



# 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

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
1.1	Cele pracy . . . . .	2
1.2	Przykłady istniejących rozwiązań dla rysowania schematów blokowych typu flowchart . . . . .	2
1.2.1	Edytor graficzny . . . . .	2
1.2.2	Narzędzie do ręcznego rysowania schematów . . . . .	2
1.2.3	Konwerter kodu na schemat blokowy . . . . .	2
1.3	Forma pracy . . . . .	3
1.4	Wykorzystane narzędzia i technologie wraz z opisem . . . . .	4
<b>2</b>	<b>Typy blozków oraz ich przykładowe zastosowania</b>	<b>5</b>
2.1	Bločki procesu oraz start/end . . . . .	5
2.2	Bloček warunkowy . . . . .	6
2.2.1	dla instrukcji <i>if – else</i> . . . . .	6
2.2.2	dla instrukcji <i>while</i> . . . . .	6
2.3	Bardziej złożony przykład . . . . .	7
<b>3</b>	<b>Opis działania aplikacji</b>	<b>8</b>
3.1	Interfejs użytkownika . . . . .	8
3.2	Lexer i parser kodu języka C-podobnego . . . . .	10
3.2.1	Lexer . . . . .	10
3.2.2	Parser . . . . .	11
3.3	Akcelerator budowania kodu w języku wymaganym przez Mermaid . . . . .	12
3.4	Konwerter kodu języka C-podobnego na język Mermaid . . . . .	13

# Rozdział 1

## Wstęp

### 1.1 Cele pracy

Narzędzie powstało głównie w celach dydaktycznych – ma ona na celu ułatwienie wizualizacji i tym samym zrozumienie początkującemu programiście działania algorytmów utworzonych w językach bazujących na języku C poprzez bezpośrednie pokazanie schematu blokowego odnoszącego się do napisanego kodu. Dodatkowo jest to przydatne narzędzie umożliwiające tworzenie schematów blokowych w łatwy i szybki sposób, wymagający jedynie podstaw programowania do dowolnego zastosowania. Narzędzie powinno również umożliwiać użytkownikom łatwe udostępnianie schematów innym użytkownikom w formie, którą łatwo można poddać dalszej edycji.

### 1.2 Przykłady istniejących rozwiązań dla rysowania schematów blokowych typu flowchart

#### 1.2.1 Edytor graficzny

Użycie dowolnego edytora graficznego – jest to najmniej wydajne rozwiązanie – rysowanie diagramów w ten sposób jest czasochłonne oraz trudne do ewentualnej edycji. Również wymaga od użytkownika dobrej znajomości edytora. Jediną zaletą na tle innych rozwiązań jest nieograniczenie wyniku końcowego do predefiniowanych form – daje największą swobodę.

#### 1.2.2 Narzędzie do ręcznego rysowania schematów

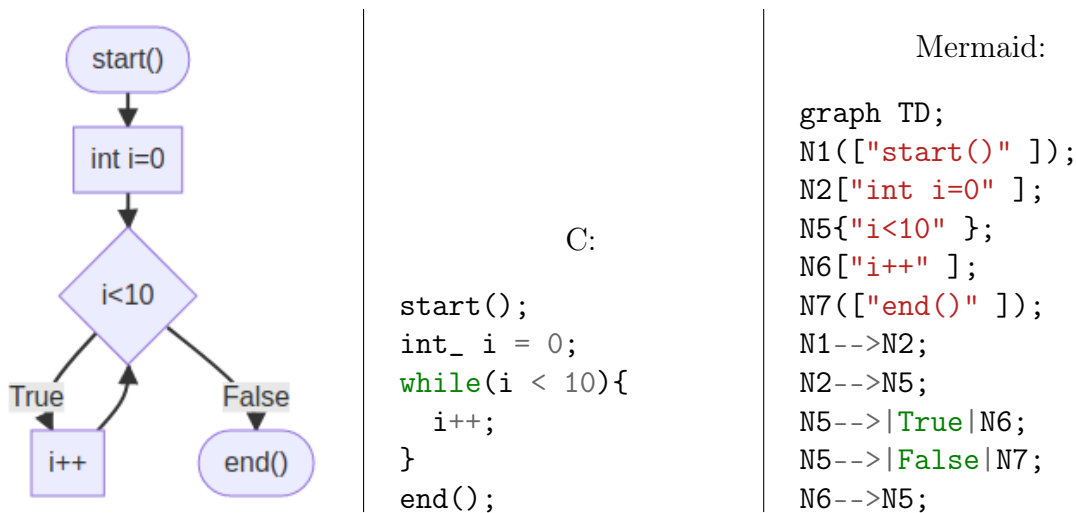
Narzędzie specjalnie przeznaczone do rysowania za pomocą ręcznego umieszczania pojedynczych bloków oraz ich opisywania i łączenia np. [lucidchart](#) – dużo wydajniejsze niż w przypadku użycia edytora graficznego dzięki gotowym elementom, które użytkownik może umieszczać i edytować w dość dużym zakresie.

#### 1.2.3 Konwerter kodu na schemat blokowy

- Narzędzie posiadające swoją własną składnię, stworzoną specjalnie na potrzeby rysowania diagramów (m. in. blokowych typu flowchart). Składnia sama w sobie zwykle jest dość prosta, jednak kod wymagany do stworzenia schematu blokowego w żaden

sposób nie reprezentuje algorytmu, który przedstawia, a w bardziej złożonych przypadkach jest on nieczytelny co znacznie utrudnia i spowalnia pracę użytkownika. Jednym z wielu przykładów takiego narzędzia jest [Mermaid](#) – używany dalej w tym projekcie jako język pośredniczący pomiędzy wprowadzanym kodem C-podobnym, a samym rysowaniem schematów na stronie HTML.

- Konwerter pseudo-kodu programistycznego wprost do schematu blokowego (którego wariantem jest ten projekt). Na rynku dostępna jest komercyjna wersja takiego konwertera pod nazwą [code2flow](#). Zaletą tych rozwiązań jest wielokrotnie wyższa wydajność rysowania schematu oraz bezpośrednie odniesienie go do napisanego kodu co daje lepszą gwarancję poprawności rozwiązania oraz najłatwiejszą edycję spośród wszystkich dostępnych metod. Na niekorzyść tej metody działa jedynie jej mała elastyczność – użytkownik ma niewielki wpływ na końcowy wygląd schematu, jest to w dużej mierze wstępnie zdefiniowane przez twórców tego typu narzędzi.



Rysunek 1.1: Porównanie składni obu typów konwerterów, które generują ten sam schemat blokowy.

## 1.3 Forma pracy

Praca ma charakter projektowy i polega na stworzeniu narzędzia, za pomocą którego będzie można w łatwy sposób narysować (wygenerować) schematy blokowe typu flowchart na podstawie elementów składniowych zaczerpniętych z języka C, takich jak:

- wywołanie funkcji lub przypisanie – jako bloczek procesu
- instrukcje warunkowe typu *if/else* – bloczek decyzyjny z dwiema gałęziami reprezentującymi wykonywanie kolejnych instrukcji w zależności od warunku podanego wewnątrz blocka decyzyjnego,
- instrukcja warunkowego wykonywania pętli *while* – jako bloczek decyzyjny, wraz z pętlą wskazującą na ten bloczek po zakończeniu instrukcji zawartych poniżej.

Generator (podobnie jak kompilator języka C) obsługuje zagnieżdżenie w sobie powyższych instrukcji (tzn. można rozbudować drzewko decyzyjne umieszczając jedną instrukcję warunkową wewnątrz drugiej). Aplikacja jest w stanie zinterpretować jedynie wyżej wymienione instrukcje, które są wystarczające do stworzenia większości algorytmów w praktyce. Odstępstwami i jednocześnie brakami w porównaniu do języka C jest brak takich instrukcji jak: `for`, `do-while` oraz `goto`. Z punktu widzenia samego działania algorytmu, każdą pętlę `for` oraz `do-while` można zastąpić najbardziej podstawową pętlą `while` – w przypadku `for`: zadeklarowanie dodatkowych zmiennych przed pętlą oraz wywołanie dodatkowej instrukcji na końcu każdej iteracji pętli, natomiast w przypadku `do-while`: zamiana warunku głównego pętli na wewnętrzne warunki wywołania instrukcji `continue` lub `break`. instrukcja `goto` została pominięta ponieważ również można ją w pełni zastąpić pętlami, używanie tej instrukcji uchodzi również za złą praktykę programistyczną. Dodatkowo w związku z tym, że interfejsem aplikacji jest formularz HTML, wywołujący zapytania do serwera poprzez Rest API, po udostępnieniu aplikacji np. na platformie chmurowej, będzie można w łatwy sposób udostępniać utworzony schemat wraz z kodem na podstawie którego powstał za pomocą linku. Dodatkową funkcjonalnością jest również obsługa znaków Unicode, co umożliwia pisanie kodu z użyciem m.in. polskich znaków oraz zawijanie tekstu w blockach, zapobiegając nadmiernemu rozrostowi bloków przy większej ilości tekstu.

## 1.4 Wykorzystane narzędzia i technologie wraz z opisem

Wszystkie użyte technologie i narzędzia są open-source i do ich użytku (nawet komercyjnego) nie wymagają żadnych dodatkowych obligacji:

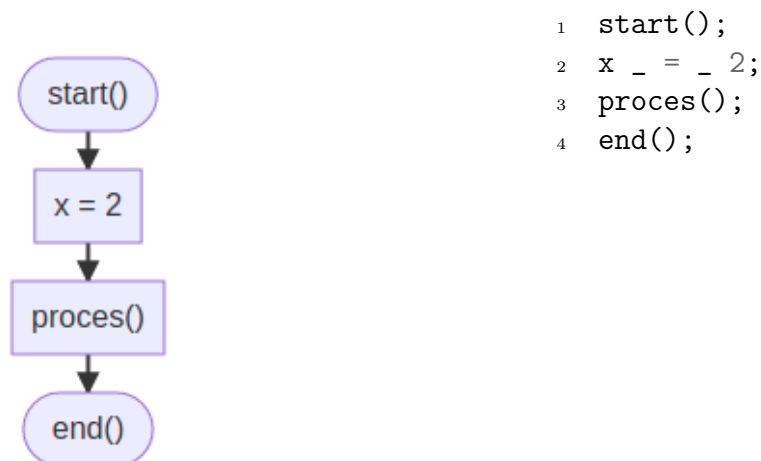
- HTML5, CSS, Thymeleaf – interfejsem aplikacji będzie prosty formularz w formie strony internetowej uzyskanej metodą szablonu HTML, komunikującej się z serwerem poprzez Rest API.
- Mermaid – narzędzie napisane w JavaScript umożliwiające rysowanie schematów na stronie HTML za pomocą specjalnej składni, unikalnej dla tego narzędzia.
- Java 11 SE, Spring Boot framework – część back-endowa aplikacji w którą m. in. wchodzi: obsługa zapytań Rest API, implementacja klas i zawartych w nich algorytmów interpretujących język C-podobny oraz zamieniających go na język wymagany przez narzędzie do rysowania schematów (Mermaid)
- Użyte narzędzia programistyczne: IntelliJ IDEA Community Edition, Visual Studio Code, system kontroli wersji Git.

## Rozdział 2

# Typy blozków oraz ich przykładowe zastosowania

### 2.1 Blozki procesu oraz start/end

Blozek procesu jak nazwa wskazuje odpowiada za każdą akcję. W przypadku analizy kodu programistycznego blozek ten odpowiada wywołaniu funkcji lub przypisaniu. Blozek start/end w przypadku tej aplikacji od blozka procesu różni się jedynie tym, że sygnalizuje on punkt startowy oraz wszystkie możliwe zakończenia algorytmu w danym schemacie. Aplikacja automatycznie zastępuje blozki procesu występujące na początku i na końcu blozkiem 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 blozku wymagane jest jedynie, by użyte były wyłącznie znaki należące do alfabetu w dowolnym języku (mające swój odpowiednik w Unicode),

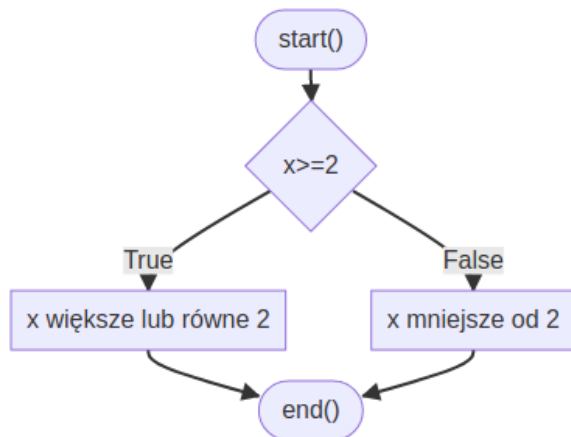


Rysunek 2.1: Blozki procesu, start/end wraz z kodem, który je generuje

## 2.2 Bloczek warunkowy

### 2.2.1 dla instrukcji *if – else*

Aplikacja rozpoznaje składnię instrukcji i z podanego warunku tworzy bloczek decyzyjny. Następnie tworzy dwa rozgałęzienia odpowiadające za akcje (dowolna ilość bloków procesu) wykonywane w przypadku spełnienia warunku (bezpośrednio pod scope’em instrukcji *if*) lub nie spełnienia (pod scope’em instrukcji *else*, a następnie instrukcje znajdujące się poza instrukcją *if*)

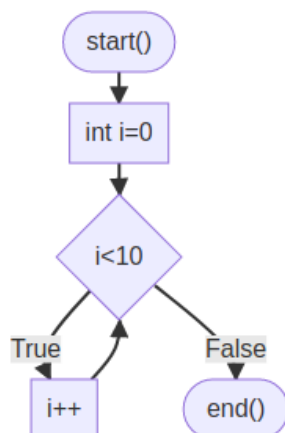


```
1 start();
2 if(x >= 2){
3     x_wieksze_lub_rowne_2;
4 }
5 else{
6     x_mniejsze_od_2;
7 }
8 end();
```

Rysunek 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 blokiem decyzyjnym instrukcji *while*. Nie spełnienie warunku odpowiada gałęzi na której znajdują się procesy umieszczone poza scope’em tego typu instrukcji warunkowej.

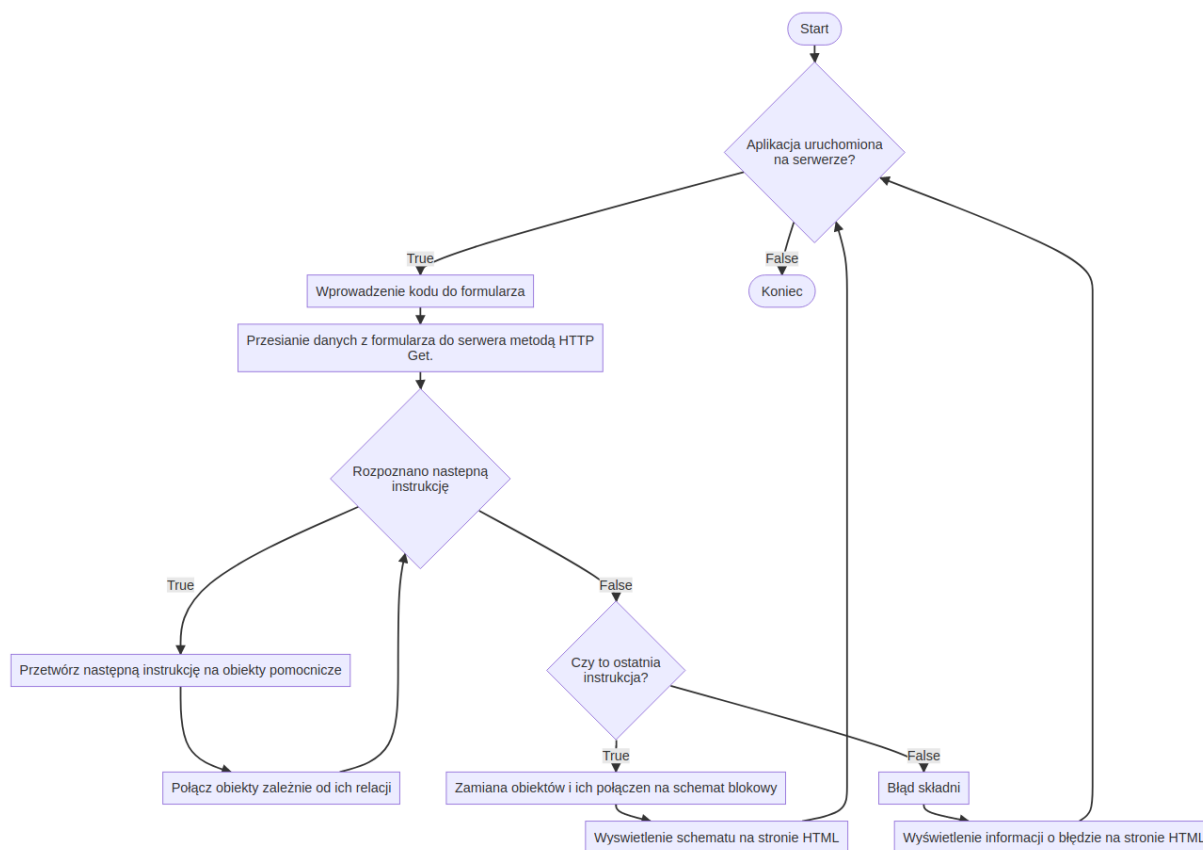


```
1 start();
2 int_ i = 0;
3 while(i < 10){
4     i++;
5 }
6 end();
```

Rysunek 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:



```

1 Start;
2 while(Aplikacja _ uruchomiona _ na _ serwerze?){
3   Wprowadzenie_kodu_do_formularza;
4   Przesłanie_danych_z_formularza_do_serwera_metodą_HTTP_Get.;
5   while(Rozpoznano_następną_instrukcję){
6     Przetwórz_następną_instrukcję_na_obiekty_pomocnicze;
7     Połącz_obiekty_zależnie_od_ich_relacji;
8   }
9   if(Czy_to_ostatnia_instrukcja?){
10    Zamiana_obiektów_i_ich_połączeń_na_schemat_blokowy;
11    Wyświetlenie _ schematu _ na _ stronie _ HTML;
12  } else {
13    Błąd _ składni;
14    Wyświetlenie _ informacji _ o _ błędzie _ na _ stronie _ HTML;
15  }
16 }
17 Koniec;
  
```

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



# Rozdział 3

## Opis działania aplikacji

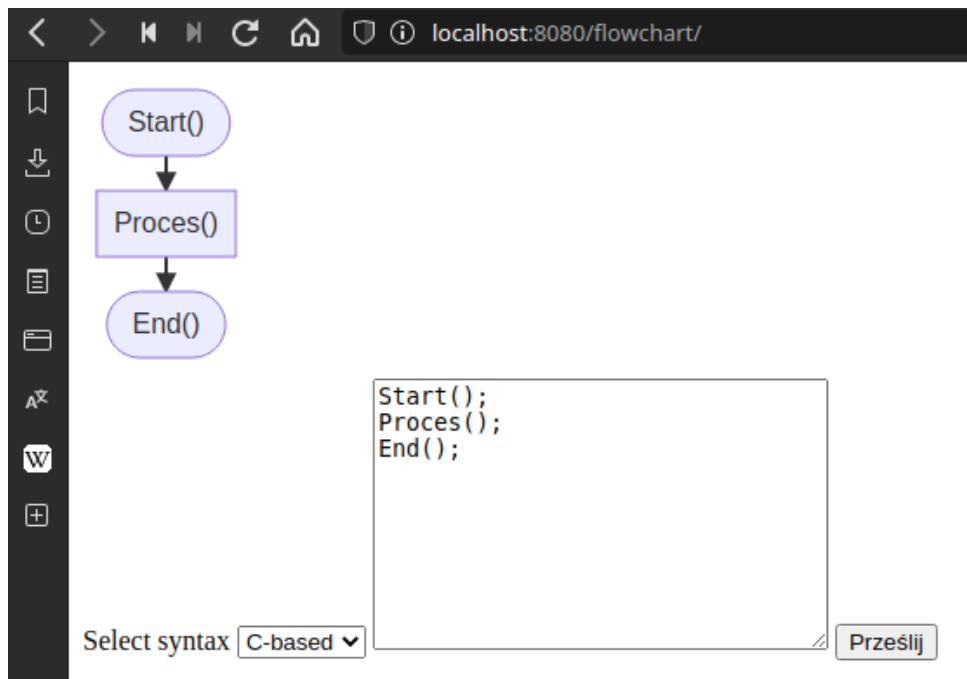
Rozdział poświęcony jest ogólnemu opisowi działania poszczególnych fragmentów aplikacji zarówno od strony jej użytkownika jak i kodu źródłowego.

### 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/](http://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 składnię Mermaid, kod umieszczony w polu tekstowym (*originalCode*) - domyślnie rysujący dwa proste blozki. Jakiegokolwiek wyrzucenie wyjątku przez program zostaje obsłużone właśnie w kontrolerze, po czym przekazana zostaje wiadomość w nim zawarta do ekranu błędu.

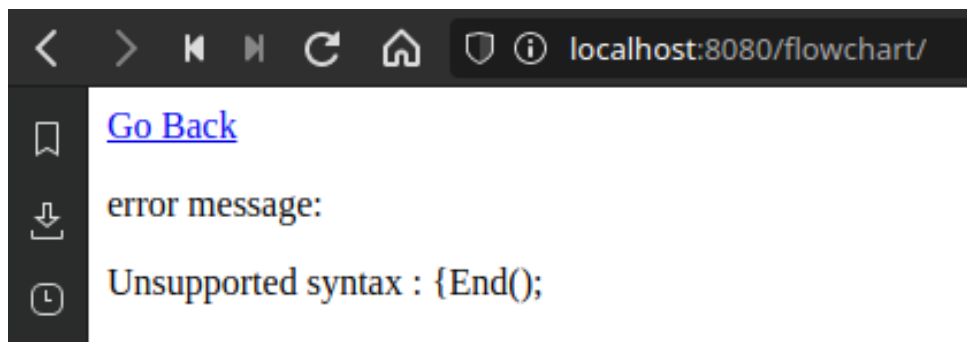


Rysunek 3.1: Interfejs graficzny użytkownika wyświetlony przy pomocy przeglądarki Vi-  
valdi 5.

Przykładowy schemat blokowy dostępny pod adresem:

[http://localhost:8080/flowchart/?type=C&originalCode=Start\(\);Proces\(\);End\(\);](http://localhost:8080/flowchart/?type=C&originalCode=Start();Proces();End();)

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



Dla kodu:

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

Rysunek 3.2: Przykład ekranu wyświetlającego błąd składniowy

```

1  @GetMapping("/flowchart")
2  public ModelAndView flowchart(
3      @RequestParam(value = "type", defaultValue = "mermaid") String type,
4      @RequestParam(value = "originalCode", defaultValue = "A --> B") String code,
5      ModelAndView mv
6  ){
7      try {
8          mv.setViewName("flowchart.html");
9          flowchartParser.setType(type);
10         flowchartParser.code2flowchart(code);
11         mv.addObject("flowchart", flowchartParser);
12         return mv;
13     } catch (Exception e){
14         mv.setViewName("errorpage.html");
15         mv.addObject("message", e.getMessage());
16         return mv;
17     }
18 }

```

Rysunek 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();

```

Rysunek 3.4: Przykładowy zapis w terminalu z przebiegu programu zakończony 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 aplikację za pomocą wyrażeń regularnych oraz zapewnienia zrównoważonego użycia nawiasów zarówno okrągłych jak i klamrowych w odpowiedniej kolejności, zgodnie z zasadami programowania języka C.

```

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

```

Rysunek 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).

```

1 private void handleIfStatement(){
2     String statement = programBuilder.toString();
3     String condition = getStringInOuterParenthesis(statement);
4     String expressions = getStringInOuterCurly(statement);
5     programBuilder.delete(0, condition.length() + 5);
6     onIfStatementEnter(condition, expressions);
7     parseProgramInstruction();
8 }

```

Rysunek 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.

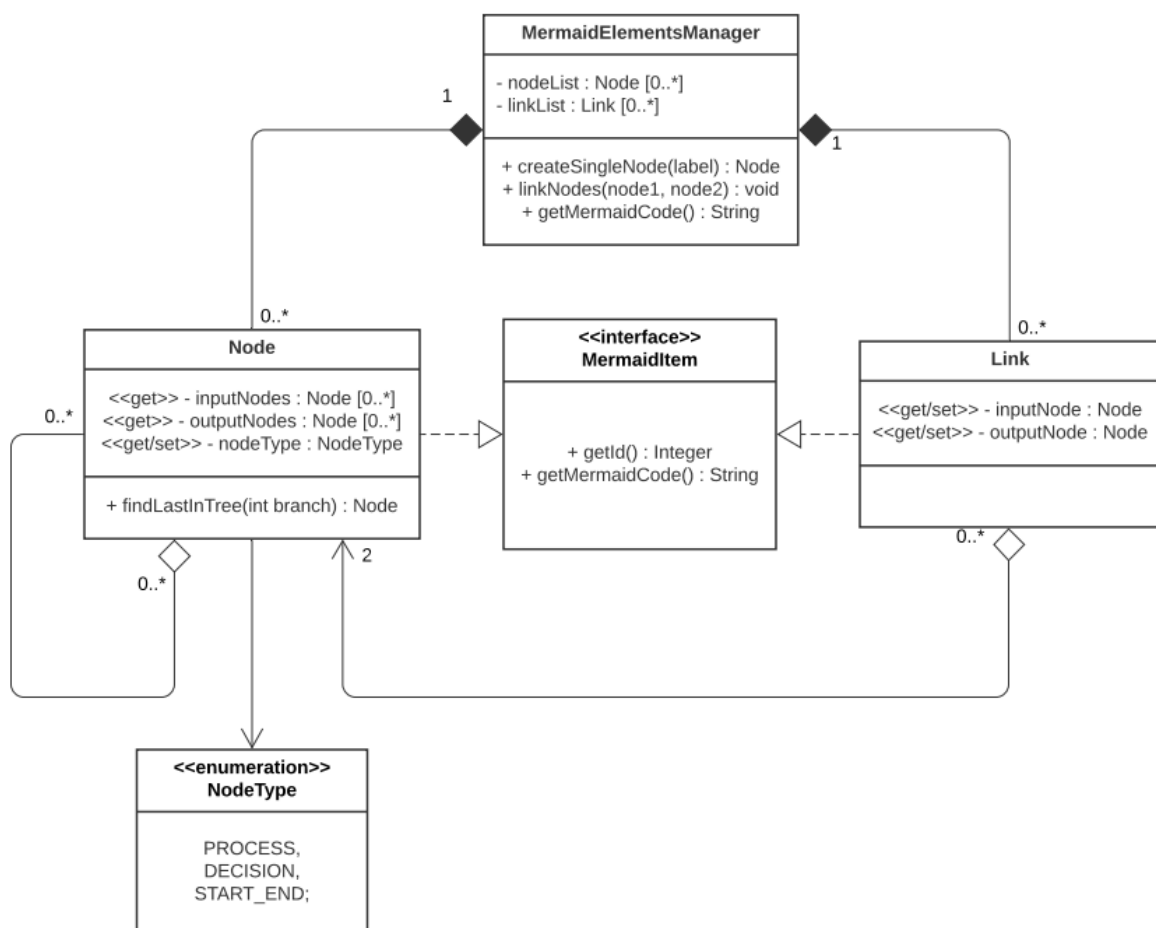
```

1  public void onIfStatementEnter(String condition, String expressions){
2      log.info("IF entered : " + condition + " than : " + expressions);
3  }
4
5  public void onIfStatementExit(){
6      log.info("IF exited");
7  }
8
9  public void onElseStatementEnter(String expressions){
10     log.info("ELSE entered : " + expressions);
11 }

```

Rysunek 3.7: Przykład domyślnych definicji metod obserwowanych dla instrukcji *if/else*

### 3.3 Akcelerator budowania kodu w języku wymaganym przez Mermaid



Rysunek 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 blo-

kowych, a następnie konwersja tych elementów w pamięci programu na kod docelowy, wymagany przez Mermaid. Docelowo składa się on z listy węzłów (nodes) odpowiadającym poszczególnym blozkom 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 wzorec 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.

```
1  @Override
2  public void onIfStatementEnter(String condition, String expressions){
3      condition = replaceChars(condition);
4      condition = wordWrap(condition, 12);
5      log.info("IF entered : " + condition + " than : " + expressions);
6      manager.setLastNode(fromNode);
7      NodeItem scopeNode = manager
8          .createDecisionNodeLinkedToLast(condition);
9      scopeNodes.push(scopeNode);
10     fromNode = scopeNode.getOutputs().get(0);
11 }
12
13 @Override
14 public void onIfStatementExit(){
15     log.info("IF exited");
16     manager.createSingleNode("#if_merge");
17     manager.linkNodes(scopeNodes.peek().findLastInTree(0),
18         manager.getLastNode());
19     manager.linkNodes(scopeNodes.peek().findLastInTree(1),
20         manager.getLastNode());
21     scopeNodes.pop();
22     fromNode = manager.getLastNode();
23 }
24
25 @Override
26 public void onElseStatementEnter(String expressions){
27     log.info("ELSE entered : " + expressions);
28     fromNode = scopeNodes.peek().getOutputs().get(1);
29 }
```

Rysunek 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 blok 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 blozka 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, zostaje zdjęty ze stosu.