

Politechnika Świętokrzyska Wydział Elektrotechniki, Automatyki i Informatyki	
<i>Technologie Obiektowe – Projekt</i>	
DOKUMENTACJA	
Temat: Graficzne programowanie	Wykonał: Zbigniew Bielecki 1ID21A

Spis treści

1. Technologie i krótki ich opis.....	2
1.1. Silnik Godot.....	2
1.2. Język C#.....	2
1.3. Język GDScript.....	2
2. Funkcjonalności.....	2
3. Algorytmy konwersji – schemat blokowy.....	5
3.1. Opis wzoru tworzonych klas.....	5
3.1.1. Dla klas C#.....	6
3.1.2. Dla klas GDScript.....	6
3.2. Schemat dla języka C#.....	7
3.3. Schemat dla języka GDScript.....	8
4. Ważniejsze fragmenty kodu.....	8
5. Przykład użycia.....	10
5.1. Dla języka C#.....	11
5.2. Dla GDScript.....	12
6. Wnioski.....	14

1. Technologie i krótki ich opis

1.1. Silnik Godot

Godot Engine jest silnikiem do tworzenia gier 2D oraz 3D na licencji MIT zyskujący coraz większą popularność na rynku gier z powodu swojej lekkości oraz szybkości działania. Kod źródłowy silnika jest publicznie udostępniony na Githubie. Edytor używany do tworzenia gier został napisany w tym samym silniku, dlatego pozwala to na tworzenie pluginów w szybki sposób.

Oficjalna strona silnika:

<https://godotengine.org/>

1.2. Język C#

C# jest obiektowym językiem programowania, ogólnego przeznaczenia, stworzonym przez firmę Microsoft. C# jest ściśle związany z środowiskiem .NET, dlatego też kod napisany w tym języku może być uruchamiany tylko na systemie, które to środowisko wspiera.

1.3. Język GDScript

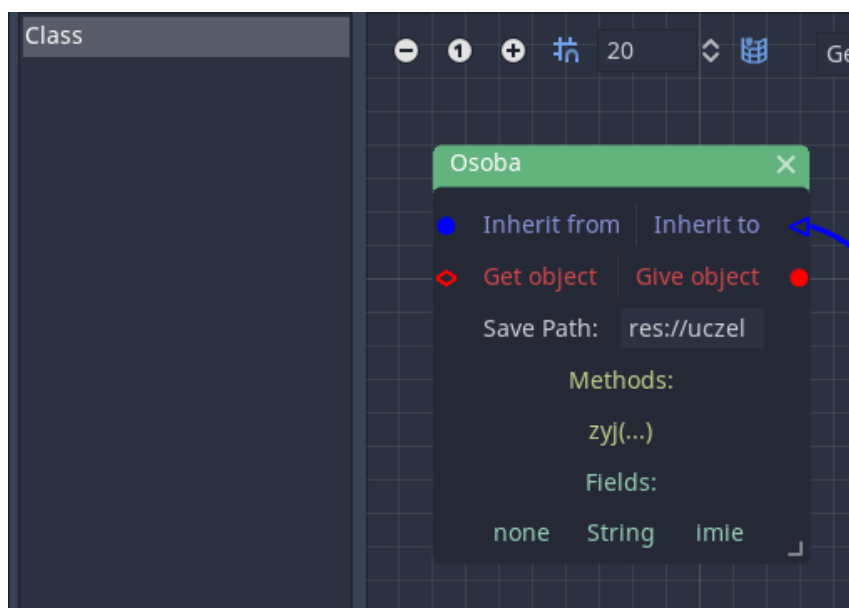
GDScript jest wysokopoziomowym, dynamicznie typowanym językiem programowania specjalnie zaprojektowanym do tworzenia gier ze współpracą z silnikiem Godot.

2. Funkcjonalności

Stworzony plugin pozwala na:

a) graficzne tworzenie klas przez drag & drop :

Plugin pozwoli na przeciąganie wzorów klas na graficzne pole, które pozwoli na ułożenie utworzonych przez siebie klas za pomocą przeciągania. W prawym dolnym rogu ekranu znajduje się mini-mapka pozwalająca na szybką nawigację po polu.



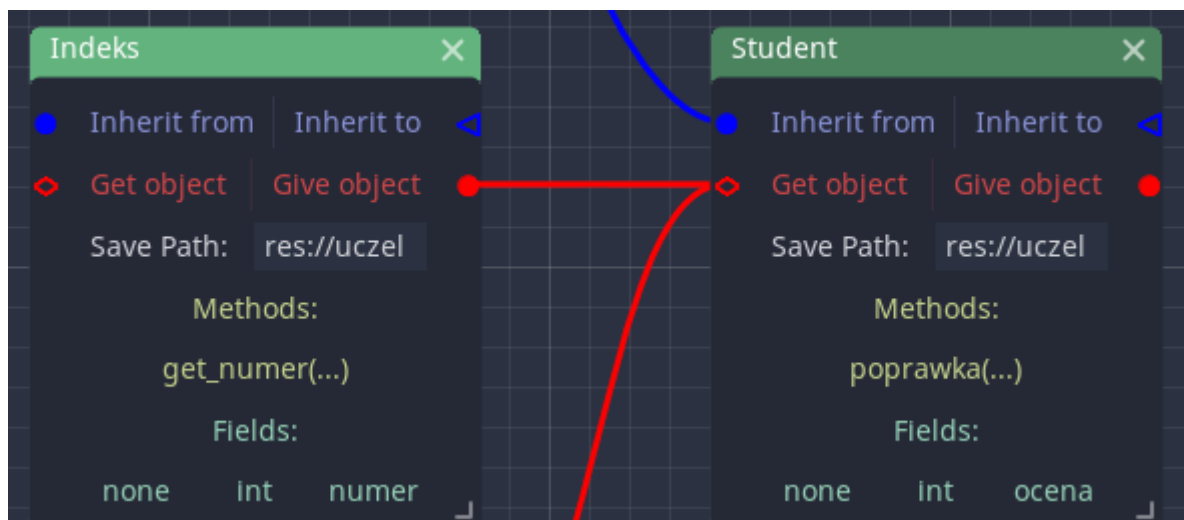
b) dziedziczenie:

Węzły klas posiadają specjalne pole pozwalające na tworzenia w szybki sposób dziedziczenia pomiędzy klasami. Jeśli dziedziczenie nie występuje to klasa będzie dziedziczyć po klasie Node. Jest to podstawowa klasa, dzięki której silnik będzie mógł bezproblemowo używać stworzone klasy.



c) agregacja:

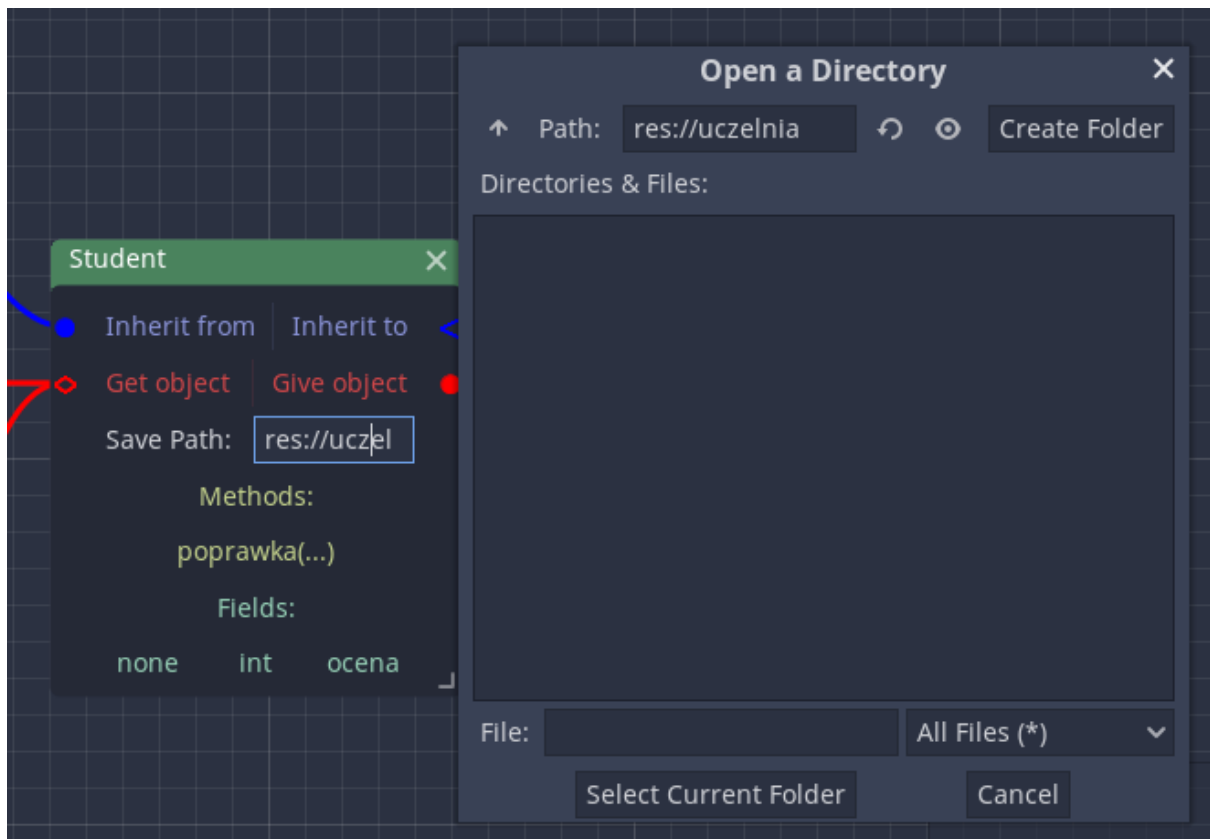
Program pozwala na użycie agregacji w schemacie. Każda klasa jest w stanie posiadać obiekty wielu innych klas.



d) wybranie miejsca zapisu każdej klasy:

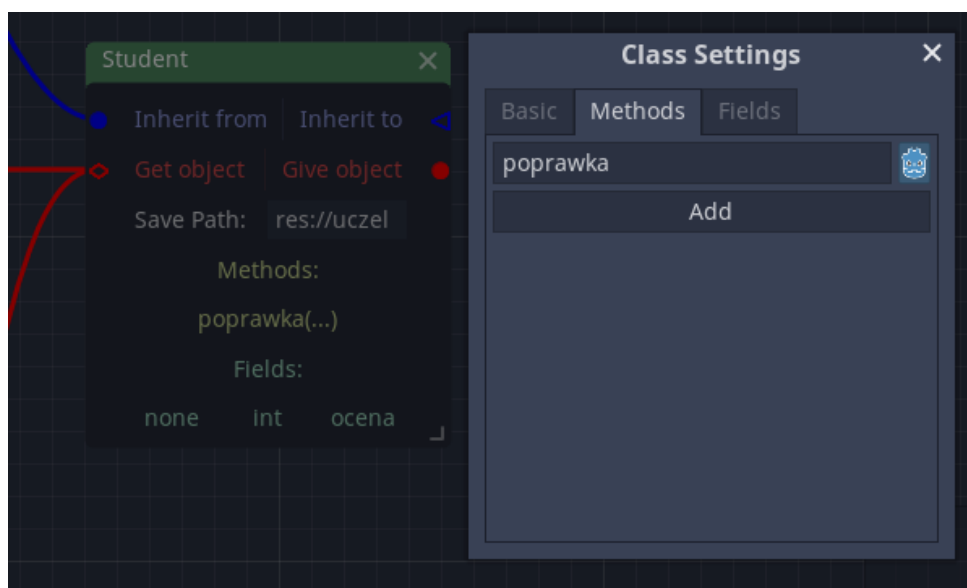
Dla każdej klasy będzie można wybrać miejsce jej zapisu. Musi to być folder, które znajduje się w zasobach konkretnie otwartego projektu (jest to folder zawierający plik godot.project).

Jeśli folder nie zostanie sprecyzowany to domyślnym miejscem zapisu będzie główny folder zasobów projektu.



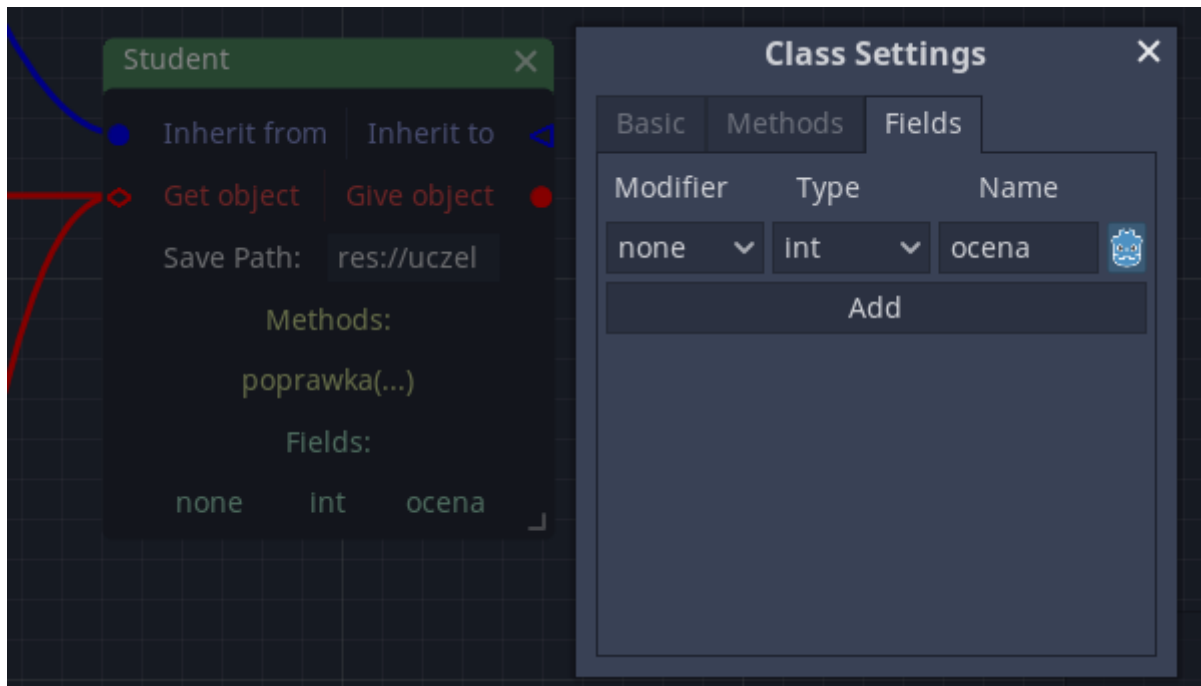
e) tworzenie metod:

Możliwe będzie utworzenie metod, które zostaną wygenerowane. Można ustalić im nazwy, oraz jeśli będzie potrzeba usunąć.



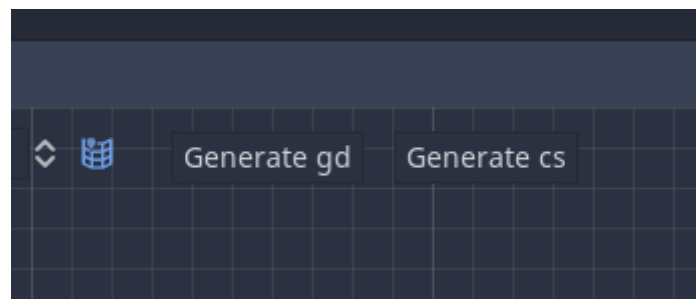
f) tworzenie pól:

W programie występuje możliwość dodania pola, dla danego pola ustawić można typ, nazwę oraz modyfikator.



g) konwersja grafu na język C# i GDScript:

Z stworzonego schematu istnieje możliwość generowania plików dla języka C# oraz GDScript



3. Algorytmy konwersji – schemat blokowy

3.1. Opis wzoru tworzonych klas

Opis użytych zmiennych:

- <CLASS NAME>: nazwa klasy;
- <PARENT>: nazwa klasy po której dziedziczy klasa;
- <F MODIFIER>: modyfikator pola;
- <F TYPE>: typ pola;
- <F NAME>: nazwa pola;
- <METHOD NAME>: nazwa metody;

3.1.1. Dla klas C#:

```
using Godot;
using System;

public class <CLASS NAME> : <PARENT>
{
    // Fields
    <F_MODIFIER> <F_TYPE> <F_NAME>;
    <F_MODIFIER> <F_TYPE> <F_NAME>;
    <F_MODIFIER> <F_TYPE> <F_NAME>;
    ...

    // Methods
    void <METHOD NAME>() {
        return
    }

    void <METHOD NAME>() {
        return
    }

    void <METHOD NAME>() {
        return
    }
    ...
}
```

3.1.2. Dla klas GDScript:

```
extends <PARENT>

class_name <CLASS NAME>

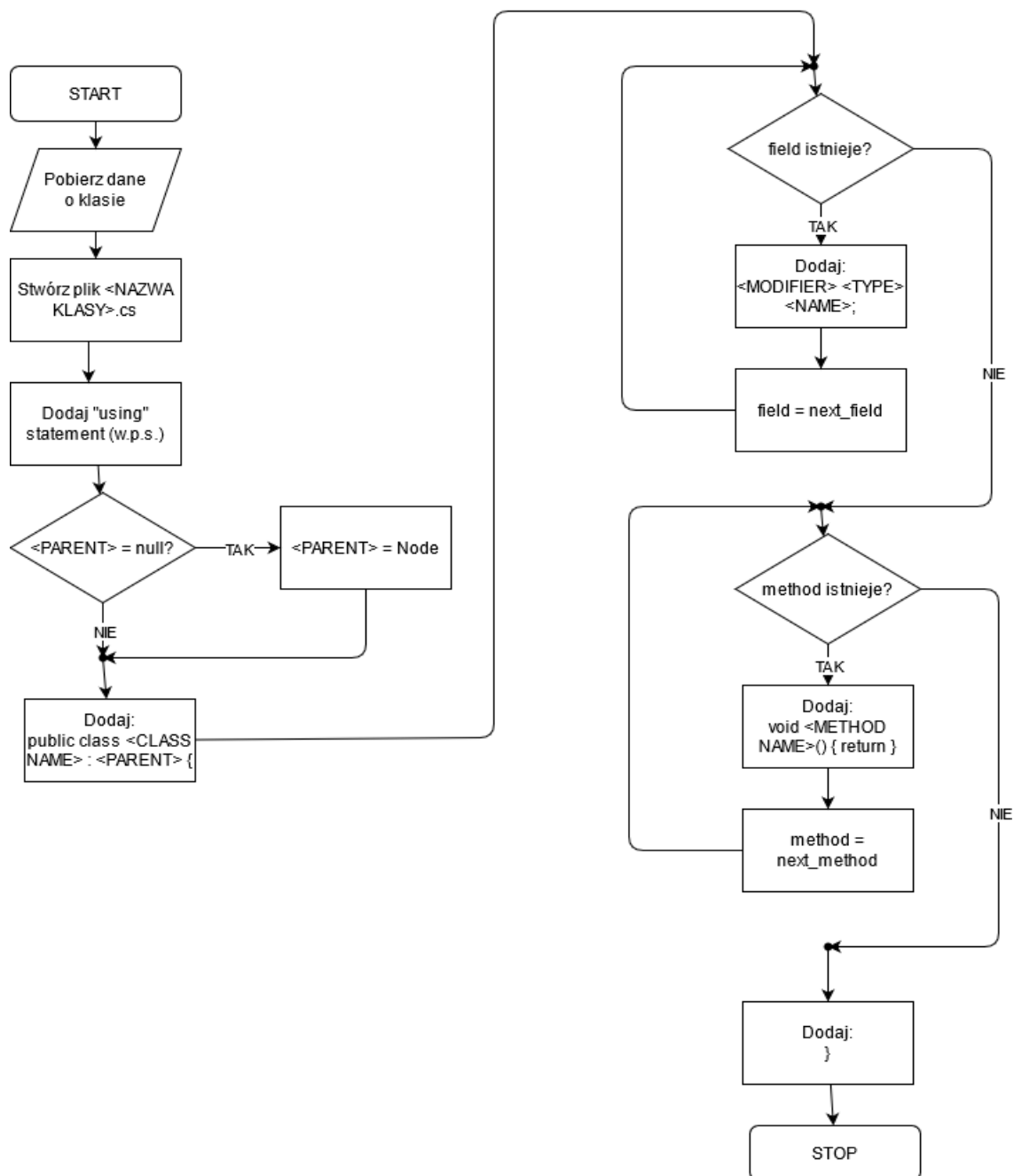
# Fields
<F MODIFIER> <F TYPE> <F NAME>
<F MODIFIER> <F TYPE> <F NAME>
<F MODIFIER> <F TYPE> <F NAME>
...

# Methods
func <METHOD NAME>():
    return

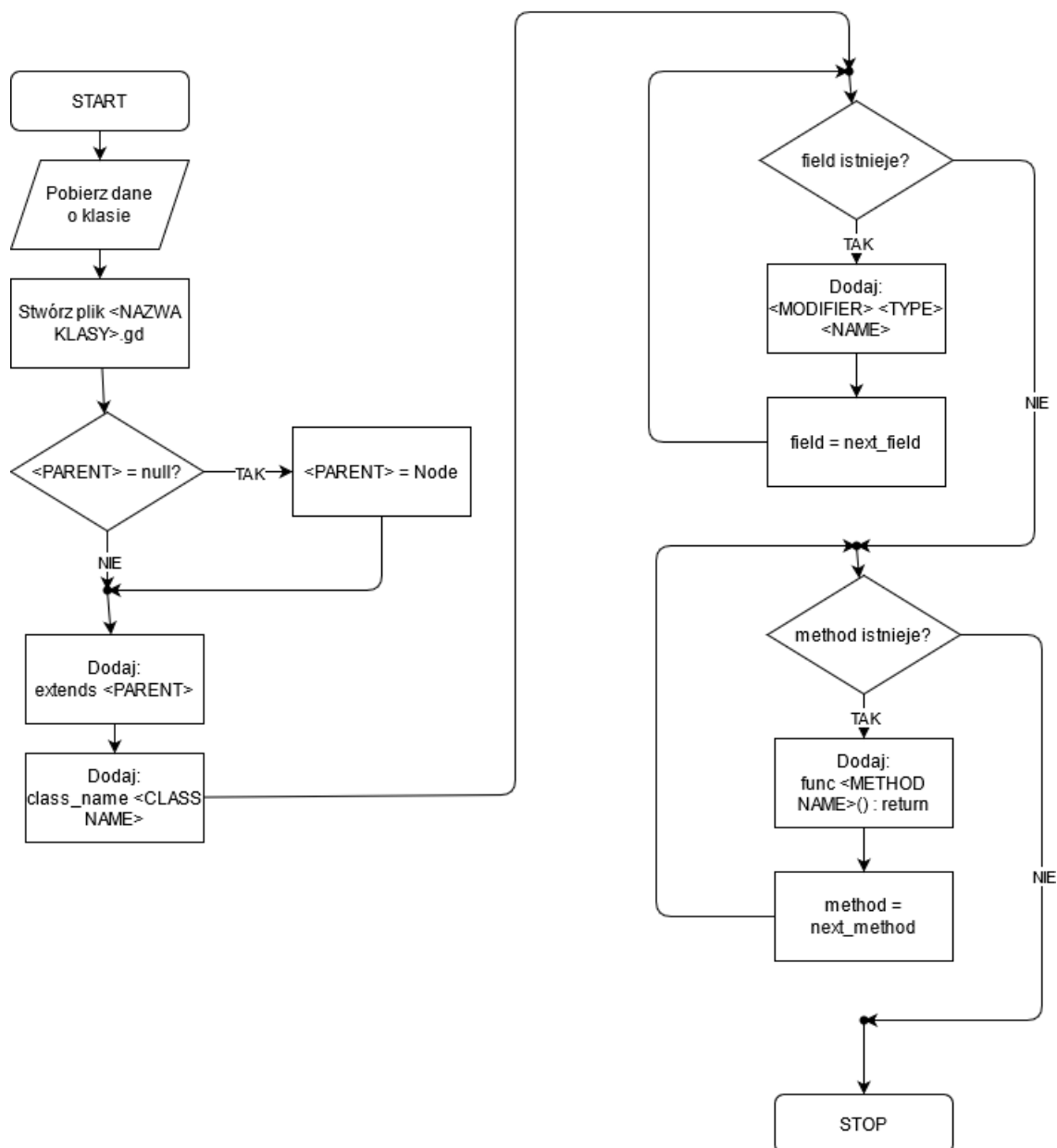
func <METHOD NAME>():
    return

func <METHOD NAME>():
    return
```

3.2. Schemat dla języka C#



3.3. Schemat dla języka GDScript



Poprzez poprawnie poukładanych danych oraz użyciu map (wyjaśnione w dalszej części dokumentacji) możliwa jest prosta konwersja do języka C#, czy też GDScript.

4. Ważniejsze fragmenty kodu

Aby móc wygenerować kody źródłowe, najpierw istotne informacje o klasach zostaną zapisane w directory o nazwie *d*, a całość będzie znajdowała się w tabeli o nazwie *list*. Kod za to odpowiedzialny wygląda następująco:

```
func get_list():
    var conn_list = get_connection_list()
    var list: Array
    for x in get_tree().get_nodes_in_group("graphnode"):
        var d = {}
        d["name"] = x.title
```



```

var parent: String = "Node"
for p in conn_list:
    if p["from_port"] == 0 and get_node(p["to"]) == x:
        parent = get_node(p["from"]).title

var aggr: Array
for p in conn_list:
    var a = {}
    if p["from_port"] == 1 and get_node(p["to"]) == x:
        a["name"] = get_node(p["from"]).title
        aggr.append(a)
        d["agr"] = aggr
        d["parent"] = parent

    d["path"] = x.get_path()
    d["fields"] = x.get_fields_list()
    d["methods"] = x.get_methods_list()
    list.append(d)

return list

```

Do generowania plików .cs użyto listy, której tworzenie pokazano wyżej. Funkcja przechodzi przez wszystkie elementy i wydobywa informacje, które następnie formatuje i zapisuje do pliku.

```

func generateCS(list):
    for node in list:
        var file:= File.new();
        var pfile: String
        if node["path"].length() < 1:
            pfile = ""
        else:
            pfile = node["path"] + "/"
        file.open(pfile + node["name"] + ".cs", File.WRITE)

        file.store_string("using Godot;\nusing System;\n\n")
        file.store_string("public class " + node["name"] + " : " + node["parent"] + "{\n")

        for obj in node["agr"]:
            file.store_string("private " + obj["name"] + " " + obj["name"].to_lower() +
";\n")

        for field in node["fields"]:
            if (field["modifier"] != "none") and (field["modifier"] != "onready"):
                file.store_string("[Export]\n")
            if field["type"] != "var":
                file.store_string("private " + field["type"] + " " + field["name"] + ";\n")
            else:
                file.store_string("Node " + field["name"] + ";\n")
            file.store_string("\n")
        for method in node["methods"]:
            file.store_string("public void " + method["name"] + "() { return; }\n\n")
        file.store_string("}")

```

Generowanie plików .gd jest analogiczne do generowania plików .cs. Występuje tutaj zmiany w formatowaniu oraz inna kolejność użycia i zapisu informacji do pliku.

```

func generateGD(list):
    for node in list:
        var file:= File.new();
        var pfile: String
        if node["path"].length() < 1:

```

```

        pfile = ""
    else:
        pfile = node["path"] + "/"
        file.open(pfile + node["name"] + ".gd", File.WRITE)
        file.store_string("extends " + node["parent"] + "\n\n")

        file.store_string("class_name " + node["name"] + "\n\n")
        for field in node["fields"]:
            if (field["modifier"] != "none"):
                file.store_string(field["modifier"] + " ")
        if field["type"] != "var":
            file.store_string("var " + field["name"] + ": " + field["type"] + "\n")
        else:
            file.store_string("var " + field["name"] + "\n")

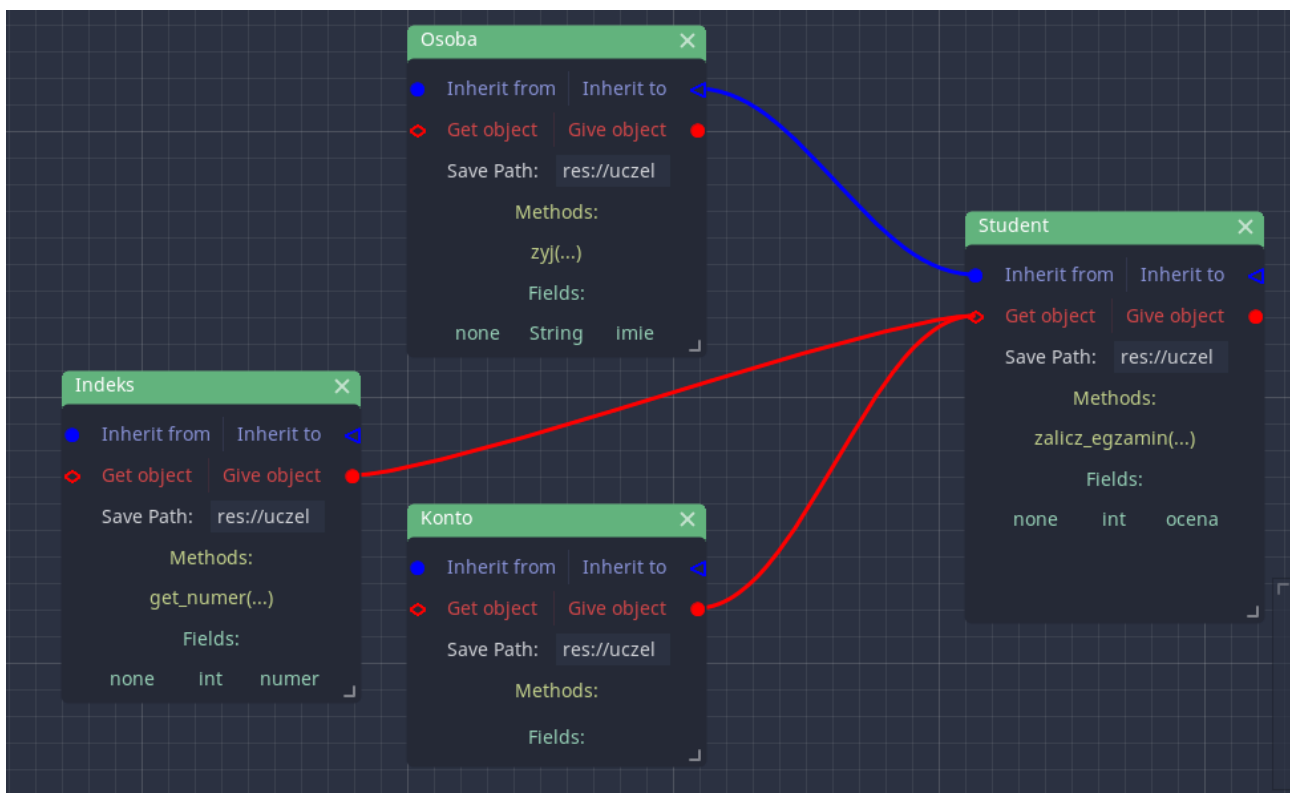
        for obj in node["agr"]:
            file.store_string("var " + obj["name"].to_lower() + " : " + obj["name"] + "\n")

        file.store_string("\n")
        for method in node["methods"]:
            file.store_string("func " + method["name"] + "() : return\n\n")
n")

```

5. Przykład użycia

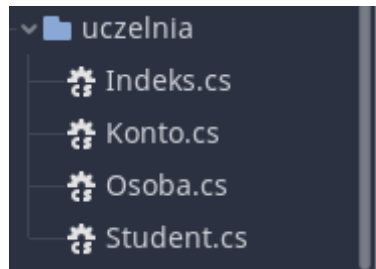
Do przetestowania programu stworzony został przypadek, w którym jest zakładane, iż użytkownik chce stworzyć system przechowujący studentów. Dlatego też utworzony został następujący graf:



Z tak utworzonego grafu będą generowane pliki, których będzie możliwość kompilacji.

5.1. Dla języka C#

Po wciśnięciu przycisku „Generate cs” stworzone zostały 4 pliki:



Zawartość pliku Indeks.cs:

```
using Godot;
using System;

public class Indeks : Node{
    private int numer;

    public void get_numer(){ return; }

}
```

Zawartość pliku Konto.cs:

```
using Godot;
using System;

public class Konto : Node{

}
```

Zawartość pliku Osoba.cs:

```
using Godot;
using System;

public class Osoba : Node{
    private String imie;

    public void zyj(){ return; }

}
```

Zawartość pliku Student.cs:

```
using Godot;
using System;

public class Student : Osoba{
    private Indeks indeks;
    private Konto konto;
    private int ocena;

    public void zalicz_egzamin(){ return; }

}
```

Aby przetestować czy silnik będzie w stanie użyć danej klasy oraz czy samo dziedziczenie zadziała, plugin został użyty w testowym programie, który jedynie otworzy okno w którym wyświetli nadany tekst, natomiast sama klasa Student zostanie zmodyfikowana tak aby metoda `zyj()` zwracała ciąg znaku. Następnie sama metoda zostanie wywołana na obiekcie klasy Student.

Zmiany klasy `Osoba`:

```
public string zyj(){ return "Zyje i jestem osoba"; }
```

Samo użycie klas polega na tym, że najpierw trzeba załadować klasę z pliku, a następnie można tworzyć jej obiekty. W teście, najpierw tworzymy obiekt klasy Student, a potem wywołujemy na tym obiekcie, metodę `zyj()`. Kod aplikacji okna:

```
var student_class = load("res://uczelnia/Student.cs")
var s = student_class.new()

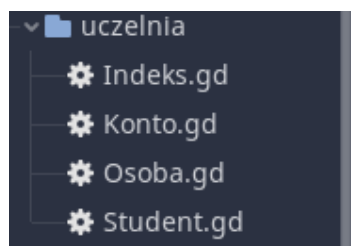
$Label.text = s.zyj()
```

Wynik:



5.2. Dla GDScript

Po wciśnięciu przycisku „Generate gd” stworzone zostały 4 pliki:



Zawartość pliku Indeks.gd:

```
extends Node

class_name Indeks

var numer: int

func get_numer(): return
```

Zawartość pliku Konto.gd:

```
extends Node

class_name Konto
```

Zawartość pliku Osoba.gd:

```
extends Node

class_name Osoba

var imie: String

func zyj(): return
```

Zawartość pliku Student.gd:

```
extends Osoba

class_name Student

var ocena: int
var indeks: Indeks
var konto: Konto

func zalicz_egzamin(): return
```


Również w tym przypadku zmodyfikowano generowany kod tak aby pokazać, że wszystko się kompiluje:

```
func zyj(): return "Jestem osoba"
```

Użycie klas w testowym programie wygląda następująco:

```
var s = Student.new()
$Label.text = s.zej()
```

Wynik:



Jestem osoba

6. Wnioski

Udało stworzyć się plugin do tworzenia graficznego schematu klas na podstawie UML. Z stworzonego schematu istnieje możliwość generowania plików dla dwóch języków, wedle początkowych założeń. Aplikacja posiada podstawowe funkcjonalności, dzięki temu już na teraźniejszym etapie posiada praktyczne zastosowanie, jednak brakuje w niej pewnych zabezpieczeń. Interfejs graficzny również mógłby zostać ulepszony.

Program można w przyszłości rozszerzyć o specyficzne elementy jak np. typy wyliczeniowe.