Politechnika Wrocławska

Wydział Mechaniczny

Praca dyplomowa inżynierska

Narzędzie do rysowania schematów blokowych

Autor: Jordan Wiszniewski
nr indeksu: 247859
Promotor: Dr inż. Wojciech Myszka
charakter pracy: projektowy

Rok akademicki 2021/2022

Contents

Chapter 1

Introduction

1.1 The goals of thesis

The tool was created mainly for didactic purposes – its purpose is to facilitate visualization and thus understand for a novice programmer the operation of algorithms created in C-based languages by directly showing the flowchart diagram relating to the written code. Additionally, it is a handy tool that allows to create flowcharts easily and quickly, requiring only the basics of programming for various purposes. The tool should also enable users to easily share diagrams with other users in a form that can be easily edited further.

1.2 Examples of existing solutions for drawing flowcharts

1.2.1 Raster graphic editor

The use of any graphic editor – this is the least efficient solution - drawing diagrams in this way is time-consuming and difficult to edit. It also requires a good knowledge of the editor from the user. The only advantage compared to other solutions is that the final result is not limited to predefined forms – it gives the greatest freedom.

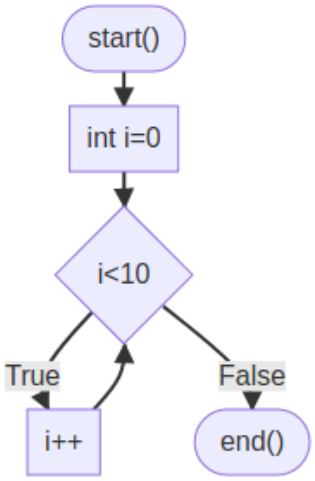
1.2.2 Tool for manual creation of diagrams

A tool specially designed for drawing various types of diagrams including flowcharts by manually placing individual blocks, describing and connecting them, e.g. lucidchart – much more efficient than when using a raster graphic editor thanks to predefined elements that the user can place and edit to a fairly large extent.

1.2.3 Code to flowchart converter

- A tool with its own syntax, created especially for drawing flowchart diagrams. The syntax itself is usually quite simple, but the code required to create the flowchart in no way relates to the algorithm it represents, and in more complex cases it is unreadable, which makes the user's work much more difficult and inefficient. One of the many examples of such a tool is Mermaid – used further in this project as an intermediary language between the input of C-based code and the final output in the form of flowchart in an HTML page.

- Converter of pseudo-programming code directly into a flowchart diagram (a variant of which is this project). A commercial version of such a converter is available on the market under the name code2flow. The advantage of these solutions is the many times higher efficiency of drawing the diagram and its direct reference to the written code, which gives a better guarantee of the correctness of the solution and the easiest edition of all available methods described before. The only disadvantage of this method is its low flexibility – the user has little influence on the final appearance of the diagram, it is largely predefined by the creators of this type of tools.



C:

```
start();
int_ i = 0;
while(i < 10){
  i++;
}
end();
```

Mermaid:

```
graph TD;
N1(["start()" ]);
N2["int i=0" ];
N5{"i<10" };
N6["i++" ];
N7(["end()" ]);
N1-->N2;
N2-->N5;
N5-->|True|N6;
N5-->|False|N7;
N6-->N5;
```

Figure 1.1: Comparison of the syntax of both types of converters that generate the same flowchart diagram.

1.3 Form of the project

The thesis is a project and consists in creating a tool with which it will be possible to easily draw (generate) flowcharts based on syntax elements taken from the C language, such as:

- function call or assignement operation – as process block
- conditional statement of type $if/else$ – a decision block with two branches representing the execution of subsequent instructions depending on the condition specified inside the decision block,
- conditional statement executing the *while* loop – as a decision block, along with a loop pointing to this block after the completion of the instructions listed inside the scope of the statement.

The generator (similarly to the C language compiler) supports the nesting of the above statements (i.e. you can extend the decision tree by placing one conditional statement inside another). The application is able to interpret only the above-mentioned instructions, which are sufficient to create most of the algorithms in practice. The deviations and shortcomings compared to the C language are the lack of such statements as: for,

do-while and goto. From the point of view of the algorithm itself, each for and do-while loops can be replaced with the most basic while loop - in the case of for: declaration of additional variables before the loop and execution of an additional statement at the end of each iteration of the loop. As in the case of do-while: replacement of the main condition of the loop to the inner conditions to call continue or break statements. The goto statement is omitted because it is fully replaceable with loops, it is also considered bad programming practice to use it. In addition, due to the fact that the application interface is an HTML form that sends requests to the server through the Rest API, after making the application available, e.g. on a cloud service, it will be possible to easily share the created diagram along with the code on the basis of which it was created using the URL link. An additional functionality is also support for Unicode characters, which allows you to write code using, among others Polish characters and text wrapping in blocks, preventing excessive block growth with larger amounts of text.

1.4 Used tools and technologies with description

All the listed technologies and tools used are open-source and do not require any additional obligations for their use:

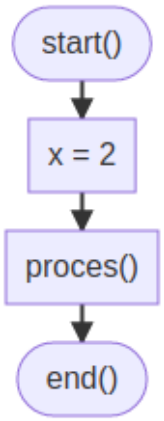
- HTML5, CSS, Thymeleaf – the user interface of the application will be a simple HTML form on a website obtained using the HTML template method, communicating with the server via the HTTP protocol and the Rest API.
- Mermaid – a tool developed in JavaScript that allows to draw diagrams on an HTML page using a special syntax, unique for this tool.
- Java 11 SE, Spring Boot framework – the back-end part of the application in which, among others include: support for Rest API queries, implementation of classes and algorithms contained in them that interpret the C-based language and convert it into the language required by the used diagram drawing tool (Mermaid)
- Used programming tools: IntelliJ IDEA Community Edition, Visual Studio Code, Git as version control system .

Chapter 2

Types of blocks and example use cases

2.1 Bloczki procesu oraz start/end

Bloczek procesu jak nazwa wskazuje odpowiada za każdą akcje. W przypadku analizy kodu programistycznego bloczek ten odpowiada wywołaniu funkcji lub przypisaniu. Bloczek start/end w przypadku tej aplikacji od bloczka procesu różni się jedynie tym, że sygnalizuje on punkt startowy oraz wszystkie możliwe zakończenia algorytmu w danym schemacie. Aplikacja automatycznie zastępuje bloczki procesu występujące na początku i na końcu bloczkiem start/end. Dodatkowo aplikacja ignoruje wszystkie znaki typu *whitespace* . Jeśli chcemy wprowadzić znak spacji do schematu należy użyć znaku "_" (underscore), który zostanie zastąpiony. Aplikacja nie wymusza użycia elementów składniowych implikujących, że jest to wywołanie funkcji (zakończenie "...();") lub przypisanie (operator "="), do narysowania tego bloczku wymagane jest jedynie, by użyte były wyłącznie znaki należące do alfabetu w dowolnym języku (mające swój odpowiednik w Unicode),



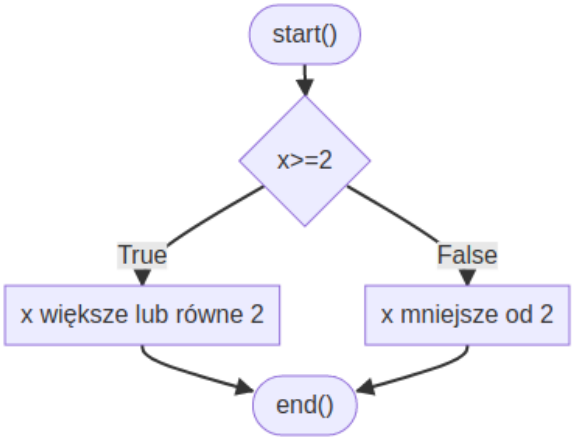

```
1 start();
2 x _ = _ 2;
3 proces();
4 end();
```

Figure 2.1: Bloczki procesu, start/end wraz z kodem, który je generuje

2.2 Bloczek warunkowy

2.2.1 dla instrukcji $if - else$

Aplikacja rozpoznaje składnie instrukcji i z podanego warunku tworzy bloczek decyzyjny. Następnie tworzy dwa rozgałęzienia odpowiadające za akcje (dowolna ilość bloczków procesu) wykonywane w przypadku spełnienia warunku (bezpośrednio pod scope'm intrukcji if) lub nie spełnienia (pod scope'm instrukcji $else$, a następnie instrukcje znajujące się poza intrukcją if)

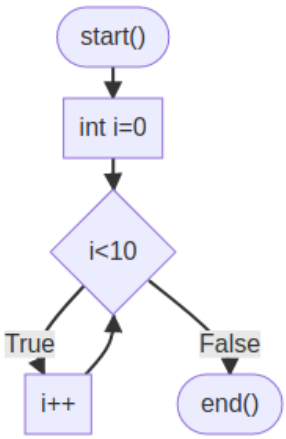



```
start();
if(x >= 2){
  x_większe_lub_równe_2;
}
else{
  x_mniejsze_od_2;
}
end();
```

Figure 2.2: Bloczek warunkowy utworzony przy użyciu instrukcji if

2.2.2 dla instrukcji $while$

Interpretacja warunku działa analogicznie jak przy instrukcji if, podobnie również działa dołączanie kolejnych instrukcji wykonujących się po spełnieniu warunku z tą różnicą, że ostatni proces tej gałęzi łączony jest automatycznie z bloczkiem decyzyjnym instrukcji $while$. Nie spełnienie warunku odpowiada gałęzi na której znajdują się procesy umieszczone poza scope'm tego typu instrukcji warunkowej.

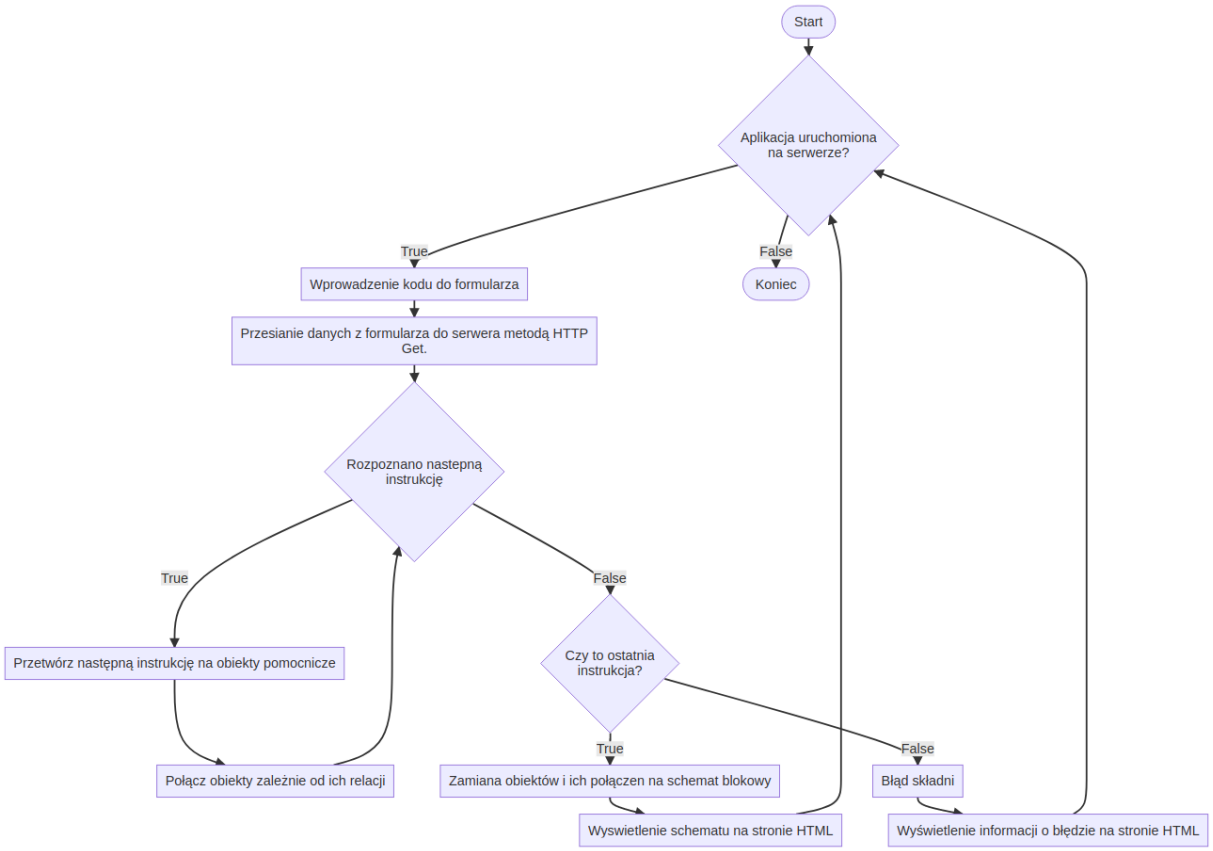



```
start();
int_ i = 0;
while(i < 10){
  i++;
}
end();
```

Figure 2.3: Bloczek warunkowy utworzony przy użyciu instrukcji $while$

2.3 Bardziej złożony przykład

Schemat blokowy przedstawiający (uproszczony) algorytm działania aplikacji w której został narysowany:




```
Start;
while(Aplikacja _ uruchomiona _ na _ serwerze?){
    Wprowadzenie_ kodu_ do_ formularza;
    Przesianie_ danych_ z_ formularza_ do_ serwera_ metodą_ HTTP_ Get.;
    while(Rozpoznano_ nastepną_ instrukcję){
        Przetwórz_ następną_ instrukcję_ na_ obiekty_ pomocnicze;
        Połącz_ obiekty_ zależnie_ od_ ich_ relacji;
    }
    if(Czy_ to_ ostatnia_ instrukcja?){
    Zamiana_ obiektów_ i_ ich_ połączeń_ na_ schemat_ blokowy;
    Wyswietlenie _ schematu _ na _ stronie _ HTML;
    } else {
        Błąd _ składni;
        Wyświetlenie _ informacji _ o _ błędzie _ na _ stronie _ HTML;
    }
}
Koniec;
```

Figure 2.4: Przykład pokazujący obsługę zagnieżdżonych instrukcji warunkowych, zawijanie tekstu oraz wprowadzanie polskich znaków

Chapter 3

Description of the application source and usage

Rozdział poświęcony jest ogólnemu opisowi działania poszczególnych fragmentów aplikacji zarówno od strony jej użytkownika jak i kodu źródłowego.

3.1 Interfejs użytkownika

Graficznym interfejsem użytkownika (GUI) jest prosty formularz HTML, domyślnie dostępny w oknie przeglądarki internetowej po uruchomieniu aplikacji pod adresem lokalnego hosta na porcie 8080:

localhost:8080/flowchart/.

Domyślnie GUI zawiera w górnej części pole z wygenerowanym diagramem blokowym, a pod nim menu rozwijane umożliwiające wybór składni kodu zarówno Mermaid jak i C-podobnego (dropdown select menu), pole tekstowe (text area) do wprowadzania samego kodu oraz przycisk potwierdzający przesłanie typu wybranej składni i kodu przez Rest API. Żądanie typu GET obsługiwane przez kontroler Rest dopuszczający dwa opcjonalne parametry: typ składni (*type*) - domyślnie przyjmujący składnie Mermaid, kod umieszczony w polu tekstowym (*originalCode*) - domyślnie rysujący dwa proste bloczki. Jakiekolwiek wyrzucenie wyjątku przez program zostaje obsłużone właśnie w kontrolerze, po czym przekazana zostaje wiadomość w nim zawarta do ekranu błędu.

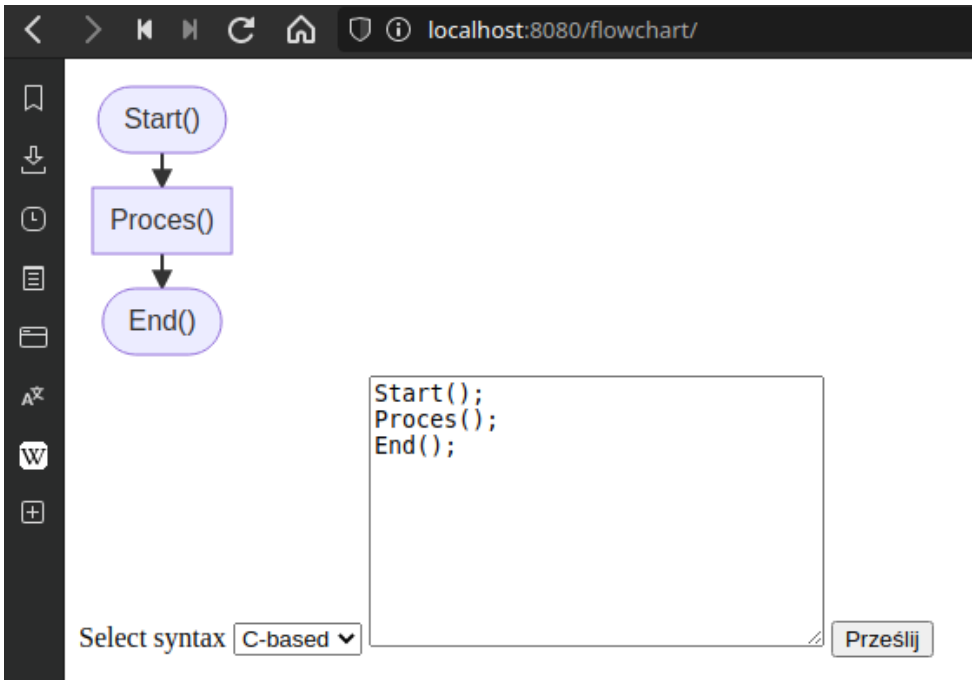
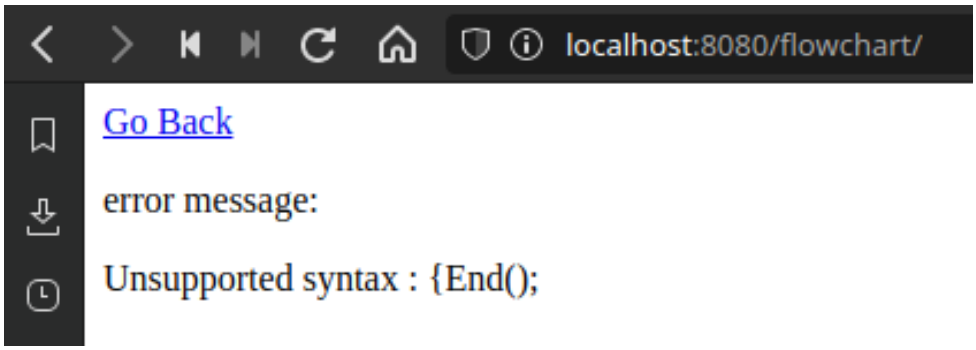


Figure 3.1: Interfejs graficzny użytkownika wyświetlony przy pomocy przeglądarki Vivaldi 5.

Przykładowy schemat blokowy dostępny pod adresem:
http://localhost:8080/flowchart/?type=C&originalCode=Start();Proces();End();

Dodatkowym ekranem jest strona HTML, która wyświetla komunikat o błędzie składniowym oraz pokazuje fragment kodu w którym ten błąd nastąpił.



Dla kodu:

```
1 Start();
2 Proces();{
3 End();
```

Figure 3.2: Przykład ekranu wyświetlającego błąd składniowy

```java
@GetMapping("/flowchart")
public ModelAndView flowchart(
    @RequestParam(value = "type", defaultValue = "mermaid") String type,
    @RequestParam(value = "originalCode", defaultValue = "A --> B") String code,
    ModelAndView mv
    ){
    try {
        mv.setViewName("flowchart.html");
        flowchartParser.setType(type);
        flowchartParser.code2flowchart(code);
        mv.addObject("flowchart", flowchartParser);
        return mv;
    } catch (Exception e){
        mv.setViewName("errorpage.html");
        mv.addObject("message", e.getMessage());
        return mv;
    }
}
```

Figure 3.3: Kontroler Rest obsługujący zapytania metodą GET protokołu HTTP z dwoma parametrami, który przekierowuje na stronę z narysowanym schematem lub ekran błędu

Do lepszego śledzenia przebiegu działania programu i ewentualnych błędów w terminalu, w którym uruchomiona jest aplikacja wyświetlają się szczegółowe logi z każdego etapu działania programu:

```
2021-12-17 00:04:31.451  INFO 39612 --- [nio-8080-exec-6] c.e.m.r.CParser
    : program instruction of type: EXPRESSION originalCode: Start();Proces();{End();
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.m.cToMermaidConverter
    : expression entered: "Start()"
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.r.CParser
    : program instruction of type: EXPRESSION originalCode: Proces();{End();
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.m.cToMermaidConverter
    : expression entered: "Proces()"
2021-12-17 00:04:31.452  INFO 39612 --- [nio-8080-exec-6] c.e.m.r.CParser
    : program instruction of type: UNKNOWN originalCode: {End();
```

Figure 3.4: Przykładowy zapis w terminalu z przebiegu programu zakończonego błędem.

3.2 Lexer i parser kodu języka C-podobnego

3.2.1 Lexer

W języku Java 11 SE stworzony został lexer kodu zintegrowany jest z parserem i jego podstawowym zadaniem jest rozpoznawanie instrukcji obsługiwanych przez aplikacje za pomocą wyrażeń regularnych oraz zapewnienia zrównoważonego użycia nawiasów zarówno okrągłych jak i klamrowych w odpowiedniej kolejności, zgodnie z zasadami programowania języka C.

```
ELSE_PATTERN = "\\}else\\{.*";
IF_PATTERN = "if\\(.*";
WHILE_PATTERN = "while\\(.*";
EXPRESSION_PATTERN = "([\\w\\+\\-=\\(\\)\\.,<>/]+?;).*";
END_SCOPE_PATTERN = "\\}.*";
```

Figure 3.5: Lista rozpoznawalnych przez lexer wyrazów regularnych (REGEX)

3.2.2 Parser

Parser kodu napisany jest w oparciu o recursive descent parser użyty m. in. w kompilatorach języka C takich jak: GCC oraz Clang. Polega on na rekursywnym wywoływaniu funkcji wczytującej kolejne instrukcje rozpoznane przez lexer oraz zależnie od rodzaju tych instrukcji wyciąga treść argumentu (w nawiasach okrągłych) oraz zawartość zakresu wewnętrznego (tzw. scope) instrukcji (w nawiasach klamrowych).

```
private void handleIfStatement(){
    String statement = programBuilder.toString();
    String condition = getStringInOuterParenthesis(statement);
    String expressions = getStringInOuterCurly(statement);
    programBuilder.delete(0, condition.length() + 5);
    onIfStatementEnter(condition, expressions);
    parseProgramInstruction();
}
```

Figure 3.6: Przykładowa funkcja, która izoluje z instrukcji *if* jej argument (condition) oraz zawartość jej obszaru wewnętrznego (expressions), a następnie przekazuje te wartości do tzw. metody obserwowanej (onIfStatementEnter)

Otwarcie nowego zakresu w tym przypadku możliwe jest jedynie w instrukcjach *if* – *else* oraz *while*. Aby umożliwić obsługę zagnieżdżeń tych instrukcji, enum reprezentujący jej typ z każdorazowym wejściem w jej zakres wewnętrzny (scope) jest odkładany na stos, a z wyjściem z zakresu zdejmowany z tego stosu co ma na celu późniejsze użycie samoczynnie wywołujących się metod obserwowanych (observable) we wzorcu projektowym *Obserwator* (*Observer*). W przypadku instrukcji odpowiadającym procesom (dowolny tekst bez znaków specjalnych zakończony średnikiem) metoda obserwująca informuje jedynie o samym jej wywołaniu, natomiast w przypadku instrukcji warunkowych występują metody dedykowane wejściu oraz wyjściu z obszaru tej instrukcji. Domyślna metoda obserwowana jedynie wyświetla w terminalu podstawowe informacje o danej instrukcji, natomiast informacje te mogą być dowolnie wykorzystane wersji tej klasy z nadpisanymi definicjami jej metod obserwowanych.

```
public void onIfStatementEnter(String condition, String expressions){
        log.info("IF entered : " + condition + " than : " + expressions);
}

public void onIfStatementExit(){
        log.info("IF exited");
}

public void onElseStatementEnter(String expressions){
        log.info("ELSE entered : " + expressions);
}
```

Figure 3.7: Przykład domyślnych definicji metod obserwowanych dla instrukcji $if/else$

3.3 Akcelerator budowania kodu w języku wymaganym przez Mermaid

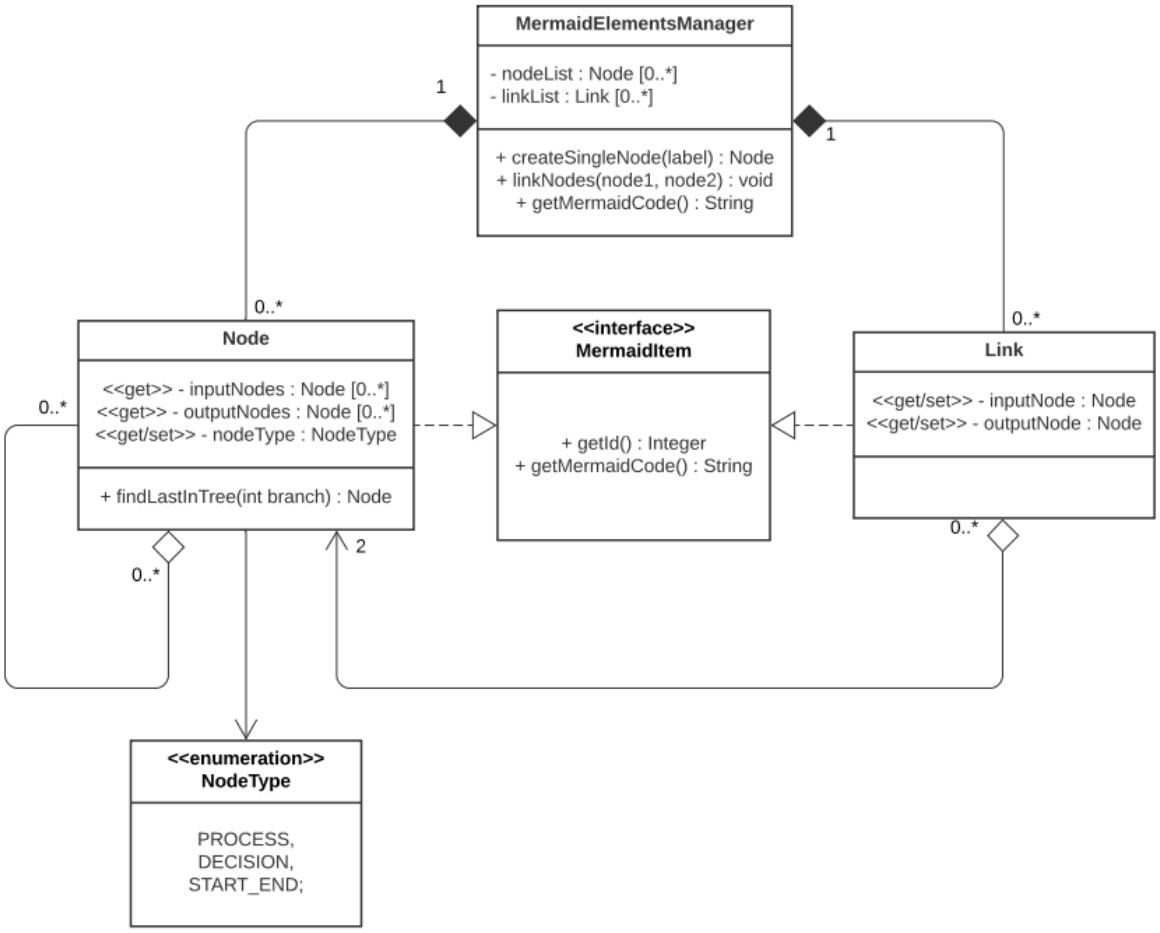


Figure 3.8: Diagram UML klas z których powstał akcelerator budowania kodu Mermaid

Do ułatwienia tworzenia schematów blokowych powstał akcelerator - zestaw klas, umożliwiający łatwe tworzenie i zarządzanie poszczególnymi elementami schematów blokowych,

a następnie konwersja tych elementów w pamięci programu na kod docelowy, wymagany przez Mermaid. Docelowo składa się on z listy węzłów (nodes) odpowiadającym poszczególnym bloczkom oraz połączeniom (links) generowanym na podstawie relacji między tymi węzłami. Za pomocą akcesorów oraz innych publicznych metod wyszczególnionych na powyższym diagramie UML można w łatwy sposób tworzyć nowe węzły, nadawać im ich typ oraz łączyć je ze sobą.

3.4 Konwerter kodu języka C-podobnego na język Mermaid

Konwerter ten wykorzystuje wcześniej wspomniany wzorzec projektowy Obserwator. W tym przypadku nadpisuje on domyślne definicje wcześniej opisanych metod obserwowanych. Zaraz po wywołaniu takiej nadpisanej metody, zależnie od tego do jakiej akcji została ona przypisana wykonuje określone akcje, również z argumentami przekazanymi przez te metody.

```
@Override
public void onIfStatementEnter(String condition, String expressions){
    condition = replaceChars(condition);
    condition = wordWrap(condition, 12);
    log.info("IF entered : " + condition + " than : " + expressions);
    manager.setLastNode(fromNode);
    NodeItem scopeNode = manager
            .createDecisionNodeLinkedToLast(condition);
    scopeNodes.push(scopeNode);
    fromNode = scopeNode.getOutputs().get(0);
}

@Override
public void onIfStatementExit(){
    log.info("IF exited");
    manager.createSingleNode("#if_merge");
    manager.linkNodes(scopeNodes.peek().findLastInTree(0),
            manager.getLastNode());
    manager.linkNodes(scopeNodes.peek().findLastInTree(1),
            manager.getLastNode());
    scopeNodes.pop();
    fromNode = manager.getLastNode();
}

@Override
public void onElseStatementEnter(String expressions){
    log.info("ELSE entered : " + expressions);
    fromNode = scopeNodes.peek().getOutputs().get(1);
}
```

Figure 3.9: Przykład nadpisanych definicji metod obserwowanych dla instrukcji $if - else$

Na powyższym przykładzie, po wejściu w instrukcję warunkową if (onIfStatementEnter) warunek zostaje poddany usunięciu wszystkich specjalnych znaków, zastąpieniu znaku "_" (underscore) przez " " (spacja). Następnie treść tego warunku poddana zostaje zawijaniu tekstu. Później po wstępnie przygotowanej zawartości warunku, z pomocą akceleratora budowania kodu Mermaid utworzony zostaje węzeł odpowiadający za bloczek decyzyjny. W związku z tym, że węzeł ten otwiera nowy obszar wewnętrzny (scope) zostaje on dodany do stosu (scopeNodes) śledzącego, który obszar jest w danym momencie wykonywany. Z bloczka decyzyjnego domyślnie wychodzą dwie gałęzi (jest połączony na wyjściu z dwoma pomocniczymi węzłami, które są niewidoczne), do końca pierwszej z nich (index = 0) przyłączane są kolejno instrukcje zawarte w obszarze wewnętrznym, odczytywane rekurencyjnie zgodnie z działaniem parsera, natomiast do końca drugiej gałęzi (index = 1) zostają przyłączane instrukcje po wywołaniu funkcji odpowiadającej instrukcji $else$. Po przetworzeniu całej zawartości opisywanej instrukcji warunkowej zostaje utworzony kolejny niewidoczny węzeł pomocniczy, który łączy obie gałęzie w momencie wychodzenia z obszaru wewnętrznego $if - else$ (wywołanie metody obserwowanej onIfStatementExit). Na koniec węzeł decyzyjny, którego obszar właśnie został zamknięty, zostaję zdjęty ze stosu.