[TKOM] LOGO

Adamski Maciej

Maj 2021

1 Wstęp/Opis

Celem projektu było napisanie aplikacji okienkowej, która będzie podobna do znanego programu *Logomocja*. Na ekranie mają poruszać się żółwie wykonujące określone przez użytkownika w formie kodu zadania. Względem tradycyjnego *Logo*, ta wersja działała na konstrukcjach obiektowych oraz równoległych.

2 Funkcjonalność

2.1 Tworzenie nowego żółwia

Tworzenie obiektu nowego żółwia, do którego później można się odwołać, a także modyfikować parametry. Obiekt złożony mający swoje metody, które można wywoływać na danym żółwiu. Podczas deklaracji, możemy określić pozycje żółwia w nawiasach kwadratowych za pomocą obiektów typu Point oraz wyrażeń, które da się ewaluować do wartości liczbowej.

```
1    Turtle zolw1;
2    Turtle zolw2(200, 300);
3    Turtle zolw3(punkt);
4    turtle zolw4(zolw3.pos.x, zolw3.pos.y);
```

2.2 Usuwanie żółwia

Usuwa obiekt żółwia z programu oraz z przestrzeni do rysowania.

```
1    delete zolw1;
2    delete zolw2;
```

2.3 Idź do przodu

Mówi żółwiowi, że ma poruszyć się do przodu o określoną ilość pikseli oraz zależnie od ustawień pędzla rysować pod sobą kreskę lub nie. Wymaga podania ilości kroków jako parametru.

```
1    zolw1.go(20); // 20 kroków do przodu
2    zolw2.go(500); // 500 kroków do przodu
```

2.4 Pozycja żółwia

Możliwość pobrania współrzędnych x, y aktualnej pozycji żółwia, a także ustawienie konkretnej pozycji.

```
1    zolw1.pos.x; // zwraca współrzędną X
2    zolw1.pos.y = 20; // ustawia‘ współrzędną Y na 20
```

2.5 Obrót

Mówi żółwiowi, że ma obrócić się o zadany kąt w lewo lub w prawo. Nie zmienia on swojej pozycji, a jedynie kierunek, w którym będzie się poruszał. Wymaga podania kąta, o który ma się obrócić żółw jako parametr.

```
1    zolw1.right(90); // obrót w prawo o 90 stoni
2    zolw2.right(-90); // obrót w prawo o -90 stopni, czyli w lewo o 90
3    zolw2.left(3 * 3 * 10); // obrót w lewo o 90 stopni
```

2.6 Kąt o jaki obrócony jest żółw

Możliwość pobrania wartości o jaki obrócony jest żółw względem początkowego ustawienia, a także ustawienie nowego bezwzględnego kierunku.

```
1    zolw1.direction; // zwraca kąt
2    zolw1.direction = 25 * 4; // ustawia kąt 100* w prawo względem
         początkowego
3    Integer dir = 200;
4    zolw2.direction = dir; // ustawia kąt zapisany w zmiennej dir
5    zolw2.direction = zolw1.direction; // ustawia ten sam kąt w żółwiu2,
         który jest aktualnie w żółwiu1
```

2.7 Czyszczenie całego ekranu

Funkcja czyści cały ekran - usuwa wszystkie narysowane linie. Nie zmienia pozycji żółwia.

```
clean();
```

2.8 Pędzel jako składowa żółwia

Pędzel jest osobną klasą, składową żółwia. Obiekt ten posiada odpowiednie parametry, do których dostęp jest przez operator wyłuskania, pozwalając na pobranie wartości lub ustawienie nowej. Cały obiekt pędzla można pobrać i np skopiować do innego żółwia.

```
zolw1.brush.size; // zwróci grubość pędzla
zolw1.brush.color; // zwróci klasę kolor
zolw1.brush.enable; // zwróci informację czy pędzel jest włączony czy wyłączony

zolw2.brush = zolw1.brush; // kopiuje wszystkie parametry żółwia1 do żółwia2
```

2.9 Zmiana grubości pędzla

Pozwala zmieniać grubość pędzla żółwia. Grubość pędzla definiuje wielkość linii rysowanej przez żółwia.

```
zolw1.brush.size; // zwraca wielkość pędzla
zolw1.brush.size = 10; // ustawia grubość pędzla na 10
zolw2.brush.size = zolw2.brush.size; // pędzel żółwia2 będzie takiej samej grubości jak żółwia1
zolw1.brush.size = 2 + 5; // wyrażenie ewaluowane do wartości liczbowej 7 i ustawiona grubość pędzla
```

2.10 Włączenie/wyłączenie pędzla

Pozwala włączyć lub wyłączyć pędzel, czyli zmienić czy podczas chodzenia żółw będzie rysował ślad czy nie.

```
zolw1.brush.enabled; // zwróci informację czy pędzel jest włączony
zolw1.brush.enabled = true; // żółw będzie rysować linię
zolw2.brush.enabled = zolw1.brush.enabled; // pędzel żółwia2 będzie miał taki sam stan jak pędzel żółwia1
zolw2.brush.enabled = 2 < 3; // wyrażenie zostanie ewaluowane do wartości logicznej true - i pędzel będzie widoczny
```

2.11 Zmiana koloru pędzla

Pozwala zmienić kolor pędzla danego żółwia na podany jako argument, w postaci stringa lub wyrażenia ewaluowanego do tej postaci. Metoda rozłoży ciąg znaków na trzy składowe.

```
zolw1.brush.color = "#000000"; // ustawia kolor pędzla na czarny
zolw2.brush.color = "#ffffff"; // ustawia kolor pędzla na biały
zolw2.brush.color = zolw1.brush.color; // ustawia kolor pędzla żół
    wia2 na taki sam jak żółwia1
Color kolor = "#123456";
zolw2.brush.color = kolor; // ustawia kolor pędzla na zapisany w
    zmiennej kolor
```

2.12 Pobranie koloru pędzla

Istnieje możliwość pobrania koloru pędzla w postaci stringa w formacie heksadecymalny. Zarówno jako cały kolor, jak i poszczególne składowe.

```
zolw1.brush.color.R; // string zawierający składową R
zolw1.brush.color.hex; // zwróci cały kolor w postaci heksadecymalnej
zolw1.brush.color; // to samo co color.hex (skrócony zapis)
```

2.13 Pokazanie/ukrycie żółwia

Pozwala zmieniać widoczność żółwia na planszy. Nie ma to wpływu na to czy żółw maluje linię, czy nie - nawet ukryty będzie rysował przy włączonym pędzlu.

```
zolw1.hidden; // zwróci informację czy żółw jest widoczny
zolw1.hidden = false; // żółw będzie widoczny
zolw2.hidden = zolw1.hidden; // widoczność żółwia2 będzie taka sama
    jak żółwia1
zolw1.hidden = 2 < 3; // Obliczona zostanie wartość logiczna
    wyrażenia 2<3 i ustawiona dla żółwia
```

2.14 Tworzenie zmiennych zawierających współrzedne x, y

Pozwala zapisać współrzedne [x, y] w jednej zmiennej typu Punkt. Współrzędne mogą być inicjalizowane podczas deklaracji lub potem za pomocą przypisania.

```
Point punkt1(10, 10); // zmienna x=10, y=10
Point punkt2; // domyślnie x=0, y=0
Point punkt3(punkt2); // inicjalizuje punkt3 wartościami
    współrzędnych punktu2
```

```
4     Point punkt4(zolw.pos); // inicjalizuje punkt4 wartościami
          współrzednych żółwia
5     punkt2.x; // pobierze współrzędną x punktu punkt2
6     punkt1.x = 50; // teraz x=50, y=20
7     punkt1 = punkt2; // kopiuje współrzędne punktu2 do punktu1
8     punkt2.y = zolw.pos.x; // ustawia wartość współrzędnej y punktu2, na
          wartość współrzędnej x żółwia
```

2.15 Przeniesienie żółwia w dane miejsce

Pozwala przenieść wybranego żółwia we wskazane miejsce. Miejsce może być wskazane przez zmienną typu Punkt lub wyrażenie, które da sie ewaluować do zmiennej typu Punkt lub wartości lczbowej

```
1     zolw1.moveTo(punkt1); // przenosi żółwia w miejsce wskazane przez
          zmienną typu Punkt
2     zolw2.moveTo(zolw.pos); // przenosi żółwia w miejsce wskazane przez
          obiekt Punkt w obiekcie żólwia
3     zolw3.moveTo(20, x + 2); // przenosi żółwia w miejsce o
          współrzędnych podanych jako wyrażenia
```

2.16 Przeniesienie wszystkich żółwi na punkt startowy

Przenosi wszystkie żółwie na środek planszy oraz ustawia domyslny kierunek ich poruszania.

```
1     allToStart();
```

2.17 Operacje matematyczne w wyrażeniach

Pozwala na użycie operatorów matematycznych w wyrażeniach czy w wyołaniach funkcji.

```
1     zolw1.go(2 * 2);
2     zolw2.right(90 + 90);
3     repeat(20 - 15) {}
4     if ((x/2) == 2) {}
```

2.18 Instrukcje warunkowe

Typowa konstrukcja warunkowa znana z różnych języków programowania. Pozwala zdefiniować szereg instrukcji, które wykonają się tylko w przypadku wystąpienia określonego warunku. Możliwe też jest dodanie bloku typu *else*, wykonywanego w przypadku wartości logicznej *false*.

```
if (x < 20) { // jeśli zmienna jest mniejsza od 20 to...
    zolw1.go(x);
} else { // w pozostalych przypadkach
    zolw.go(x - 20);
}
```

2.19 Tworzenie funkcji

Pozwala definiować własne funkcje z określonymi parametrami i zwracanym typem. Funkcja może przyjmować dowolną ilość parametrów oraz może zwracać wskazany typ lub żadnego. Do zdefiniowanych funkcji można się później odwołać.

```
function square(Turtle zolw) // definicja nowej funkcji square z
    parametrem
{
    ...
}
square(zolw1); // wywołanie nowej funkcji square

Integer function retInt() // definicja nowej funkcji retInt bez
    parametrów, zwracająca typ Integer
{
    return 1;
}

Integer testInt = retInt(); // testInt ma wartość 1
```

2.20 Pętla

Konstruckja pozwalająca na wykonanie szeregu instrukcji wyznaczoną ilość razy. Należy podać ile razy pętla ma się wykonać, a także w bloku zdefiniować instrukcje, które mają się wykonać.

```
repeat(4) { // powtórz 10 razy kod w nawiasach klamrowych
    zolw1.go(100);
    zolw1.right(90);
}
```

2.21 Zadania cykliczne

Pozwala na dodanie zadania cyklicznego wykonywanego co określoną ilość milisekund. Dodatkowo można ustalić ile razy ma się wykonać - jest to parametr opcjonalny (maksymalna ilość iteracji to 10 - wartość domyślna).

```
    repeatTime(1000, 10) { // Zadanie wykona się 10 razy co sekundę
        zolw1.go(20);
    }
    repeatTime(1000) { // To samo zadanie, ale nie mające określonej
        ilości wykonań
        zolw1.go(20);
    }
```

2.22 Pętla warunkowa

Konsturkcja pętli warunkowej pozwala na powtórne wykonywanie danego fragmentu kodu dopóki spełnionyj jest jakiś warunek logiczny.

```
    Integer x = 10;
    repeatCondition(x > 5) // Będzie się wykonywać dopóki zmienna x
        będzie miała wartość większą niż 5
    {
        [...]
        x = x - 5
    }
```

2.23 Funkcje rekurencyjne

Program pozwala na użycie konstrukcji funkcji rekurencyjnych - takich, które wywołają samego siebie.

```
Integer function fibonacciSum(Integer n)
{
   if (n < 2)
   {
      return n;
   }
   return fibonacciSum(n - 1) + fibonacciSum(n - 2);
}
```

3 Gramatyka

Gramatyka języka używanego w tym projekcie, wyrażona w notacji *EBNF*.

```
    program = { instruction | functionDef };
    instruction = functionCall | if | repeat | repeatTime |
        repeatCondition | varDef | varAssignment;

```

```
4
5
6   (* konstrukcje *)
7       repeat = "repeat", "(", expression, ")", block;
8       repeatTime = "repeatTime", "(", expression, [",", expression ], ")",
            block;
9       repeatCondition = "repeatCondition", "(", condition, ")", block;
10      if = "if", "(", condition, ")", block, ["else", block];
11      delete = "delete", id;
12
13      block = "{", { instruction }, [ return ]"}";
14
15
16
17  (* funkcje *)
18      parameters = parameter, { ",", parameter };
19      parameter = varDec;
20
21      arguments = argument, { "," argument };
22      argument = expression;
23
24      functionCall = anyMemberLevel, "(", [ arguments ], ")",
            endInstruction;
25      functionDef = [allTypes], "function", id, "(", [ parameters ], ")",
            block;
26
27      return = "return", expression, endInstruction;
28
29      expression = condition;
30
31      conditionExpression = andCondition, { orOperator, andCondition };
32      andConditionExpression = relationCondition, { andOperator,
            relationCondition };
33      relationConditionExpression = [ notOperator ], (( expression, [
            relationOperator, expression ] ) | booleanWord )
34
35
36      arithmeticAddExpression = term, { addOperator, term };
37      arithmeticMutliExpression = factor, { multiOperator, factor };
38
39      factorExpression = [ "-" ], ( ( "(", expression, ")" ) |
            anyMemberLevel | int | functionCall );
40
41
42
43  (* operatory *)
44      orOperator = "||" | "or";
45      andOperator = "&&" | "and";
46
47      equalOperator = "==" | "!=";
```

```
    relationOperator = "<" | ">" | "<=" | ">=" | equalOperator ;

    addOperator = "+" | "-";
    multiOperator = "*" | "/";

    assignOperator = "=";

    notOperator = "!";

    accessOperator = ".";



(* znaki *)
    digitWithoutZero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
        "9";
    digitZero = "0";
    digit = digitZero | digitWithoutZero;

    lowerCase = "a"..."b";
    upperCase = "A"..."B";
    letter = lowerCase | upperCase;

    symbol = "!" | "@" | "#" | "$" | "%" | "^" | "&" | "*" | "(" | ")" |
        "-" | "=" | "_" | "+" | "{" | "}" | "[" | "]" | ":" | ";" | "'"
        | "," | "." | "/" | "<" | ">" | "?" | "`" | "~";

    alphanumeric = letter | digit | symbol;

    booleanWord = "true" | "false";



(* zmienne *)
    anyMemberLevel = id, { ".", id };

    id = letter, {letter | digit | "_"};
    varDec = allTypes, id;
    varDef = varDec, [ classAssign | simpleAssign ];
    assign = anyMemberLevel, simpleAssign;
    classAssign = "(", [ expression, { ",", expression } ], ")";
    simpleAssign = assignOperator, ( expression | string | booleanWord );


    allTypes = turtleType | pointType | stringType | intType | boolType;
    turtleType = "Turtle";
    pointType = "Point";
    stringType = "String";
    boolType = "Boolean";
    intType = "Integer";
```

```
95
96      string = '"', { alphanumeric | " "}, '"';
97
98      int = digit, {digit};
99
100
101
102 (* inne *)
103     comment = "//", {alphanumeric | " "};
104     endInstruction = ";";
```

4 Problemy i decyzje z nimi związane

4.1 Brak wyróżnionej funkcji startowej

Program nie posiada wyróżnionej funkcji startowej pokroju *main* czy *init*. Cały kod jest wykonywany po kolei.
Wiąże się to z kolejnymi dwoma decyzjami:

- Brak definicji funkcji przed jej wywołaniem (Patrz Punkt 4.2)
- Brak definicji zmiennej przed jej użyciem (Patrz Punkt 4.3)

4.2 Brak definicji funkcji przed jej wywołaniem

Funkcja nie musi być zdefiniowana przed jej wywołaniem. Kod jest analizowany w całości przed jego wykonaniem w poszukiwaniu definicji funkcji. Dzięki temu funkcje mogą być zdefiniowane nawet na końcu pliku oraz można korzystać z rekurencji.

```
1     Poprawne konstrukcje:
2         function xx() {};
3         xx();
4
5         yy();
6         function yy() {};
```

4.3 Brak definicji zmiennej przed jej użyciem

Nie można się odwołać do zmiennej, która nie została wcześniej zdefiniowana. W innym przypadku zostanie zwrócony błąd.

```
1     Poprawne konstrukcje:
2         Integer xx = 20;
3         use(xx);
```

```
4
5       Niepoprawne konstrukcje:
6           use(xx);
7           Integer xx = 20; // lub całkowity brak tej linii
```

4.4 Brak rzutowania Integer na Boolean

W programie nie występuje rzutowanie typu **Integer** na typ **Boolean**. W przypadku użycia wyrażeń matematycznych i liczbowych należy użyć dowolnego operatora relacyjnego, żeby takie wartości można przypisać do typu **Boolean**.

```
1       Poprawne konstrukcje:
2           Boolean x = true;
3           Boolean y = 2 < 3;
4           Boolean z = true && false;
5           Integer integer = 2;
6           Boolean w = integer == 2;
7           itd.
8
9       Niepoprawne konstrukcje
10          Boolean x = 2;
11          Integer integer = 2;
12          Boolean y = integer;
```

5 Tokeny

W projekcie, na poziomie analizy leksykalnej rozróżniane są następujące tokeny:

- **Repeat**
- **RepeatTime**
- **RepeatCondition**
- **Delete**
- **Function**
- **If**
- **Else**
- **Return**
- **Identifier**
- **CurlyBracketOpen {**, **CurlyBracketClose }**

- **RoundBracketOpen** (, **RoundBracketClose**)
- **Plus** +, **Minus -**, **Multiply ***, **Divide** /
- **Comma ,**
- **Dot .**
- **Semicolon ;**
- **Equal** ==, **NotEqual** !=, **Less** <, **Greater** >, **LessOrEqual** <=, **GreaterOrEqual** >=
- **AssignOperator** =, **NotOperator !**
- **And** &&, **Or** ||
- **True**, **False**
- **Integer**, **Turtle**, **Point**, **Boolean**, **ColorVar**
- **Digit**, **ColorValue**
- **UNKNONW**, **INVALID**
- **EndOfFile**

6 Dostępne typy zmiennych

W programie dostępne są następujące typy zmiennych:

- **Turtle** - klasa reprezentująca żółwia
- **Point** - klasa reprezentująca punkt kartezjański na planszy
- **Color** - zawierająca ciąg znaków reprezentujący kolor w zapisie hexadecymalnym
- **Integer** - liczba całkowita
- **Boolean** - typ logiczny *true/false*

7 Obsługa błędów

Każdy moduł podczas swojego działania i przetwarzania programu wyszukuje odpowiednie błędy. Gdy jakikolwiek błąd zostanie znaleziony, wysyła on informację do **Loggera**, żeby stworzył odpowiedni *log*.
Logi dzielą się na dwie grupy:

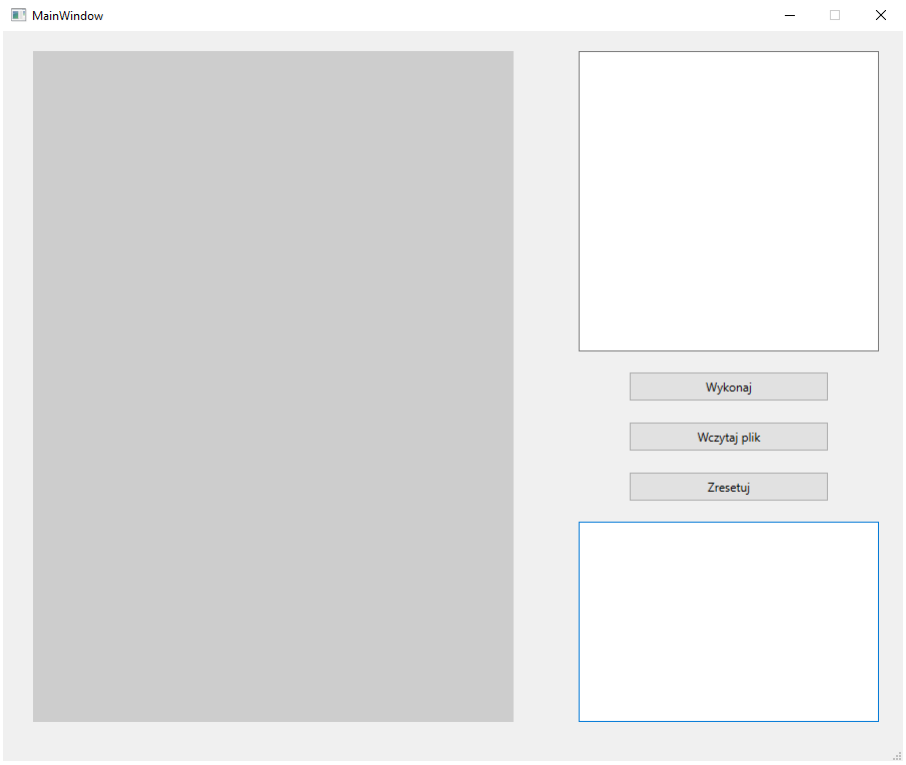
- **Błędy krytyczne** - które przerywaja natychmiastowo działanie programu
- **Ostrzeżenia** - nie przerywają działania programu, są informacją dla użytkownika, że należy coś poprawić w kodzie, ale nie sprawia to, że kod nie jest niemożliwy do wykonania

Komunikaty zawierają takie informacje jak linia, w której wystąpił oraz prawdopdobną przyczynę.
Są one wyświeltane w odpowiednim polu GUI.

8 Interfejs graficzny

GUI zostało stworzone z użyciem *QT*. Okno aplikacji podzielone jest na następujące części

- Okno rysowania, w którym będą wyświetlane żółwie oraz to co narysują
- Pole tekstowe, w którym będziemy wpisywać kod, który ma się wykonać
- Pole tekstowe, w którym będą wyświetlane wyniki przetwarzania kodu
- Przycisk *Wykonaj*, który uruchamia wykonywanie kodu podanego w polu tekstowym wyżej
- Przycisk *Wczytaj plik*, który pozwala wskazać plik z zapisanym kodem do wykonania
- Przycisk *Zresetuj*, który resetuje program - czyści tablicę, usuwa wszystkie zmienne i zdefiniowane funkcje



Rysunek 1: Interfejs graficzny

9 Sposób uruchomienia, wej./wyj.

Aplikacja uruchomiana jest w sposób dostarczony przez środowisko QT. Natomiast kod, może być wprowadzony do programu w następujący sposób:

- Przez pole tekstowe w oknie aplikacji
- Wcztytanie z pliku

Wyjście aplikacji zrealizowane jest za pomocą pola tekstowego w oknie aplikacji, na którym program będzie mógł wypisywać informacje.

10 Moduły i klasy

10.1 Lexer - analiza leksykalna

Moduł odpowiadający za rozbicie podanego kodu na kolejne elementarne części - tokeny (lista tokenów, patrz Punkt 5). Lexer odczytuje wejście znak po zaaku, próbując dopasować sekwencje do jednego ze zdefiniowanych tokenów. Moduł ten wystawiaa główną metodę *getNextToken()*, służącą do pobrania następnego dopasowanego tokena.
Lexer tworzy również kilka błędów, które mogą być wykryte już na tym etapie - jak brakujący znak czy niepoprawna sekwencja liczb.

10.2 Parser - analiza składniowa

Parser ma za zadanie sprawdzić czy kolejność odczytanych tokenów pokrywa się ze zdefiniowaną gramatyką języka. Pobiera on z *Lexera* kolejne tokeny i próbuje je stopniowo dopasować do odpowiednich konstrukcji. Gdy uda się dopasować jakąś konstrukcję, tworzone są kolejne węzły drzewa składniowego AST.
Na tym etapie wyłapywane są błędy związane z niepoprawnymi kombinacjami tokenów, błędnymi konstrukcjami względem gramatyki oraz brakujące tokeny.

10.3 Interpreter

Interpreter odpowiada za wykonanie instrukcji zawartych w drzewie składniowym. Na początku przechodzi on strukturę w poszukiwaniu definicji funkcji, a następnie przeszukując drzewo w głąb wykonując polecenia.
Moduł ten wyłapuje błędy związane ze zmiennymi czy niepoprawnymi przypisaniami wartości.

10.4 Kontekst

Klasa przenoszona jako referencja przez interpreter wgłąb drzewa składniowoweg. Zawiera aktualnie zdefiniowane zmienne, funkcje, stan obliczonych wartości, czy wartości zwracane przez funkcje. Odpowiada za prawidłowe przechowywanie wszystkich danych pomiędzy poszczególnymi węzłami drzewa. Tworzy "świat/przestrzeń", w którym wykonuje się kod.

10.5 Rysowanie

Rysowanie odbywa się w przestrzeni *QWidget* środowiska *QT*. Z widgetami komunikują się wykonywane instrukcje za pomoca `Kontekstu`

11 Testowanie

11.1 Testy jednostkowe

Poszczególne konstrukcje i moduły będą sprawdzane za pomocą testów jednostkowych pozwalające przetestować elementarne funkcjonalności.
Testy te zostaną prawdopodobnie zrealizowane w bibliotece *Catch2*, ale może to ulec zmianie.

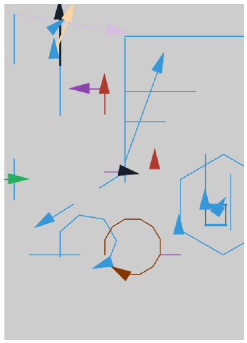
11.2 Testy integracyjne

Testy integracyjne będą przeprowadzane automatycznie. Zostanie przygotowanych kilka scenariuszy testujących wiele warstw kodu i różnych komponentów celem sprawdzenia ich współnego działania.
Sceneriusze będą napisane prawodopodbnie w tej samej bibliotece co testy jednostkowe.

11.3 Testy GUI

Działanie graficznego interfesju użytkownika będzie testowane manualnie.

12 Prezentacja działania



Rysunek 2: Wynik wykonania przykładowego kodu

Kod użyty do wygenerowania takiego wyniku:

```
1 Boolean test = true;
2 test = false;
3
4 Integer test2 = 20;
5 test2 = 30;
6
7 Integer x = 200;
```

```
Integer y = 220;
Turtle zolw1(x,y);
zolw1.brush.color = "#B03A2E";
Turtle zolw11(x + 100, y+100);
zolw11.brush.color = zolw1.brush.color;

Point point1(300, 350);
point1.x = 20;
Turtle zolw2(point1);
zolw2.brush.color = "#27AE60";
Turtle zolw22(point1.x, point1.y + 40);
zolw22.go(80);
Boolean hide = true;
zolw22.hidden = hide;
Point point2(point1.x, 400);
Turtle zolw3;
Color colorZolw3 = "#17202A"
zolw3.brush.size = 4;
zolw3.brush.color = colorZolw3;

zolw1.go(50);
zolw2.right(90);
zolw3.direction = 100;

Turtle zolw4(100, 100);
Turtle zolw5(zolw4.pos.x, 50);
zolw5.right(50);

Turtle zolwRepeatUnder(400, 400);
Turtle zolwRepeat(zolwRepeatUnder.pos);

zolwRepeatUnder.brush.color = "#5F6A6A";

zolwRepeat.brush.size = 5;

Integer doBrush = 1;
Boolean doBrushFlag = true;

repeat(4) {
   if (doBrush == 1)
   {
      doBrushFlag = true;
      doBrush = doBrush - 1;
   }
   else
   {
      doBrushFlag = false;
      doBrush = doBrush + 1;
   }

```

```
   zolwRepeatUnder.right(90);
   zolwRepeatUnder.go(40);

   zolwRepeat.brush.enabled = doBrushFlag;
   zolwRepeat.right(90);
   zolwRepeat.go(40);


   Turtle testinRepeat(20, 20);
   testinRepeat.direction = 180;
   testinRepeat.go(100);
}


Integer funcInt = 200;
Boolean doBrushFunc = false;
testFunc(x + 50, doBrushFunc);

funcInt = 120;
doBrushFunc = false;
testFunc(funcInt + 20, doBrushFunc);

funcInt = 70;
doBrushFunc = true;
testFunc(funcInt + 10, doBrushFunc);


function testFunc(Integer testInt, Boolean doBrush) // first parse all
    func def
{
   Turtle zolw;
   zolw.go(20);
   zolw.brush.enabled = doBrush;
   zolw.go(testInt);
   zolw.brush.enabled = true;
   zolw.right(90);
   zolw.go(testInt);
}

Turtle zolwMoveTo;
zolwMoveTo.brush.color = "#8E44AD";
zolwMoveTo.left(90);
zolwMoveTo.go(40);
zolwMoveTo.moveTo(350, 500);
zolwMoveTo.go(40);
zolwMoveTo.moveTo(point1);
zolwMoveTo.go(40);
zolwMoveTo.moveTo(zolw1);
zolwMoveTo.go(40);

```

```
107 Point toAssignPoint(111, 222);
108 Point testAssignPoint;
109 testAssignPoint = toAssignPoint;
110 Turtle turtlecos(testAssignPoint);
111 turtlecos.go(100);
112 turtlecos.brush = zolw3.brush;
113 turtlecos.go(100);
114
115 Turtle showPos(425, 411);
116 showPos.right(34);
117 Turtle showPosAssign;
118 showPosAssign.pos = showPos.pos;
119 showPosAssign.direction = showPos.direction;
120 showPos.direction = 300;
121
122
123 Color colorToAssign = "#D7BDE2";
124 Color col = colorToAssign;
125 Turtle turtle(20, 20);
126 turtle.right(100);
127 turtle.brush.color = col;
128 turtle.go(200);
129
130 function test(Turtle testTurtle)
131 {
132    testTurtle.go(200);
133 }
134
135 Turtle testTurtle;
136 testTurtle.go(20);
137 testTurtle.right(20);
138 test(testTurtle);
139
140 function testFuncWithMathOp(Integer intTest)
141 {
142    Turtle testTurtle;
143    testTurtle.go(intTest);
144 }
145
146 testFuncWithMathOp(2 * 2 + 2 - 2 * 30 / 2 - 10 + 300);
147
148 Turtle toAssign1(100, 100);
149 toAssign1.brush.color = "#FAD7A0";
150 toAssign1.brush.size = 4;
151 toAssign1.direction = 22;
152
153 Turtle toAssign2;
154 toAssign2 = toAssign1;
155 toAssign2.go(80);
156
```

```
157
158
159 Integer function testReturnInt()
160 {
161    Turtle zolw;
162    zolw.go(200);
163
164    return 100;
165
166    Turtle zolw2;
167    zolw2.right(810);
168    zolw2.go(200);
169 }
170
171
172 function testUseReturnedIntAsParameter(Integer testInt)
173 {
174    Turtle zolw(testInt * 4, testInt * 4);
175    zolw.go(testInt);
176 }
177
178 Integer intFromReturnValue = testReturnInt();
179 Turtle zolw(intFromReturnValue, intFromReturnValue);
180
181 testUseReturnedIntAsParameter(testReturnInt());
182
183 Integer toDelete = 200;
184 delete toDelete;
185
186 Integer toDelete = 450;
187 Turtle zolwWithDeleteInt(toDelete, toDelete);
188 zolwWithDeleteInt.go(111);
189 delete zolwWithDeleteInt;
190
191 Turtle zolwInRepeatTime(350, toDelete);
192
193 repeatTime(450, 6)
194 {
195    zolwInRepeatTime.go(100);
196    zolwInRepeatTime.right(60);
197    Turtle new;
198    new.direction = 180;
199    new.go(20);
200 }
201
202 Integer repeatConditionInteger = 20;
203 Turtle repeatConditionTurtle(111, 504);
204
205 repeatCondition(repeatConditionInteger > 15)
206 {
```

```
   repeatConditionTurtle.go(50);
   repeatConditionTurtle.right(50);
   repeatConditionInteger = repeatConditionInteger - 1;
}


function booleanInArgs(Boolean testBool)
{
   if (testBool)
   {
      Turtle booleanInArgsTurtle;
      booleanInArgsTurtle.go(200);
   }
}

test(2 + 2 > 3);

Boolean function booleanReturnWithMath()
{
   return 2 + 2 * 2 < 3 / 3 * 6 + 1; // 6 < 7
}

Turtle booleanReturnWithMathTurtle;
booleanReturnWithMathTurtle.left(90 + 33);
booleanReturnWithMathTurtle.go(60);
booleanReturnWithMathTurtle.brush.enabled = !booleanReturnWithMath();
booleanReturnWithMathTurtle.go(60);
booleanReturnWithMathTurtle.brush.enabled = true;
booleanReturnWithMathTurtle.go(60);

Integer function fibonacciSum(Integer n)
{
   if (n < 2)
   {
      return n;
   }
   return fibonacciSum(n - 1) + fibonacciSum(n - 2);
}

Integer fibonacci9 = fibonacciSum(9)

if (fibonacci9 == 34)
{
   Turtle fibonacciTurtle(50, 500);
   fibonacciTurtle.direction = 90;
   fibonacciTurtle.go(100);
}
```

```
257 Turtle synchronizationInRepeatTimeTurtle(200, 500);
258 synchronizationInRepeatTimeTurtle.brush.color = "#873600";
259 Integer synchronizationInRepeatTimeCondition = 20;
260 Integer synchronizationInRepeatTimeCounter = 0;
261
262 repeatTime(1000, 7)
263 {
264    synchronizationInRepeatTimeCounter =
          synchronizationInRepeatTimeCounter + 1;
265    if (synchronizationInRepeatTimeCounter >= 5)
266    {
267       synchronizationInRepeatTimeCondition = 10;
268    }
269 }
270
271 repeatTime(2000, 10)
272 {
273    repeatCondition (synchronizationInRepeatTimeCondition > 10)
274    {
275    }
276    synchronizationInRepeatTimeTurtle.go(30);
277    synchronizationInRepeatTimeTurtle.right(30);
278 }
```