, z którym połączony jest dany blok przetwarzania; konstrukcje takie jak odwołania do atrybutu *length* tablic zostają przekształcone na wywołania metody *size()* klasy *std::vector* itd.
- *emitTypeDeclaration(type : Type) : String*
Metoda wykorzystywana przy generowaniu deklaracji zmiennych, parametrów formalnych oraz typów zwracanych funkcji. Przekształca typy wykorzystywane w schematach na odpowiednie typy języka C++ [7], zgodnie z Tabelą 1.:

Typ użyty w schemacie	Typ C++
<i>int</i> (liczba całkowita)	<i>int</i>
<i>real</i> (liczba zmiennoprzecinkowa)	<i>float</i>
<i>string</i> (łańcuch znaków)	<i>std::string</i>
<i>bool</i> (typ bool'owski, prawda/fałsz)	<i>bool</i>
<i>T[N]...</i> (tablica <i>N</i> elementów typu <i>T</i> , gdzie <i>T</i> jest jednym z typów wymienionych w tej kolumnie)	<i>std::vector<T'>(N)</i> (wektor <i>N</i> elementów typu <i>T'</i> , gdzie <i>T'</i> jest jednym z typów w tej kolumnie)

Tabela 1. Przekształcenia typów

Argument *type* jest instancją klasy reprezentującej typ zmiennej (zdefiniowanej w pakiecie *eval*). Informacje o typach zmiennych i parametrów formalnych w obrębie każdego bloku startowego są odczytywane za pomocą komponentu *eval.Compiler*.

- *interpolate(string : String) : String*

Metoda wykonująca interpolację zmiennych w łańcuchu znakowym *string*. Zastępuje odniesienia do nazw zmiennych postaci *\$nazwa_zmiennej* konstrukcją `<< nazwa_zmiennej <<` wykorzystywaną przy wyprowadzaniu danych do strumienia wyjściowego *std::cout* w blokach wejścia/wyjścia.

5.6.4. Klasy *JavaScriptCodeGenerator* i *WebPageCodeGenerator*

Klasy te reprezentują dwie pozostałe implementacje interfejsu *ICodeGenerator*, generujące odpowiednio – kod JavaScript i stronę HTML. Ponieważ schemat działania obydwu generatorów jest zbliżony do omówionego w poprzednim rozdziale generatora kodu C++ (podobne wykorzystanie algorytmu DFS do przechodzenia po blokach schematu, przekształcanie sekwencji rozpoczynających się od bloku startowego na funkcje itd.), klasy te zostaną tutaj omówione jedynie pokrótce.

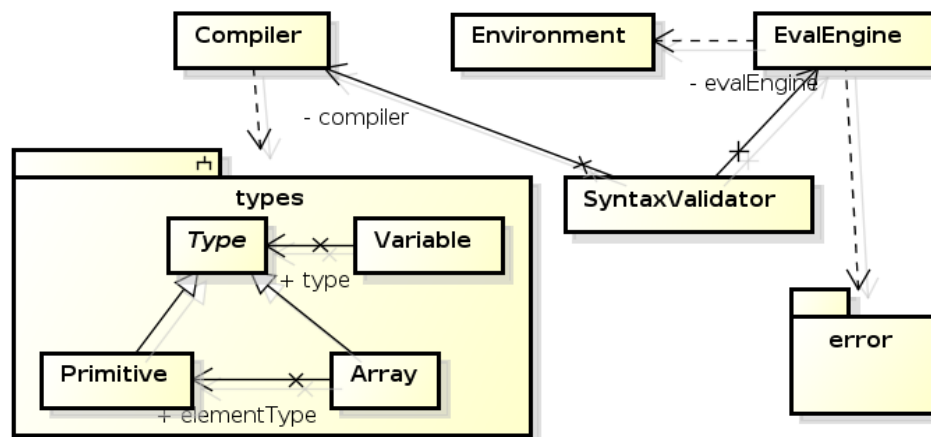
JavaScriptCodeGenerator implementuje generator kodu JavaScript. Ponieważ składnia wykorzystywana do zapisu instrukcji w blokach bazuje na tym języku, proces generowania nie wymaga praktycznie żadnych przekształceń. Jedyną wykonywaną transformacją jest usunięcie deklaracji typów zmiennych i parametrów formalnych funkcji, podyktowane dynamiczną naturą systemu typowania JavaScript (w przeciwieństwie do C++, język ten nie oferuje mechanizmu deklaracji typów) [8]. Usuwanie informacji o typach odbywa się za pomocą metody *ignoreTypeDeclarations* klasy *eval.Compiler*.

WebPageCodeGenerator generuje stronę HTML zawierającą wykonywalny kod JavaScript oraz schemat w postaci pliku PNG. Do wygenerowania kodu wykorzystywa-

ny jest wspomniany powyżej *JavaScriptCodeGenerator*. Obrazek przedstawiający schemat osadzany jest na stronie bezpośrednio (nie jest zapisywany do osobnego pliku), za pomocą mechanizmu *data URI* [11] - w atrybucie *src* tag'a *img*, zamiast ścieżki do zewnętrznego pliku, podaje się zawartość obrazka zakodowaną w formacie Base64. Dzięki temu rozwiązaniu wygenerowana strona może być zapisana w postaci pojedynczego pliku HTML, nie wymagającego odwołań do innych zewnętrznych zasobów.

5.7. Pakiet *eval*

Pakiet *eval* zawiera komponenty wykorzystywane przez pozostałe moduły aplikacji – zwłaszcza *runtime* i *codegen* – przy interpretacji i analizie instrukcji wprowadzanych w blokach schematu. Kluczowymi klasami są tu: klasa *Compiler* przekształcająca schemat w wykonywalny kod JavaScript oraz klasa *EvalEngine* dokonująca ewaluacji tego kodu.

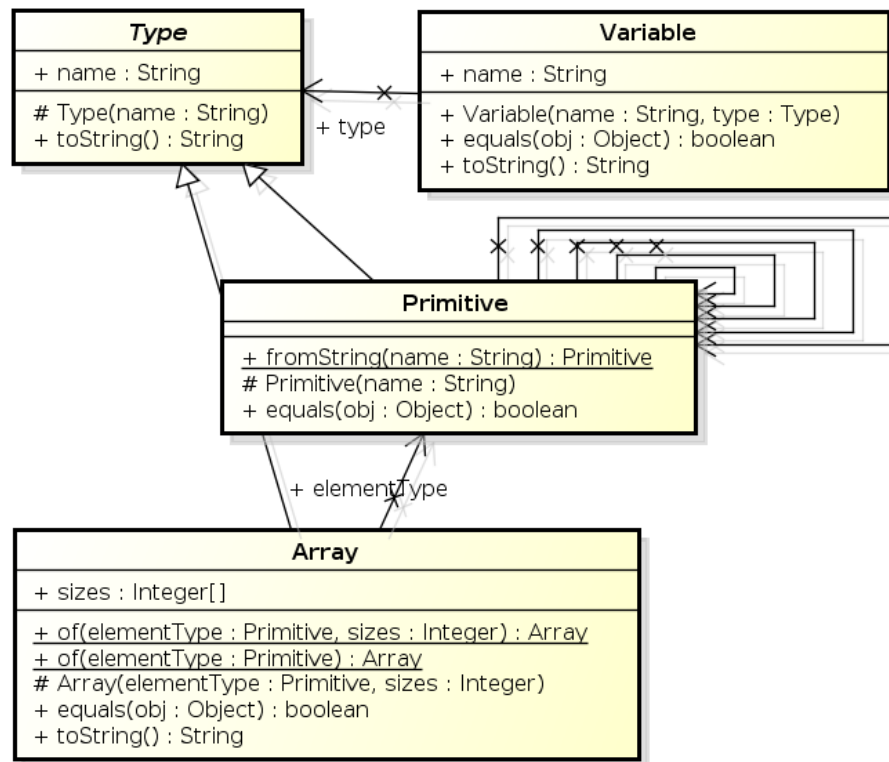


Rysunek 31. Struktura pakietu *eval*
(Źródło: opracowanie własne)

5.7.1. Typy zmiennych

System typów zmiennych (w tym również parametrów formalnych funkcji) został wprowadzony głównie na potrzeby generowania kodu w (statycznie typowanym) języku C++. Bez niego, z uwagi na to, że do ewaluacji schematu wykorzystywany jest silnik JavaScript z dynamicznym typowaniem, uzyskanie informacji o typie zmiennej byłoby możliwe dopiero w czasie wykonania. Dzięki wprowadzeniu deklaracji typów natomiast, do uzyskania tej informacji wystarczy analiza samego kodu.

Należy przy tym zaznaczyć, że deklaracje typów, jakkolwiek wymagane pod względem składniowym, funkcjonują jedynie na zasadzie podpowiedzi (dla generatorów) – i jako takie, są ignorowane w czasie wykonania (symulacji) schematu.



Rysunek 32. Diagram klas reprezentujących zmienne i ich typy
(Źródło: opracowanie własne)

System typów został zamodelowany w postaci hierarchii klas, z abstrakcyjną klasą *Type* jako bazową. Dziedziczące z niej klasy – *Primitive* i *Array* – definiują kolejno: typ podstawowy oraz typ złożony – tablicę [6].

5.7.1.1. Klasa *Primitive*

Klasa ta reprezentuje typ podstawowy – liczbę całkowitą, zmiennoprzecinkową, wartość bool'owską bądź łańcuch znaków.

Najważniejsze metody i atrybuty:

- *INT : Primitive*
- *REAL : Primitive*
- *BOOL : Primitive*
- *STRING : Primitive*

Instancje reprezentujące poszczególne typy podstawowe – kolejno: liczba całkowita, liczba zmiennoprzecinkowa, prawda/fałsz, łańcuch znaków; zdefiniowane jako statyczne pola finalne, ponieważ typy te funkcjonują w systemie na zasadzie oznaczeń (a nie kontenerów, jak w przypadku typu tablicowego), a zatem tworzenie nowej instancji dla każdej zmiennej w schemacie byłoby nieoptymalne [2].

- *fromString(name : String) : Primitive*

Zwraca instancję typu na podstawie podanej nazwy *name*; nazwa jest częścią deklaracji typu.

- *equals(obj : Object) : boolean*

Standardowa metoda klasy *Object*, z której niejawnie dziedziczą wszystkie klasy w języku Java; pozwala zdefiniować mechanizm porównywania obiektów pod względem ich „wartości” (tzn. biorąc pod uwagi ich atrybuty, a nie – co jest zachowaniem domyślnym - lokalizację w pamięci) [1]. Metoda ta została tutaj nadpisana, aby podkreślić fakt, że dwa typy podstawowe są sobie równe, jeśli ich referencje są takie same (gwarantuje to, że porównania z użyciem *equals()* oraz operatora *==* są sobie równoważne).

5.7.1.2. **Klasa Array**

Klasa ta reprezentuje (dynamiczny) typ tablicowy – kolekcję *N* obiektów tego samego typu, indeksowaną w przedziale *[0, N-1]*.

Najważniejsze metody i atrybuty:

- *elementType : Primitive*

Typ pojedynczego elementu: *Primitive.INT*, *Primitive.READ*, *Primitive.BOOL* albo *Primitive.STRING*.

- *sizes : Integer[]*

Rozmiary tablicy – ilość elementów na poszczególnych „wymiarach”. Dzięki temu można definiować tablice wielowymiarowe (których pojedyncze elementy same są tablicami), np.: *sizes = new Integer[] { 10 }* oznacza tablicę jednowymiarową (wektor) o 10 elementach; *sizes = new Integer[] { 2, 3 }* oznacza tablicę 2x3 (macierz), itd.

- *of(elementType : Primitive, sizes : Integer[]) : Array*

- *of(elementType : Primitive) : Array*

Metody fabryczne do tworzenia instancji tablicy o elementach typu *elementType* oraz wymiarach *sizes*; domyślnie *sizes = new Integer[] { 1 }*.

- *equals(obj : Object) : boolean*

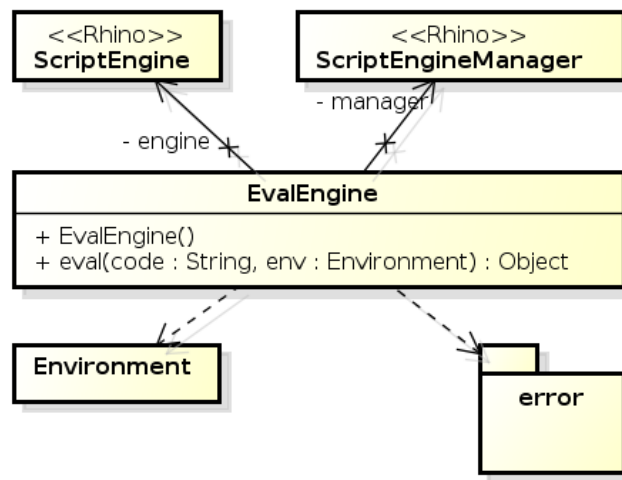
Metoda porównująca; dwa typy tablicowe są uważane za równe, jeśli posiadają taki sam typ elementów (porównywanych również za pomocą *equals()*) oraz takie same wymiary.

5.7.1.3. Klasa *Variable*

Prosta klasa reprezentująca zmienną lub parametr formalny zadeklarowany w schemacie. Każda zmienna posiada (unikalną) nazwę (atrybut *name : String*) oraz typ (atrybut *type : Type*) będący instancją klasy *Primitive* lub *Array*.

5.7.2. Klasa *EvalEngine*

Klasa *EvalEngine* jest prostym adapterem przystosowującym interpreter JavaScript (*Rhino*) - dostępny w ramach biblioteki standardowej Javy - do potrzeb ewaluacji i wykonywania schematów.



Rysunek 33. Diagram klasy *EvalEngine*
(Źródło: opracowanie własne)

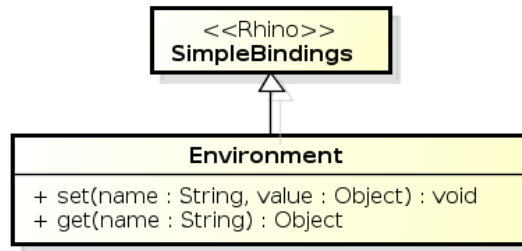
Najważniejsze metody i atrybuty:

- *eval(code : String, env : Environment) : Object*

Ewaluuje podany kod JavaScript (*code*) w ramach danego środowiska (kontekstu) *env* przechowującego nazwy i wartości bieżących zmiennych. Metoda zwraca rezultat wykonania (wartość ostatnio wykonanego wyrażenia lub zmiennej) albo wyrzuca odpowiedni wyjątek (z pakietu *eval.error*) w przypadku błędu.

5.7.3. Klasa *Environment*

Klasa reprezentująca środowisko wykonawcze pozwalające na komunikację między kodem w języku Java oraz kodem JavaScript reprezentującym wykonywalną postać schematu. Przechowuje bieżące wartości zmiennych (indeksowane nazwami) oraz udostępnia możliwość ich modyfikacji zarówno po stronie interpretera (Java) jak i interpretowanego kodu (JavaScript).



Rysunek 34. Diagram klasy *Environment*
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

- *set(name : String, value : Object) : void*

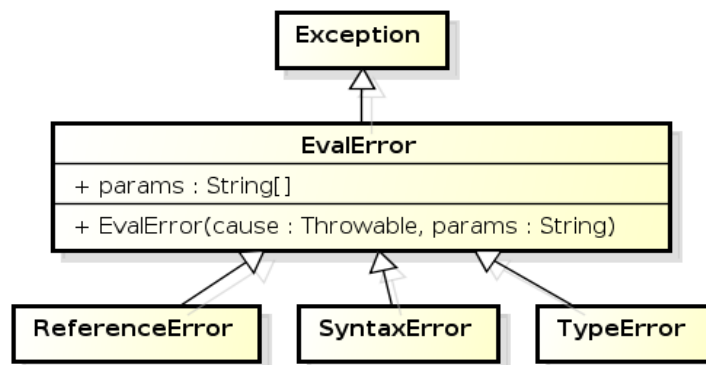
Przypisuje zmiennej o podanej nazwie *name* wartość *value*.

- *get(name : String) : Object*

Zwraca wartość zmiennej o podanej nazwie *name*, dokonując konwersji na odpowiedni typ języka Java. Zmienne numeryczne nie posiadające rozwinięcia dziesiętnego zwracane są jako *Integer* (ponieważ w JavaScript wszystkie liczby reprezentowane są wewnętrznie jako *Double*). Tablice konwertowane są na standardowe listy *java.util.ArrayList*.

5.7.4. Pakiet *eval.error*

Pakiet ten zawiera klasy reprezentujące typy błędów, które mogą wystąpić podczas ewaluacji kodu. Zorganizowano je na zasadzie hierarchicznej z podstawową klasą bazową *EvalError*.



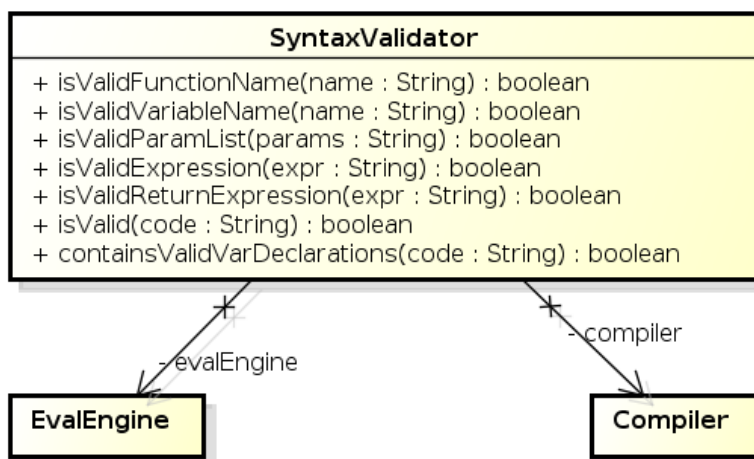
Rysunek 35. Struktura klas pakietu *eval.error*
(Źródło: opracowanie własne)

Zdefiniowane klasy błędów:

- *EvalError* – ogólny błąd ewaluacji;
- *SyntaxError* – błąd składniowy;
- *ReferenceError* – błąd (nieznanej) referencji, np.: odwołanie do niezadeklarowanej zmiennej;
- *TypeError* – nieprawidłowy typ zmiennej;

5.7.5. Klasa *SyntaxValidator*

Komponent pozwalający sprawdzić poprawność składniową fragmentu kodu użytego w schemacie. Wykorzystywany głównie w formularzach edycyjnych poszczególnych bloków.



Rysunek 36. Diagram klasy *SyntaxValidator*

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

- *isValidFunctionName(name : String) : boolean*
Zwraca *true*, jeśli podana nazwa jest poprawną nazwą funkcji (bloku startowego).
- *isValidVariableName(name : String) : boolean*
Sprawdza, czy podana nazwa jest poprawną nazwą zmiennej.
- *isValidParamList(params : String) : boolean*
Zwraca *true*, jeśli podany ciąg znaków zawiera poprawną listę parametrów formalnych funkcji (bloku startowego).
- *isValidExpression(expr : String) : boolean*
Sprawdza, czy podany ciąg znaków zawiera poprawne wyrażenie, mogące zostać użyte w kontekście bloku warunkowego.

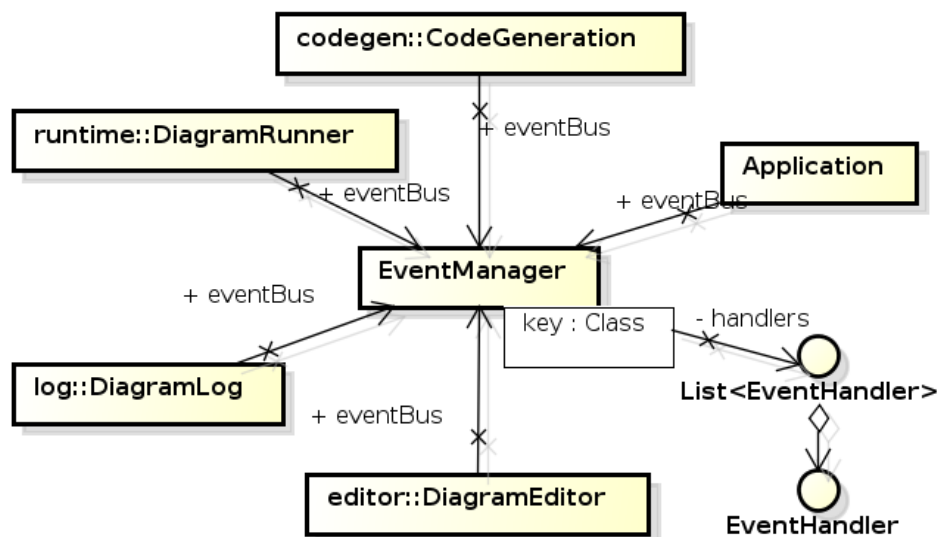
- *containsValidVarDeclarations(code : String) : boolean*

Zwraca *true*, jeśli podany kod zawiera poprawne składniowo deklaracje zmiennych (uwzględniając deklaracje typów).

5.8. Pakiet *event*

Pakiet *event* zawiera implementację prostego systemu zdarzeniowego [10], który jest wykorzystywany do komunikacji między najważniejszymi komponentami aplikacji.

System ten opiera się na zdarzeniach – obiektach określonego typu – które mogą być generowane i obsługiwane w różnych częściach aplikacji. Dzięki temu główne komponenty - edytor, symulator schematów oraz generator kodu – są od siebie niezależne, współdzielą jedynie instancję menedżera zdarzeń (*eventBus*, tworzoną w klasie *Application*), co pozwala im na emitowanie zdarzeń i definiowanie specyficznej dla siebie ich obsługi.

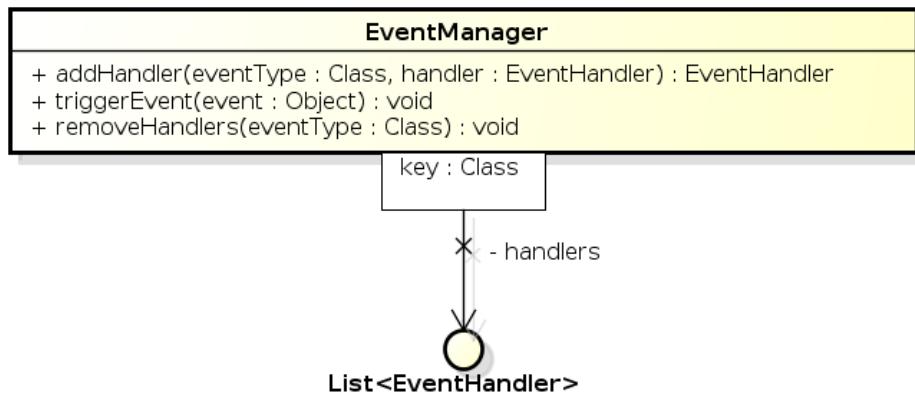


Rysunek 37. Struktura pakietu *event*

(Źródło: opracowanie własne)

5.8.1. Klasa *EventManager* i interfejs *EventHandler*

Zadaniem klasy *EventManager* jest zarządzanie mechanizmem generowania i obsługi zdarzeń. Interfejs *EventHandler* pozwala na enkapsulację kodu obsługującego dane zdarzenie (z uwagi na to, że funkcje w języku Java nie są obiektami typu *first-class* [1] – nie można ich bezpośrednio przypisywać do zmiennych, przechowywać w kolekcjach itd. - zwykle opakowuje się je w tym celu w interfejsy).



Rysunek 38. Diagram klasy *EventManager*
(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

- *handlers : Map<Class, List<EventHandler>>*
Listy handlerów – obiektów implementujących interfejs *EventHandler* – przyporządkowane poszczególnym typom (klasom) zdarzeń.
- *addHandler(eventType : Class, handler : EventHandler) : EventHandler*
Rejestruje kod obsługujący dany typ zdarzenia – dodaje nowy *handler* do odpowiedniej listy w kolekcji *handlers*. W wywołaniach tej metody, *handler* definiowany jest jako anonimowa klasa wewnętrzna implementująca *EventHandler* (Listing 1.)

```

addHandler(DiagramClosedEvent.class,
    new EventHandler<DiagramClosedEvent>() {
        public void handle(DiagramClosedEvent e) {
            // Obsługa zdarzenia...
        }
    }
);
  
```

Listing 1. Rejestracja handlera dla zdarzenia *DiagramClosedEvent*

- *removeHandlers(eventType : Class) : void*
Usuwa wszystkie handlery zarejestrowane dla danej klasy zdarzeń.
- *triggerEvent(event : Object) : void*
Powoduje zgłoszenie zdarzenia – dla każdego handlera skojarzonego z klasą danego zdarzenia wywoływana jest metoda *handle* z obiektem *event* przekazanym jako argument.

5.8.2. Typy zdarzeń

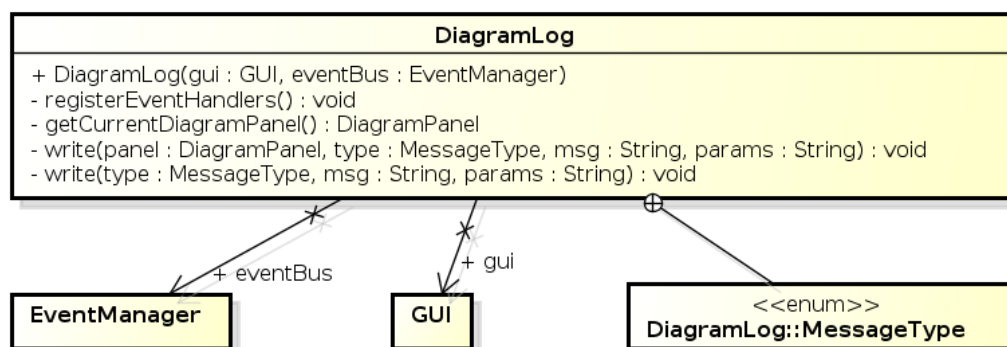
Pakiet *event* zawiera definicje wszystkich typów zdarzeń dostępnych w systemie. Zostały one zaimplementowane jako zwykłe klasy, posiadające jedynie konstruktor i publiczne pola do przechowywania specyficznych dla siebie danych (np. instancja schematu dla zdarzenia reprezentującego zamknięcie zakładki z jego obszarem roboczym itd.).

Dostępne klasy zdarzeń:

- *DiagramAddedEvent* – dodanie nowego schematu do edytora;
- *DiagramClosedEvent* – zamknięcie schematu;
- *DiagramCodeGenerated* – wygenerowanie kodu dla schematu;
- *DiagramExportedEvent* – wyeksportowanie schematu;
- *DiagramNodeAdded* – dodanie nowego bloku do schematu;
- *DiagramNodeDeleted* – usunięcie bloku;
- *DiagramRunEvent* – rozpoczęcie symulacji (uruchomienie) schematu;
- *DiagramRuntimeErrorEvent* – wystąpienie błędu podczas symulacji;
- *DiagramSavedEvent* – zapisanie schematu do pliku;
- *DiagramStoppedEvent* – zatrzymanie wykonywania (symulacji) schematu;

5.9. Pakiet log

Pakiet *log* zawiera pojedynczy komponent *DiagramLog*, którego zadaniem jest rejestrowanie (logowanie) najważniejszych zdarzeń zachodzących w aplikacji i zapisywanie ich do odpowiedniego panelu interfejsu użytkownika. Jego implementacja w oczywisty sposób opiera się na mechanizmie obsługi zdarzeń, omówionym w Rozdziale 5.8.



Rysunek 39. Diagram klasy *DiagramLog*

(Źródło: opracowanie własne)

Najważniejsze metody i atrybuty:

- *registerEventHandlers() : void*
Rejestruje zapisywanie zdarzeń w systemie wykorzystując metodę *addHandler* globalnej instancji menedżera *eventBus*.
- *write(type : MessageType, msg : String, params : String) : void*
Zapisuje komunikat *msg* (odpowiadający danemu zdarzeniu) wraz z dodatkowymi parametrami *params* (dane specyficzne dla konkretnego zdarzenia) w panelu interfejsu użytkownika. Każdy wpis posiada ponadto datę i czas oraz informację o typie określaną na podstawie przekazanej instancji wyliczenia *MessageType*: *INFO* – zwykły wpis, *SUCCESS* – pomyślne zakończenie operacji, *ERROR* – błąd.

5.10. Pakiet *ui*

Pakiet *ui* grupuje komponenty oraz zasoby bezpośrednio związane z interfejsem użytkownika. Zawiera menu, panele i formularze, zbudowane w większości przy pomocy narzędzia JFormDesigner (dostępnego w ramach IntelliJ IDEA [12]) oraz pliki ikon (w tym ikony dla poszczególnych bloków schematu).

Najważniejsze klasy:

- *MainFrame* – główne okno aplikacji;
- *DiagramPanel* – panel zawierający obszar roboczy schematu oraz panel podglądu zmiennych i log operacji;
- *GUI* – globalna instancja reprezentująca interfejs użytkownika i zapewniająca dostęp do jego poszczególnych elementów, przekazywana głównym komponentom aplikacji;

5.11. Pozostałe klasy: *Config* i *I18n*

Omówione tutaj pokrótce klasy pełnią funkcje pomocnicze w stosunku do pozostałych komponentów aplikacji. Ponieważ realizowane przez nie funkcjonalności są dość powszechne w typowych aplikacjach desktopowych, starano się aby ich implementacja była możliwie uniwersalna.

Klasa *Config* ułatwia ładowanie konfiguracji zapisanej w plikach **.properties* (jest to prosty format tekstowy typu klucz-wartość wspierany przez Javę [1]). Domyślna konfiguracja aplikacji została umieszczona w pliku *config/default.properties* i zawiera

ustawienia dotyczące języka interfejsu użytkownika, szybkości wykonywania symulacji schematów itp.

Klasa *I18n* reprezentuje komponent wspierający obsługę wersji językowych (internacjonalizację – nazwa *I18n* jest zwyczajowo stosowanym skrótem dla tego typu komponentów). Pozwala na umieszczenie tekstów i nazw używanych w aplikacji (np. w interfejsie użytkownika) w zewnętrznych plikach tekstowych w formacie **.properties*. Każdy tekst przypisywany jest do jednoznacznie identyfikującego go klucza, na podstawie którego metoda *I18n.translate()* znajduje odpowiednie tłumaczenie. Dzięki temu rozwiązaniu można łatwo (bez potrzeby modyfikowania kodu źródłowego) zmieniać aktualne teksty oraz przygotowywać kolejne wersje językowe aplikacji (tworząc nowe pliki **.properties* z tłumaczeniami).

6. TESTOWANIE

W celu weryfikacji i walidacji tworzonego projektu pod kątem wymagań oraz poprawności działania zastosowano tzw. testowanie dynamiczne, które polega na uruchomieniu aplikacji, zasileniu jej danymi testowymi, a następnie porównaniu uzyskanego wyniku bądź zaobserwowanego zachowania z oczekiwanym. W procesie tym zostały wykorzystane zarówno testy automatyczne jak i ręczne.

6.1. Testy automatyczne

Testy automatyczne zostały wykonane za pomocą popularnej biblioteki JUnit należącej do szerszej rodziny bibliotek xUnit służących do tworzenia i uruchamiania automatycznych przypadków testowych w wielu różnych językach programowania. Zaimplementowano je na poziomie pojedynczych komponentów/klas (*unit tests*), aby zweryfikować poprawność ich funkcjonowania w sposób niezależny od pozostałych części aplikacji.

Automatyczne przypadki testowe zostały stworzone dla tych kluczowych komponentów, które nie wymagają interakcji z użytkownikiem oraz są niezależne od interfejsu graficznego aplikacji - dla generatorów kodu, walidatora składni, kompilatora produkującego wykonywalny kod JavaScript oraz symulatora schematów (*RunEngine*). Poniżej omówiono przykładowy przypadek testowy dla komponentu *SyntaxValidator*.

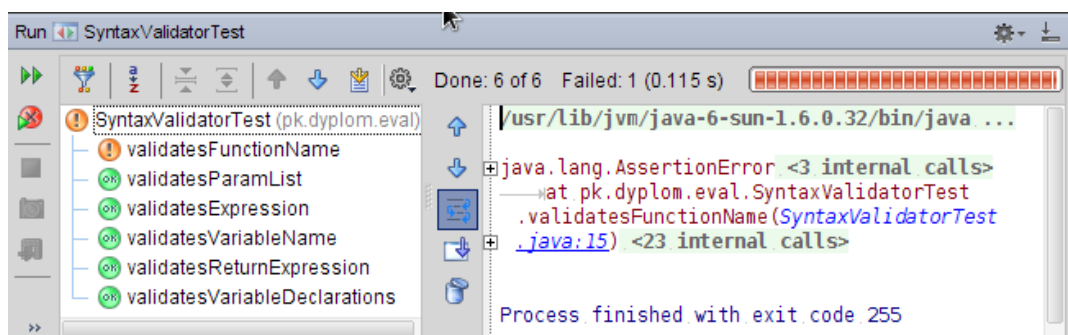
```

public class SyntaxValidatorTest {
    SyntaxValidator validator;
    @Before public void setUp() {
        validator = new SyntaxValidator();
    }
    @Test public void validatesFunctionName() {
        assertTrue(validator.isValidFunctionName("foo"));
        assertTrue(validator.isValidFunctionName("foo_bar"));
        assertTrue(validator.isValidFunctionName("$foo"));
        assertTrue(validator.isValidFunctionName("foo2"));
        assertFalse(validator.isValidFunctionName(""));
        assertFalse(validator.isValidFunctionName("foo bar"));
        assertFalse(validator.isValidFunctionName("2foo"));
    }
    // Pozostale przypadki testowe...
}

```

Listing 2. Fragment testu jednostkowego dla klasy SyntaxValidator

Listing 2. prezentuje typową strukturę testu jednostkowego wykorzystującego bibliotekę JUnit [4]. Poszczególne przypadki testowe dla danego komponentu są grupowane w klasie jako publiczne metody oznaczone specjalną adnotacją *@Test* – tutaj jest to metoda *validatesFunctionName()* reprezentująca wariant testujący walidację nazw funkcji (bloków startowych schematu). Wykorzystano w niej dwie spośród wielu udostępnianych przez JUnit asercji – *assertTrue* i *assertFalse* – badających odpowiednio: prawdziwość i fałszywość podanego warunku. W przypadku niespełnienia asercji, podczas wykonania testu wyrzucany jest wyjątek *java.lang.AssertionError* wraz ze szczegółowymi informacjami o błędzie (Rysunek 40.):



Rysunek 40. Komunikat o niespełnionej asercji podczas uruchamiania testów JUnit w IntelliJ IDEA

(Źródło: opracowanie własne)

Metoda *setUp()* z adnotacją *@Before* jest automatycznie wykonywana przed każdym przypadkiem testowym i zwykle służy do inicjalizacji i odpowiedniej konfiguracji testowanego komponentu (*pre-processing*) – tutaj tworzy instancję walidatora wykorzystywaną w poszczególnych wariantach testowych.

6.2. Testy ręczne

Jak wspomniano w poprzednim rozdziale, nie wszystkie części aplikacji zostały poddane testom automatycznym. Testy komponentów realizujących funkcjonalności ściśle oparte o interakcję z użytkownikiem czy interfejs graficzny są trudne lub wręcz niemożliwe do pełnego zweryfikowania przez automat, natomiast stosunkowo łatwe dla człowieka.

Z tego względu część aplikacji odpowiedzialna za edycję schematów – dodawanie/usuwanie bloków, przesuwanie, tworzenie i modyfikowanie połączeń – testowana była w sposób ręczny, poprzez bezpośrednią interakcję z perspektywy docelowego użytkownika. Pozwoliło to na weryfikację nie tylko poprawności funkcjonalnej, lecz także ergonomii i użyteczności (*usability*) interfejsu użytkownika.

7. INSTRUKCJA UŻYTKOWNIKA

7.1. Opis programu

Zrealizowana w ramach projektu aplikacja umożliwia tworzenie i wykonywanie schematów blokowych (diagramów reprezentujących przepływ sterowania w algorytmach/programach), oraz generowanie na ich podstawie kodu w wybranym języku programowania.

7.1.1. Wymagania

Aplikacja wymaga środowiska Java SE 6 (JRE, do pobrania ze strony Oracle).

7.1.2. Uruchomienie

Aplikacja nie wymaga instalacji. Można ją standardowo uruchomić klikając dwukrotnie na plik *aplikacja.jar* albo wpisując w linii poleceń:

```
java -jar aplikacja.jar
```

7.2. Tworzenie i edycja diagramów

7.2.1. Tworzenie diagramu


Aby utworzyć nowy diagram, należy kliknąć przycisk **Nowy** na pasku pod głównym menu lub z menu **Plik** wybrać opcję **Nowy** (można też skorzystać ze skrótu **Ctrl+N**).


Aby otworzyć już istniejący diagram należy kliknąć przycisk **Otwórz...** na pasku pod głównym menu lub z menu **Plik** wybrać opcję **Otwórz...** [**Ctrl+O**], a następnie w oknie dialogowym wybrać odpowiedni plik XML z diagramem.



7.2.2. Dodawanie i edycja bloków

Aplikacja oferuje 6 standardowych rodzajów bloków: blok startowy, blok przetwarzania, blok warunkowy, blok wejścia, blok wyjścia i blok zakończenia, reprezentowanych przez odpowiednie figury geometryczne.

7.2.2.1. Dostępne rodzaje bloków

Nazwa	Oznaczenie	Opis	Atrybuty i połączenia
Blok startowy	 START	Początek przepływu sterowania. Identyfikuje ciąg następujących po	Nazwa - ciąg składający się ze znaków <i>[0-9a-zA-z_]</i> ;

		<p>nim połączonych bloków pod swoją nazwą.</p>	<p>pozwala wywołać dany ciąg bloków, np.</p> <p><i>Sortowanie</i></p> <p>Parametry - opcjonalna lista nazw oraz typów rozdzielona przecinkami; pozwala zadeklarować parametry formalne, np. <i>lista:int[10], kierunek:string</i></p> <p>Połączenia wchodzące: BRAK Połączenia wychodzące: 1</p> <p>Diagram może mieć kilka bloków startowych.</p> <p>Dany blok startowy może zostać wywołany poprzez podanie jego nazwy oraz wartości parametrów, np. <i>Sortowanie(lista, "<")</i></p>
Blok przetwarzania	 <p>PRZETWARZANIE</p>	<p>Zawiera operacje, obliczenia, wywołania funkcji i zdefiniowanych w diagramie bloków startowych oraz deklaracje i przypisania</p>	<p>Operacje - ciąg instrukcji rozdzielonych znakami średnikami i opcjonalnie znakami nowego wiersza ; np. <i>var x:int; var y:real; x = 1;</i></p>

		zmiennych	$y = f(x);$ Połączenia wchodzące: WIELE Połączenia wychodzące: 1
Blok warunkowy	 WARUNEK	Przedstawia rozgałęzienie wykonania na podstawie wyrażenia warunkowego	Wyrażenie warunkowe - wyrażenie ewaluujące się jako prawda lub fałsz, np. $x > 0 \ \&\& \ y == 5$ Połączenia wchodzące: WIELE Połączenia wychodzące: 2 (TAK, NIE)
Blok wejścia	 WEJŚCIE	Umożliwia wprowadzenie przez użytkownika wartości dla podanej zmiennej w czasie wykonywania; wyświetla okno dialogowe z polem tekstowym	Zmienna - nazwa zmiennej, do której ma zostać przypisana wprowadzona wartość, np. a Tekst - opcjonalny tekst, który ma zostać pokazany w oknie dialogowym; aby wyświetlić wartość zmiennej należy poprzedzić jej nazwę znakiem \$, np. <i>Obecna wartość zmiennej a to \$a. Podaj</i>



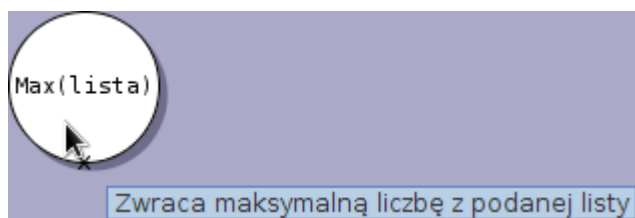
			<i>nową wartość:</i> Połączenia wchodzące: WIELE Połączenia wychodzące: 1
Blok wyjścia	 WYJŚCIE	Umożliwia wyświetlenie tekstu w oknie dialogowym podczas wykonywania	Tekst - opcjonalny tekst do wyświetlenia w oknie dialogowym; aby wyświetlić wartość zmiennej należy poprzedzić jej nazwę znakiem \$, np. <i>Wynik = \$wynik</i> Połączenia wchodzące: WIELE Połączenia wychodzące: 1
Blok zakończenia	 KONIEC	Oznacza zakończenie przepływu sterowania rozpoczętego przez blok startowy; umożliwia zwrócenie wartości do miejsca wywołania	Zwracana zmienna - nazwa zmiennej, której wartość ma zostać zwrócona jako rezultat wywołania bloku startowego (może być pusta). Połączenia wchodzące: WIELE Połączenia wychodzące: BRAK

Tabela 2. Dostępne rodzaje bloków

Oprócz wymienionych wyżej atrybutów każdy blok posiada pole **Komentarz**, które umożliwia dodanie dowolnego komentarza opisującego wykonywane w nim operacje.

Dodany komentarz jest widoczny jako *tooltip* po umieszczeniu kursora na danym bloku (Rysunek 41.)



Rysunek 41. Komentarz bloku
(Źródło: opracowanie własne)

7.2.2.2. Dodawanie nowego bloku

Aby dodać nowy blok do diagramu należy wybrać jego symbol z palety po lewej stronie lub z menu **Edycja** wybrać opcję **Wstaw** a następnie odpowiednią nazwę.

Blok zostanie umieszczony domyślnie w lewym górnym rogu obszaru roboczego.

7.2.2.3. Przesuwanie bloków

Aby przesunąć dany blok lub grupę bloków należy zaznaczyć je lewym przyciskiem myszy a następnie umieścić kursor na środku dowolnego bloku; po zmianie kursora można przeciągnąć zaznaczenie za pomocą lewego przycisku myszy w żądane miejsce.

7.2.2.4. Łączenie bloków

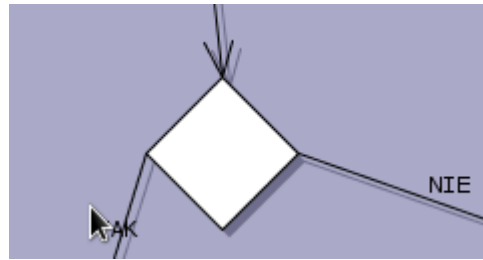
Każdy blok posiada określone punkty, z których można wyprowadzić i do których można doprowadzić połączenia:

- bloki: PRZETWARZANIE, WEJŚCIE, WYJŚCIE posiadają 4 punkty: 3 dla połączeń wejściowych i 1 dla wychodzącego w konfiguracji przedstawionej poniżej (Rysunek 42.)



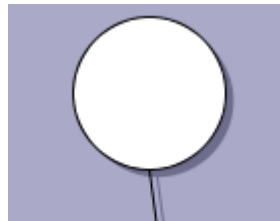
Rysunek 42. Umieszczenie punktów dla połączeń
(Źródło: opracowanie własne)

- blok WARUNEK posiada 3 punkty: 1 dla połączeń wejściowych i 2 dla połączeń wyjściowych TAK/ NIE (Rysunek 43.)



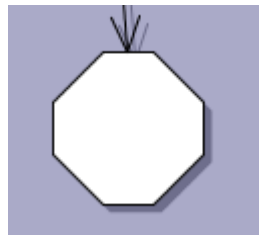
*Rysunek 43. Punkty dla połączeń bloku warunkowego
(Źródło: opracowanie własne)*

- blok START posiada tylko jeden punkt: dla jednego połączenia wychodzącego (Rysunek 44)



*Rysunek 44. Połączenia dla bloku startowego
(Źródło: opracowanie własne)*

- blok KONIEC posiada tylko jeden punkt: dla połączeń wejściowych (Rysunek 45.)



*Rysunek 45. Połączenia dla bloku końcowego
(Źródło: opracowanie własne)*

Aby **połączyć** dwa bloki ze sobą należy umieścić kursor na źródłowym bloku; punkty dla połączeń zostaną oznaczone krzyżykami (Rysunek 46.)



*Rysunek 46. Lokalizacja punktów dla połączeń
(Źródło: opracowanie własne)*

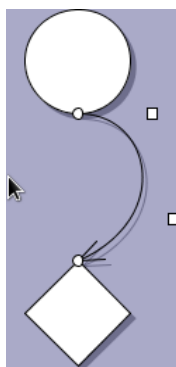
Następnie kliknąć lewym przyciskiem myszy i przeciągnąć powstałą strzałkę symbolizującą połączenie w kierunku docelowego bloku, w okolice jednego z punktów oznaczonych krzyżykami a następnie zwolnić przycisk myszy. Połączenie zostanie automatycznie skierowane do właściwego punktu akceptującego połączenia wejściowe albo nie zostanie utworzone jeśli brak takich punktów.

Uwaga

Dla połączeń wychodzących z bloku WARUNEK pierwsze tworzone połączenie oznacza zawsze wyjście TAK (o ile takie wyjście już nie istnieje).

Aby **usunąć** powstałe połączenie należy je zaznaczyć lewym przyciskiem myszy a następnie wcisnąć klawisz **Delete**.

Każde połączenie jest symbolizowane przez linię (krzywą) zakończoną strzałką i posiada dwa punkty (białe kwadraty), za pomocą których można **zmieniać jego kształt** (Rysunek 47.)

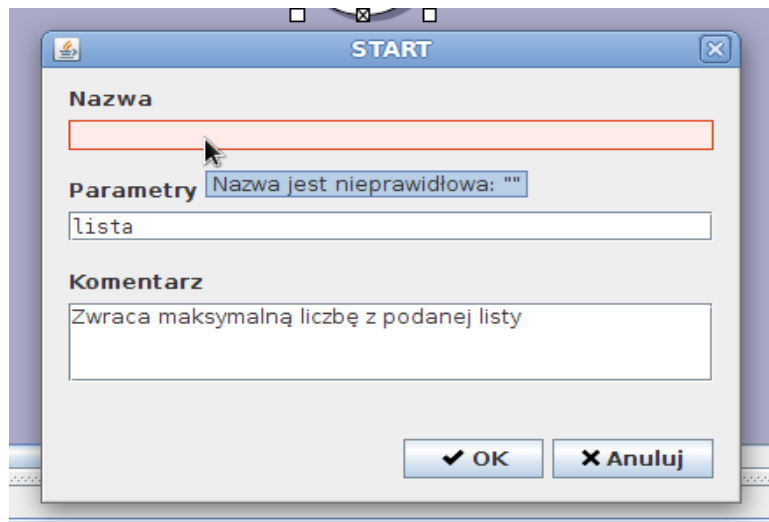


*Rysunek 47. Edycja połączenia
(Źródło: opracowanie własne)*

7.2.2.5. Edycja bloków

Aby zmienić wartości atrybutów danego bloku, wymienionych w Rozdziale 7.2.2.1., należy kliknąć dwukrotnie na nim lewym przyciskiem myszy; lub kliknąć prawym przyciskiem i z menu kontekstowego wybrać **Edytuj**; lub zaznaczyć dany blok i z menu głównego wybrać **Edycja** i następnie **Edytuj**. Zostanie wyświetlone okno dialogowe z odpowiednim formularzem dla każdego rodzaju bloku. Aby zaakceptować wprowadzone zmiany należy wybrać przycisk **OK** - zawartość bloku zostanie zmieniona. Aby anulować wprowadzone zmiany, należy wybrać **Anuluj**.

Wartości w polach formularza są sprawdzane pod kątem poprawności - w przypadku błędu dane pole tekstowe zostanie odpowiednio zaznaczone, a komunikat błędu będzie widoczny w tooltipie po umieszczeniu kursora w polu tekstowym (Rysunek 48.):



Rysunek 48. Informacje o błędach podczas edycji bloku
(Źródło: opracowanie własne)

7.2.2.6. *Usuwanie bloków*

Aby usunąć pojedynczy blok należy kliknąć na nim prawym przyciskiem myszy i z menu kontekstowego wybrać **Usuń**; lub zaznaczyć dany blok i z menu głównego wybrać **Edycja** a następnie **Usuń**. Grupę bloków można usunąć klawiszem **Delete** po uprzednim ich zaznaczeniu.

7.2.2.7. *Składnia instrukcji oraz typy zmiennych*

Typy zmiennych:

- liczba całkowita: `int`
`var i:int; i = 43;`
- liczba zmiennoprzecinkowa: `real`
`var r:real; r = 4.3;`
- łańcuch znakowy: `string`
`var s:string; s = "tekst";`
- typ logiczny: `bool`
`var b:bool; b = true; b = false;`
- tablica:
 - deklaracja
`var t:int[10]; // tablica 10 liczb int`
`var tt:string[2][2]; // tablica 2 tablic, każda zawierająca po 2 elem. typu string`

- inicjalizacja:
`t = [1,2,3,4];`
gdzie t jest tablicą zadeklarowaną jako $t: \text{int}[4]$
- pobranie i -tego elementu: `tablica[i]`, gdzie i jest indeksem zaczynającym się od 0
- pobranie długości tablicy: `tablica.length`
- wstawienie elementu e na pozycję i : `tablica[i] = e`

Składnia:

- deklaracja zmiennej:

```
var nazwa-zmiennej : typ;
nazwa-zmiennej ::= [a-zA-Z_][a-zA-Z_0-9]+
typ ::= typ-podstawowy | typ-tablicowy
typ-podstawowy ::= int | real | bool | string
typ-tablicowy ::= typ-podstawowy([rozmiar])+
rozmiar ::= [0-9]+
```

Przykłady

```
var i:int;
var s:string[10];
var b:bool[2][2];
```

- deklaracja parametrów bloku startowego: lista nazw oraz typów, rozdzielonych przecinkami - podobnie jak przy deklaracji zmiennych, z wyjątkiem słowa kluczowego `var`

```
nazwa-zmiennej : typ, ...
```

Przykład:

```
i:int, s:string[10]
```

- operatory:
 - = przypisanie wartości
 - == równość
 - > większe
 - < mniejsze
 - >= większe bądź równe
 - <= mniejsze bądź równe
 - && koniunkcja

- | | alternatywa
- + dodawanie liczb lub łączenie łańcuchów znakowych
- odejmowanie liczb
- * mnożenie liczb
- / dzielenie liczb
- % dzielenie modulo liczb
- [] indeksowanie tablicy
- wywołanie bloku startowego zdefiniowanego w diagramie:
`nazwa-bloku(lista-argumentow)`
lub (bez podawania argumentów)
`nazwa-bloku()`
gdzie *nazwa-bloku* jest ciągiem znaków [a-zA-Z_0-9] niezaczynającym się od cyfry
np.
`var lista:int[4]; Sortuj(lista);`

7.2.3. Cofanie/ponawianie operacji

Aby cofnąć ostatnio wykonaną operację na diagramie, należy wybrać z menu głównego **Edycja** opcję **Cofnij** lub skorzystać ze skrótu **Ctrl+Z**.

Aby ponowić ostatnią operację należy użyć opcji **Wykonaj ponownie** z menu głównego **Edycja** lub użyć skrótu **Ctrl+Y**.

7.2.4. Zapisywanie i zamykanie diagramu

Diagramy są zapisywane w plikach tekstowych w formacie XML.

Aby **zapisać diagram** należy kliknąć przycisk **Zapisz...** na pasku pod głównym menu; lub z menu głównego wybrać **Plik** i następnie **Zapisz...**; lub skorzystać ze skrótu **Ctrl+S**. W oknie dialogowym należy wpisać nazwę pliku (podawanie rozszerzenia **.xml** jest opcjonalne - zostanie ono automatycznie dodane w razie potrzeby).

Diagram może zostać również zapisany (wyeksportowany) jako plik graficzny (PNG/JPEG/GIF) lub plik PDF. Aby dokonać eksportu diagramu do jednego z tych formatów należy z menu głównego wybrać **Plik** a następnie **Eksportuj...**. Aby zapisać plik jako PDF należy w oknie dialogowym wybrać typ ***.pdf** i podać nazwę pliku (rozszerzenie zostanie automatycznie dodane w miarę potrzeby). Aby zapisać plik w formacie PNG/JPEG/GIF należy w oknie dialogowym wybrać typ **Pliki graficzne** i dodać odpowiednie rozszerzenie do nazwy pliku.

Aby **zamknąć diagram** należy kliknąć przycisk **Zamknij** na pasku pod głównym menu; lub z menu głównego wybrać **Plik** i następnie **Zamknij**; lub skorzystać ze skrótu **Ctrl+W**. Aktualna zakładka z diagramem zostanie zamknięta.

Uwaga:

Wszystkie *niezapisane* zmiany zostaną utracone

7.3. Wykonywanie diagramu

Aby przetestować działanie algorytmu zrealizowanego za pomocą diagramu, aplikacja oferuje możliwość interaktywnego prześledzenia przepływu sterowania oraz bieżących wartości zmiennych.

7.3.1. Uruchomienie

Każdy z bloków **START** definiuje wykonywalną jednostkę w ramach danego diagramu. Aby wykonać żądany blok startowy należy podać jego wywołanie (nazwa oraz wartości argumentów ujęte w nawiasy, zgodnie ze składnią podaną w 7.2.2.7.) w pole znajdujące się obok przycisku **Uruchom** pod głównym menu (Rysunek 49.):



Rysunek 49. Pole wyboru bloku startowego

(Źródło: opracowanie własne)

Uwaga:

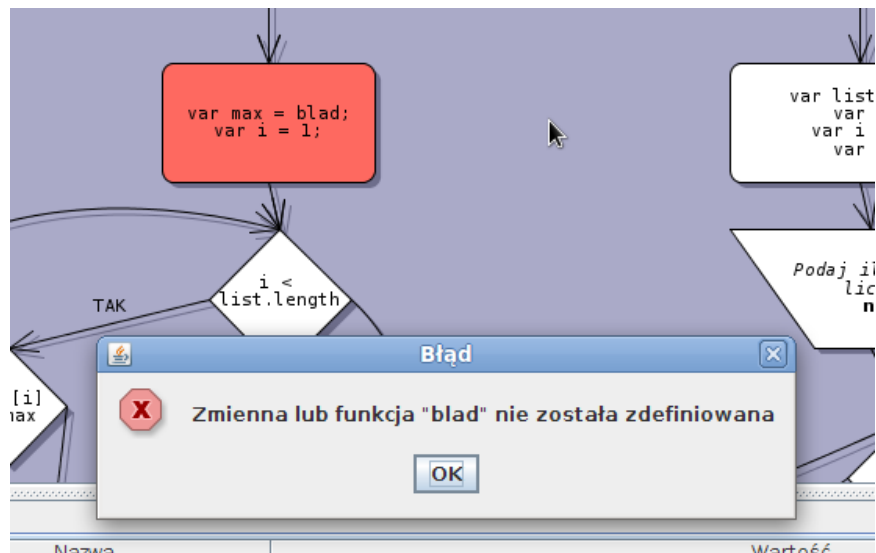
Argumenty typu tablicowego należy wprowadzić w formie literału tablicowego (z użyciem konstrukcji [...]), np.: aby wywołać blok startowy o nazwie „Sortuj” przyjmujący jako argument tablicę liczb całkowitych, należy wpisać: `Sortuj ([1, 2, 3, 4])`

Wykonanie następuje po kliknięciu przycisku **Uruchom**; lub wybraniu opcji **Uruchom** z głównego menu; lub skorzystaniu ze skrótu **Ctrl+R**.

Wykonywanie odbywa się zgodnie z przepływem sterowania określonym przez połączenia między blokami. Aktualnie przetwarzany blok jest wyróżniany kolorem.

Zakładka **Zmienne** w panelu pod diagramem zawiera listę wszystkich zmiennych wraz z ich bieżącymi wartościami, które są aktualizowane w miarę przetwarzania kolejnych bloków.

W przypadku błędu wykonywanie zostaje zatrzymane, wyświetlany jest odpowiedni komunikat, a blok który spowodował błąd jest wyróżniany kolorem (Rysunek 50.):



Rysunek 50. Błąd czasu wykonania
(Źródło: opracowanie własne)

7.3.2. Zatrzymanie wykonywania

Aby zatrzymać wykonywanie należy kliknąć przycisk **Zatrzymaj**; lub skorzystać z głównego menu **Uruchom => Zatrzymaj**; lub użyć skrótu **Ctrl+T**.

W przypadku bloku WEJŚCIE wykonywanie może być również zatrzymane klikając w oknie dialogowym przycisk **Cancel**.

7.4. Generacja kodu

Aplikacja oferuje możliwość wygenerowania na podstawie danego diagramu kodu w wybranym języku programowania. Wygenerowany kod może zostać następnie manualnie skopiowany w żądane miejsce jako tekst albo zapisany do pliku.

7.4.1. Generowanie kodu

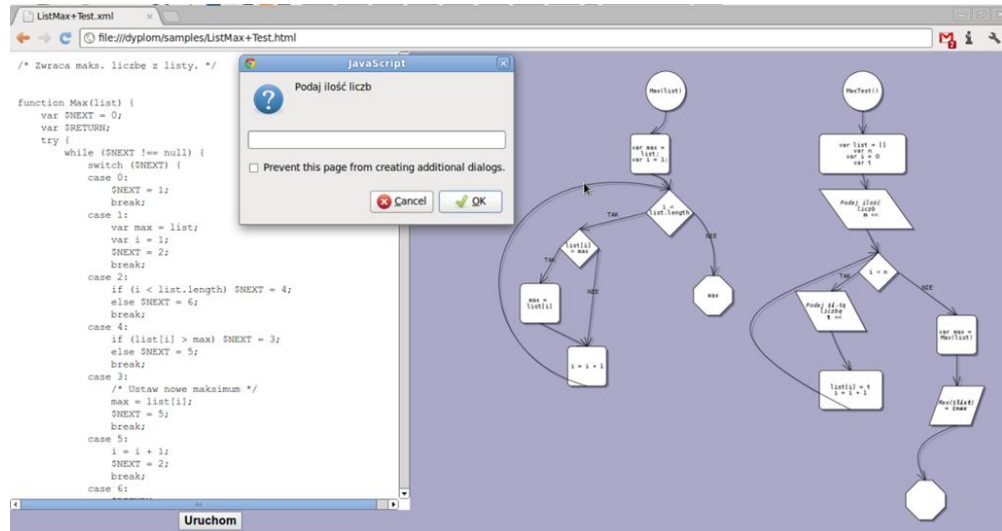
Aby wygenerować kod na podstawie diagramu w aktualnie wybranej zakładce należy kliknąć przycisk **Generuj kod...** lub wybrać opcję o tej samej nazwie w menu głównym **Kod**; lub skorzystać ze skrótu **Ctrl+G**.

Następnie, w oknie dialogowym, należy wybrać z rozwijanej listy na dole żądany język programowania, w którym ma zostać wygenerowany kod.

Dostępne opcje:

- C/C++: generuje kompletny program w języku C++. Jeśli diagram zawiera bezparametrowy blok startowy o nazwie **Main**, jego wywołanie zostanie umieszczone wewnątrz wygenerowanej funkcji **main()**.

- JavaScript
- Strona internetowa: generuje kompletną stronę HTML zawierającą graficzną reprezentację diagramu, wygenerowany kod JavaScript wraz z możliwością uruchomienia go w przeglądarce internetowej (Rysunek 51.):



Rysunek 51. Wygenerowana strona WWW
(Źródło: opracowanie własne)

Po wybraniu docelowego języka należy kliknąć przycisk **Generuj**. W polu tekstowym powinien pojawić się wygenerowany kod, który można zaznaczyć i skopiować za pomocą standardowych skrótów **Ctrl+A** **Ctrl+C** lub zapisać do pliku.

7.4.2. Zapisywanie wygenerowanego kodu

W celu zapisania wygenerowanego kodu należy kliknąć przycisk **Zapisz...** i następnie w oknie dialogowym wybrać lokalizację i wpisać nazwę pliku wraz z rozszerzeniem (np. **.cc** w przypadku C/C++, **.js** w przypadku JavaScript, **.html** w przypadku strony internetowej).

7.5. Konfiguracja

Istnieje możliwość konfiguracji kilku podstawowych parametrów działania aplikacji. Ustawienia mogą być zmienione poprzez plik **default.properties** w katalogu **config**.

Plik zawiera listę wszystkich ustawień w formacie *nazwa = wartosc* (# oznacza komentarz). Jego treść przedstawia Listing 3.:

```
# Jezyk (plik *.properties w config/)
lang = pl
# Opoznienie przy wykonywaniu diagramu [int, ms]
runtime.stepDelay = 800

# Kolor aktualnie wykonywanego bloku [hex]
runtime.currentNodeColor = #98FB98
# Kolor bloku zawierajacego blad [hex]
runtime.errorNodeColor = #FE6960
```

Listing 3. Ustawienia aplikacji

8. PODSUMOWANIE

W ramach projektu opracowana została aplikacja pozwalająca na tworzenie i testowanie schematów blokowych oraz generację kodu. Rozdział ten podsumowuje jej kluczowe wymagania funkcjonalne oraz zagadnienia związane z ich realizacją.

Aplikacja pozwala na budowanie schematów ze standardowych bloków: start, koniec, przetwarzanie, warunek, wejście, wyjście. Instrukcje i wyrażenia wprowadzane w ramach bloków zapożyczono z języka JavaScript. Na potrzeby generowania kodu w językach o statycznym typowaniu (np. C++) zaimplementowany został system typów zmiennych oferujący cztery typy podstawowe (liczba całkowita, liczba zmiennoprzecinkowa, wartość bool’owska i łańcuch znaków) oraz typ tablicowy. Ponadto aplikacja oferuje możliwość definiowania parametryzowanych procedur (podprogramów) reprezentowanych jako sekwencje bloków zaczynające się od bloków START.

Funkcje związane z testowaniem schematów zaimplementowane zostały w postaci symulatora, który przekształca w locie dany schemat do kodu JavaScript a następnie go interpretuje. Aby ułatwić analizę operacji i przepływu sterowania dodano śledzenie aktualnie wykonywanego bloku oraz wartości zmiennych.

W ramach modułu do generowania kodu zrealizowane zostały trzy generatory: dla języka C++, JavaScript oraz dla dokumentów w formacie HTML. Ze względu na statyczne typowanie C++, implementacja generatora dla tego języka wymagała odpowiednich przekształceń deklaracji zmiennych, parametrów formalnych oraz typów zwracanych funkcji. Obydwa generatory do reprezentacji przepływu sterowania w danym schemacie wykorzystują instrukcję *switch*. Rozwiązanie to, mimo że nie zawsze prowadzi do uzyskania w pełni idiomatycznego kodu (w przypadku pętli), zostało zastosowane ze względu na mniejszą podatność na przypadki brzegowe związane ze strukturą schematu i mniejszą złożoność implementacji w stosunku do rozwiązań wymagających wykrywania cykli (zwłaszcza w przypadku schematów będących tzw. grafami nieredukowalnymi posiadającymi pętle o więcej niż jednym wejściu [16]). Dzięki zdefiniowaniu wspólnego interfejsu dla wszystkich generatorów, mechanizm generowania kodu można w stosunkowo prosty sposób rozszerzać o kolejne języki programowania lub formaty.

W celu weryfikacji poprawności działania, oprócz testowania manualnego, dla kluczowych komponentów niewymagających interakcji z interfejsem użytkownika zostały zaimplementowane testy jednostkowe.

Biorąc pod uwagę zrealizowane funkcjonalności, jako potencjalny obszar zastosowań opracowanej aplikacji można wskazać dydaktykę, w szczególności nauczanie podstaw algorytmiki i programowania. Ponadto, ze względu na możliwość generowania kodu, aplikacja mogłaby również służyć jako narzędzie do szybkiego prototypowania stosunkowo prostych programów konsolowych.

9. BIBLIOGRAFIA

9.1. Literatura

1. B. Eckel, *Thinking in Java*, wydanie IV, Wydawnictwo Helion, 2006
2. J. Bloch, *Effective Java, Second Edition*, Addison-Wesley, 2008
3. J. Elliott, R. Eckstein, M. Loy et al., *Java Swing*, O'Reilly Media, 2002
4. P. Tahchiev, F. Leme, V. Masson et al., *JUnit in Action*, Manning Publications, 2010
5. T. Cormen, Ch. Leiserson, R. Rivest, *Wprowadzenie do algorytmów*, Wydawnictwa Naukowo-Techniczne, 2001
6. A. Aho, R. Sethi, J. Ullman, *Kompilatory. Reguły, metody i narzędzia*, Wydawnictwa Naukowo-Techniczne, 2002
7. B. Stroustrup, *Język C++*, Wydawnictwa Naukowo-Techniczne, 2000
8. D. Crockford, *JavaScript: The Good Parts*, O'Reilly Media, 2008
9. M. M. Sysło, *Algorytmy*, Wydawnictwa Szkolne i Pedagogiczne, 2008
10. E. Freeman, B. Bates, K. Sierra et al., *Head First Design Patterns*, O'Reilly Media, 2004
11. L. Masinter, *RFC 2397 - The "data" URL scheme*, Internet Engineering Task Force, 1998
12. D. Fields, S. Saunders, E. Belyaev, *IntelliJ IDEA in Action*, Manning Publications, 2006
13. D. Harel, *Rzecz o istocie informatyki – algorytmika*, Wydawnictwa Naukowo-Techniczne, 2001
14. S. Holzner, *Ant: The Definitive Guide*, O'Reilly Media, 2009
15. C. M. Pilato, B. Collins-Sussman, B. W. Fitzpatrick, *Version Control with Subversion*, O'Reilly Media, 2008
16. C. Cifuentes, *Control Flow Analysis [W:] Reverse Compilation Techniques*, Queensland University of Technology, 1994

9.2. Źródła internetowe

17. Dokumentacja biblioteki JDiagram,
<http://www.mindfusion.eu/onlinehelp/jdiagram/index.htm>, dostęp 29.09.2012
18. Dokumentacja biblioteki Rhino, https://developer.mozilla.org/en-US/docs/Rhino_documentation, aktualizacja 05.10.2012

19. Dokumentacja biblioteki Apache Commons Codec,
<http://commons.apache.org/codec/>, aktualizacja 11.09.2012
20. Strona internetowa aplikacji JavaBlock, <http://javablock.sourceforge.net/>, do-
stęp 11.09.2012

10. SPIS ILUSTRACJI

RYSUNEK 1. INTERFEJS APLIKACJI JAVABLOCK	7
RYSUNEK 2. INTERFEJS PAKIETU ELBOX - LABORATORIUM INFORMATYKI	9
RYSUNEK 3. INTERFEJS ŚRODOWISKA MICROSOFT VPL	10
RYSUNEK 4. DIAGRAM PRZYPADKÓW UŻYCIA	12
RYSUNEK 5. MODULARNA STRUKTURA APLIKACJI	19
RYSUNEK 6. DIAGRAM KLASY APPLICATION	19
RYSUNEK 7. HIERARCHIA KLAS REPREZENTUJĄCYCH ELEMENTY SCHEMATU	20
RYSUNEK 8. DIAGRAM KLASY BASENODE	21
RYSUNEK 9. DIAGRAM KLASY CONDITIONALNODE	23
RYSUNEK 10. DIAGRAM KLASY ENDNODE	24
RYSUNEK 11. DIAGRAM KLASY INPUTOUTPUTNODE	24
RYSUNEK 12. DIAGRAM KLASY INPUTNODE (ŹRÓDŁO: OPRACOWANIE WŁASNE)	25
RYSUNEK 13. DIAGRAM KLASY OUTPUTNODE	26
RYSUNEK 14. DIAGRAM KLASY PROCESSINGNODE	26
RYSUNEK 15. DIAGRAM KLASY STARTNODE	27
RYSUNEK 16. DIAGRAM KLAS THENLINK I ELSELINK	28
RYSUNEK 17. WYSOKOPOZIOMOWA STRUKTURA PAKIETU EDITOR	29
RYSUNEK 18. DIAGRAM KLASY DIAGRAMEDITOR	30
RYSUNEK 19. HIERARCHIA KLAS PAKIETU EDITOR.ACTION	33
RYSUNEK 20. DIAGRAM KLAS REPREZENTUJĄCYCH EKSPORTERY	35
RYSUNEK 21. DIAGRAM KLAS REPREZENTUJĄCYCH FORMULARZE EDYCJI BLOKÓW	35
RYSUNEK 22. DIAGRAM KLASY NODEFORM	36
RYSUNEK 23. DIAGRAM KLAS PAKIETU RUNTIME	37
RYSUNEK 24. DIAGRAM KLASY DIAGRAMRUNNER	38
RYSUNEK 25. DIAGRAM KLASY RUNENGINE	40
RYSUNEK 26. DIAGRAM KLASY ERRORHANDLER	41
RYSUNEK 27. DIAGRAM KLASY VARIABLESTABLEMODEL	42
RYSUNEK 28. STRUKTURA PAKIETU CODEGEN	43
RYSUNEK 29. DIAGRAM KLASY CODEGENERATION	43
RYSUNEK 30. DIAGRAM INTERFEJSU ICODEGENERATOR	45
RYSUNEK 31. STRUKTURA PAKIETU EVAL	50
RYSUNEK 32. DIAGRAM KLAS REPREZENTUJĄCYCH ZMIENNE I ICH TYPY	51
RYSUNEK 33. DIAGRAM KLASY EVALENGINE	53
RYSUNEK 34. DIAGRAM KLASY ENVIRONMENT	54
RYSUNEK 35. STRUKTURA KLAS PAKIETU EVAL.ERROR	54
RYSUNEK 36. DIAGRAM KLASY SYNTAXVALIDATOR	55
RYSUNEK 37. STRUKTURA PAKIETU EVENT	56
RYSUNEK 38. DIAGRAM KLASY EVENTMANAGER	57
RYSUNEK 39. DIAGRAM KLASY DIAGRAMLOG	58

RYSUNEK 40. KOMUNIKAT O NIESPEŁNIONEJ ASERCJI PODCZAS URUCHAMIANIA TESTÓW JUNIT W INTELLIJ	
IDEA.....	62
RYSUNEK 41. KOMENTARZ BLOKU	68
RYSUNEK 42. UMIEJSCOWIENIE PUNKTÓW DLA POŁĄCZEŃ	68
RYSUNEK 43. PUNKTY DLA POŁĄCZEŃ BLOKU WARUNKOWEGO	69
RYSUNEK 44. POŁĄCZENIA DLA BLOKU STARTOWEGO.....	69
RYSUNEK 45. POŁĄCZENIA DLA BLOKU KOŃCOWEGO.....	69
RYSUNEK 46. LOKALIZACJA PUNKTÓW DLA POŁĄCZEŃ	69
RYSUNEK 47. EDYCJA POŁĄCZENIA	70
RYSUNEK 48. INFORMACJE O BŁĘDACH PODCZAS EDYCJI BLOKU	71
RYSUNEK 49. POLE WYBORU BLOKU STARTOWEGO	74
RYSUNEK 50. BŁĄD CZASU WYKONANIA	75
RYSUNEK 51. WYGENEROWANA STRONA WWW	76