Politechnika Świętokrzyska w Kielcach
Wydział Elektrotechnik, Automatyki i Informatyki

Projekt: Technologie obiektowe	
Temat: Graficzne programowanie	**Autorzy:** Karol Gardian Dawid Spychalski Damian Gajda
Grupa: 1ID21A	**Data wykonania:** 16.06.2024

1. Wstęp

1.1. Opis tematu

Temat programowania graficznego jest znanym zagadnieniem jednak nie zbyt popularnym przez swoje problem związane z wygodą użytkowania oraz wydajnością czasową. Jest wykorzystywany między innymi w prostych aplikacjach programistycznych głównie przeznaczonych dla dzieci i młodzieży takich jak Scratch czy też Minecraft Education Edition.

1.2. Cel i zakres projektu

Celem projektu było stworzenie programu, który pozwoli tworzyć kod dla języków JavaScript, Python oraz C# bez konieczności bezpośredniego jego pisania, lecz za pomocą bloczków odpowiadającym danym elementom języków programowania takim jak na przykład pętle, warunki, obiekty itp.

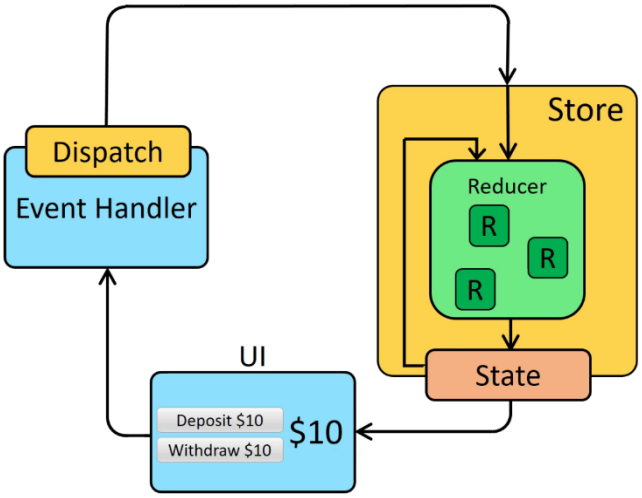
Zakres prac obejmował wiele aspektów, które składały się na działająca finalnie aplikację. Do głównych elementów prac należą:

- Opracowanie graficznego projekt aplikacji obejmujące stworzenie odpowiedniej palety kolorów oraz zaprojektowanie rozmieszczenia kluczowych sekcji aplikacji takich jak paleta bloczków, przestrzeń robocza i obszar generowania kodu.
- Wybór przydatnych technologii po przeanalizowaniu wymagań aplikacji i dostępnych technologii pod kontem tworzenia optymalnego kodu i aplikacji łatwej i prostej w użytkowaniu dla potencjalnych użytkowników.
- Opracowanie logiki przeciągania bloczków opartej na strukturze zapisywanej w JSON pozwalającej przechowywać ich rozmieszczenie i umożliwiającą ich nieskończone zagnieżdżanie.
- Dodanie kluczowych bloczków po przeanalizowanie podstawowych elementów języków programowania o nie standardowej strukturze takich jak pętle i warunki i implementacja ich jako przeciągane bloczki.
- Opracowanie logiki pozwalającej na tworzenie własnych klas zawierających aspekty dla nich dostępne takie jak pola metody i konstruktory wraz z określaniem poziomu dostępności.
- Dodanie generowania kodu języków wspieranych przez aplikacje w oparciu o strukturę rozmieszczonych bloczków.
- Podłączenie aplikacji z dostępnym kompilatorem online celem sprawdzania poprawności działania kodu.
- Ładowanie standardowych klas używając refleksji co pozwala na szeroki pole manewru w kwestii budowy kodu ponieważ użytkownik nie musi się ograniczać tylko do tych stworzonych przez siebie

2. Narzędzia i platforma

Aplikacja została zrealizowana w postaci strony internetowej dzięki czemu może być wykorzystywana na każdej platformie, która jest w stanie korzystać z przeglądarki internetowej. Jako podstawowa technologia użyta do realizacji aplikacji został wybrany framework do JavaScript o nazwie React Js. Pozwala on na tworzenie w ławy i wydajny sposób reaktywnych aplikacji opartych o komponenty dzięki czemu podczas tworzenia kodu można unikać jego nadmiarowości.

Dodatkowo użyto specjalnych bibliotek przewidzianych dla React Js. Pierwszą z nich jest Redux pozwalający na scentralizowanie stanu aplikacji dzięki czemu dane użyte w jednym komponencie nie muszą być ściśle z nim związane tylko przechowywane w globalnym magazynie „stor", który może udostępniać aktualny stan konkretnych danych każdemu komponentowi, który tego potrzebuje. Cały proces działania Redux obrazuje poniższa grafika gdzie UI wysyła zdarzenie wraz z danymi przez Dispatch. Trafia to do odpowiedniej części odpowiedniego Reducera i na podstawie przesłanych



Rysunek 1Przykład działania Redux z strony: https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow

danych jest aktualizowany globalny stan, którego zmiana jest od razu uwzględniana w widokach obecnie z nich korzystających.

Drugą kluczową biblioteką jest Dnd-kit pozwalający stworzyć aplikacje typu „drag and drop". Umożliwia ona stworzenie dwóch typów komponentów „droppable", czyli obszar do którego może zostać przeniesiony obiekt oraz „draggable" czyli komponent który można przemieszczać.

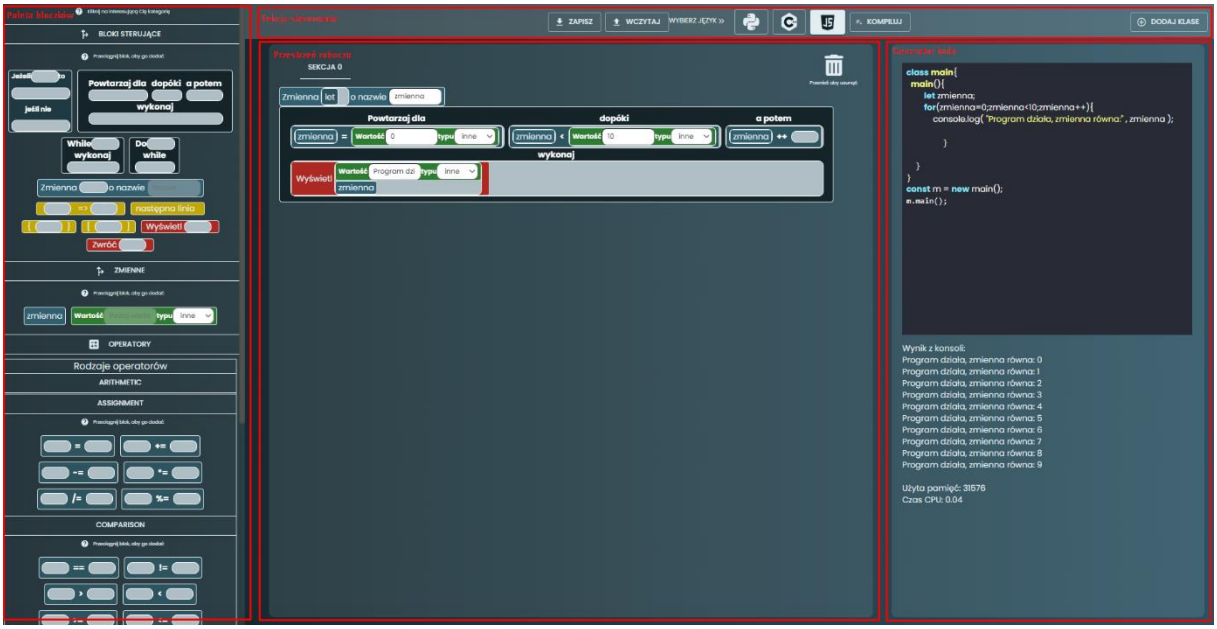
Rysunek 2Komponenty z biblioteki Dnd-kit z strony: https://docs.dndkit.com/introduction/getting-started

Z punktu widzenia naszej aplikacji jest to kluczowa technologia pozwalająca na budowę struktury kodu na zasadzie przeciągania. Każdy obiekt bloczku kodu taki jak pętla, zmienna, metoda itp. jest implementowana jako draggable, natomiast pola wewnątrz tych bloczków w których można umieścić elementy tak jak na przykład w przypadku warunku, który ma miejsce na wartość logiczną warunku i akcje w przypadku spełnienia warunku są realizowane jako droppable.

3. Realizacja

3.1. Projekt graficzny aplikacji

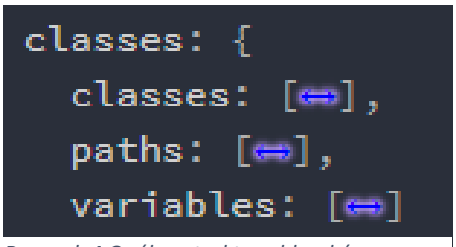
Pierwszym etapem tworzenia aplikacji był rozplanowanie rozmieszczenia wszystkich jej kluczowych elementów. Na rysunku poniżej kolorem czerwonym zostały zaznaczone jej elementy. Po lewej stronie znajdują się rozwijane listy z bloczkami używanymi do tworzenia kodu. Są sekcje z podstawowymi bloczkami, zmiennymi, operatorami, typami danych i utworzonymi klasami. Środkowa część ekranu reprezentuje przestrzeń roboczy czyli miejsce do którego są przeciągane ploczki z palety celem budowania kodu. Prawa sekcja nazwana „Generator kodu" zawiera strukturę kodu wygenerowana na podstawie bloczków w sekcji środkowej, a poniżej wyniki otrzymane od kompilatora. Ostatnim elementem jest sekcja górna pozwalająca zapisywać i wgrywać pliki, zmieniać język programowania, kompilować kod oraz dodawać nowe klasy.



Rysunek 3 Interfejs aplikacji

3.2. Struktura bloczków

W projekcie struktura tworzonego kodu jest reprezentowana jako JSON z podziałem na trzy sekcje classes, paths oraz variables. Całość tych danych jest przechowywana w magazynie Redux dlatego każdy komponent który potrzebuje danych może swobodnie uzyskać do nich dostęp.



Rysunek 4 Ogólna struktura bloczków

Pierwsza sekcja czyli classes przechowuje strukturę klas wykorzystywanych w projekcie użytkownika w tym też domyślnej klasy main od której rozpoczyna się w kodzie użytkownika działanie programu. Struktura ta zawiera nazwę klasy oraz listy pól, metod oraz konstruktorów. Na poniższym obrazku widać fragment struktury głównej klasy z metodą mainMethod w której został zagnieżdżony bloczek odpowiedzialny za deklaracje zmiennej w której ustawione jej typ na let oraz nazwę na zmienna.

Rysunek 5 Struktura zagnieżdżeń bloczków

W części struktury nazwanej paths znajdują się składające się z „id” oznaczającego bloczek którego dotyczą oraz „path” czyli tablicy zawierające kolejne elementy ścieżki pozwalające odnaleźć element.

Ścieżki pozawalają szybko odnajdywać przenoszone bloczki. Jest to kluczowe do poprawnego działania tego systemu ponieważ, gdy bloczek zostanie przeniesiony z jednej lokalizacji do drugiej kod aplikacji otrzymuje id bloczku przenoszonego oraz id pola do którego został dany bloczek przeniesiony. Na poniższej grafice jest widoczna ścieżka odpowiadająca bloczkowi typu zmiennej zagnieżdżonemu w

bloczku deklaracji zmiennej który został przedstawiony w strukturze powyżej. Zamieszczone ścieżka wskazuje podczas przemieszczania bloczku jest odczytywana i interpretowana w następujący sposób:

1. **classes** -Wejście w listę „classes” z wszystkimi klasami utworzonymi przez użytkownika aplikacji.
2. **mainClass|-1** - Wybranie z listy klasy o id „mainClass”.
3. **methods** - Wejście w listę „method” w której znajdują się wszystkie metody utworzone w ramach klasy.
4. **mainMethod|2** – Wejście do metody o methodID „mainMethod”, a następnie do drugiej listy w elemencie „children”
5. **le2b892e-f7…|0** - Wyszukanie w liście elementu o podanym id, a następnie przejście do listy numer 0 w elemencie „children”.

Rysunek 6. Struktura śćieżek bloczków

W ten sposób program odnajduje lokalizacje wyszukiwanego elementu i po przejściu przez całą ścieżkę ma pewność, że dana lokalizacja zawiera element o żądanym id .W elementach ścieżek można wyróżnić kilka charakterystycznych elementów:

- <nazwa listy> - pozwala określić listę w elemencie do której należy przejść np. „classes” lub „methods”
- <id elementu listy>|-1 – pozwala określić element listy do którego należy przejść na podstawie id elementu listy. Dodatkowe „-1” na końcu oznacza że nie należy przechodzić automatycznie do potomka bo na przykład może nie istnieć.
- <id elementu listy>|<numer potomka> - działa podobnie jak poprzednie element lecz dodatkowo po przejściu do wskazanego elementu przechodzi do „children”, a następnie do listy podanej jako numer potomka.

3.3. Logika przeciągania bloczków

Każdy operacja przeciągnięcia wykonywana przez użytkownika generuje zdarzenie z informacjami o id elementu przeciągane i id obszaru do którego element jest przeciągany. Można wyróżnić dwa szczególne przypadki tej operacji. Pierwszym jest przeciągnięcie elementu z palety, czyli utworzenie nowego elementu w strukturze, co obejmuje wygenerowanie struktury obiektu, zlokalizowanie

miejsca docelowego oraz dodanie w nim bloczku oraz zapisanie w „paths” ścieżki do nowo dodanego elementu. Drugi rodzajem operacji jest przeniesienie bloczku z jednej lokalizacji do drugiej. Jest to bardziej złożona operacja ponieważ wymaga zlokalizowania w strukturze obiektu przenoszonego oraz lokalizacji docelowej. Następnie struktura przenoszonego bloczku jest kasowana z poprzedniej lokalizacji i dodawana w nowej. Ponadto jest aktualizowana ścieżka do przeniesionego bloczka.

Całość operacji związanych z przeciąganiem bloczków jest realizowana w funkcji zaimplementowanej w Redux. Do funkcji są przekazywane 3 wartości. Id obiektu przenoszonego nazwane „object”, id lokalizacji docelowej „to” oraz id klasy w obrębie której ta operacja jest realizowana nazwane „classId”. Dwa pierwsze id muszą zostać podzielone funkcja split z dzielnikiem „|” ponieważ są tam zawarte dodatkowe informacje.

```
const { object, to, classId } = action.payload;
//podział id obiektu dodawanego
const objectSplit = object.split("|");
//podział id lokalizacji
const toSplit = to.split("|");
```

Po podziale danych element zerowy wynikowych tablicy „toSplit” i „objectSplit” jest identyfikatorem bloczku. Jest on wykorzystywany do wyszukania ścieżek tych elementów w strukturze omawianej powyżej.

```
//pobranie ścieżek lokalizacji docelowej
let pathTo = findPath(state, toSplit[0]);
//pobranie ścieżek lokalizacji obiektu
let pathObject = findPath(state, objectSplit[0]);
```

Sytuacją szczególną jest tu przeciąganie bloczka z palety. W tej sytuacji „object” zamiast zawierać identyfikator bloczka posiada unikalna nazwę typu bloczku. W tej sytuacji zmienna „pathObject” po wyszukiwaniu ścieżki będzie zawierała wartość undefined. Jest to znakiem dla programu, że jest wykonywana akcja dodawania nowego bloczka. W tym przypadku używając funkcji „GetBlocksStructure” jest generowana struktura adekwatna do dodawanego elementu. Jeśli jednak do „pathObject” udało się pobrać ścieżkę metoda „findObject” korzystając z niej wyszukuje w strukturze kodu istniejący bloczek i zapisuje go w „objectValue”.

```
let objectValue = undefined;
if (pathObject === undefined) {
  //generowanie struktury nowego obiektu
  if (objectSplit[0] === classVariableBlock)
    objectValue = GetBlockStructure(
      object +
        "|" +
        classId +
        "|" +
        (pathTo.length > 3 ? pathTo[3].split("|")[0] : toSplit[0])
    );
  else objectValue = GetBlockStructure(object);
} else {
  //pobranie struktury obiektu z starej lokalizacji
  objectValue = findObject(state, object, pathObject);
}
```

Pobieranie wartości elementu struktury realizowane w „findObject” dodatkowo używa „findLocationByPath”. Ta funkcja pozwala odnaleźć listę w której znajduje się szukany obiekt na podstawie ścieżki. Gdy lokalizacja zostanie zwrócona wtedy korzystając z standardowej funkcji „find” jest wyszukiwany element o żądanym id.

```
export const findObject = (state, id, path) => {
  return findLocationByPath(state, path).find(
    (el) => el.id === id
  );
};
```

Wyszukiwanie lokalizacji przez „findLocationByPath” jest złożonym procesem. Funkcja otrzymuje dwa parametry, object zawierający strukturę JSON w której ma być realizowane przejście po ścieżce oraz path z ścieżką, którą program ma podążać. Głównym elementem funkcji jest przejście pętlą „forEach” po wszystkich elementach ścieżki. Ponieważ każdy jej element może zawierać dwie dane oddzielone separatorem „|” jest używana funkcja split do podziału danych. Jeśli po podziale uzyskamy dwa element oznacza to, że pierwszy z nich jest identyfikatorem elementu tablicy w która program obecnie przetwarza, a drugi element numerem potomka z elementu „children”. Dlatego w kodzie najpierw za pomocą „find” jest wyszukiwany wymagany element, a potem następuje przejście do jego pola „children” w, którym jest wybierana tablica odpowiadająca numerowi indeksu zapisanemu w „splitTargent[1]”. Jednak w sytuacji, gdy drugi element po podziale będzie miał wartość -1 oznacza to, że należy tylko przejść do elementu o id podanym w „splitTarget[0]” ponieważ przetwarzany w danej chwili obiekt nie zawiera pola „children” bo jest na przykład strukturą klasy.

Funkcja „findLocationByPath” ma jeszcze drugi tryb działania używany, gdy po podziale elementu ścieżki nie uzyskamy dwóch elementów. Oznacza to, że należy przejść do pola struktury JSON, której nazwa jest podana w elemencie ścieżki korzystając z funkcji „reduce”

```
export const findLocationByPath = (obj, path) => {
  let newObj = obj;
  let tmp = [];
  let nextIsSplitArray;
  if (!path) return;
  path.forEach((v, i) => {
    const splitTarget = v.split("|");
    if (splitTarget.length === 2) {
      if (splitTarget[1] !== "-1") {
        newObj = newObj.find((el) => el.id === splitTarget[0]).children;
        newObj = newObj[splitTarget[1]];
      } else {
        newObj = newObj.find((el) => el.id === splitTarget[0]);
      }
      nextIsSplitArray = JSON.stringify(newObj);
    } else {
      tmp = [splitTarget[0]];
      newObj = tmp.reduce((acc, key) => acc[key], newObj);
      nextIsSplitArray = JSON.stringify(newObj);
    }
  });
  return newObj;
};
```

Wracając do głównej funkcji realizującej logikę przeciągania w kolejnych krokach mamy zlokalizowanie miejsca docelowego przenoszonego bloczku do czego jest używana omawiana przed chwila funkcja. Jeśli uda się odnaleźć żądaną lokalizacje oznacza to, że użytkownik dokonał dozwolonego przeniesienia bloczku i operacja może być dalej kontynuowana. Kolejne operacje obejmują precyzyjne wyznaczenie obiektu i pola do którego ma zostać przeniesiony bloczek, a ostatnia zadaniem na tym etapie jest skasowanie przenoszonego elementu z starej lokalizacji realizowane przez „findAndDeleteByPath" w przypadku gdy element nie jest nowy elementem dodanym z palety co sprawdza warunek na końcu fragment kodu poniżej.

```
      //odszukanie lokalizacji docelowej na podstawie ścieżki
      let destinationValue = findLocationByPath(state, pathTo);
      if (!destinationValue) return;
      destinationValue = destinationValue.find((el) => el.id === toSplit[0]);
      if (toSplit.length === 2) {
        destinationValue = getObjectByPath(destinationValue, ["children"]);
        destinationValue = destinationValue[toSplit[1]];
      }
      //wyznaczenie i usunięcie obiektu w starej lokalizacji
      if (pathObject !== undefined)
        findAndDeleteByPath(state, pathObject, object);
```

Sam mechanizm usuwania bloczku jest dość prosty. W swoim działaniu używa przedstawionej już metody „findLocationByPath" do zlokalizowania elementu do usunięcia oraz wyznaczenia indeksu elementu na liście po to, aby korzystając z funkcji „splice" go usunąć.

```
export function findAndDeleteByPath(state, pathToObject, objectId) {
  //wyznaczenie starej lokalizacja "object"
  const oldObjectLoaction = findLocationByPath(state, pathToObject);
  //wyznaczenie indeksu "object" w starej lokalizacja
  const oldObjectLoactionIndex = findLocationByPath(
    state,
    pathToObject
  ).findIndex((el) => el.id === objectId);
  //usuniecie object z starej lokalizacji
  oldObjectLoaction.splice(oldObjectLoactionIndex, 1);
}
```

Ostatnimi akcjami podejmowanymi w logice bloczków jest manipulacja ściekami obejmująca dodanie nowej ścieżki w przypadku tworzenia nowego bloczku i aktualizacje już istniejących ścieżek w sytuacji przenoszenia bloczka do innej lokalizacji.

3.4. Tworzenie własnych klas

Program stawia na uniwersalność dlatego pozwala użytkownikowi tworzyć jego własne klasy. Każda klasa jest prezentowana w interfejsie graficznym jako komponent składające się z 3 głównych sekcji. Pierwsza z nich pozwala na dodawanie pól klasy poprze użycie przycisku „Dodaj nowe pole klasy". Każde pole pozwala na określenie poziomu dostępu, typu danych oraz nazwy. Kolejna sekcja odpowiada z konstruktory posiadające możliwość określenia stopnia dostępności oraz listy parametrów poprzez przeciągnięcie w odpowiednie miejsce odpowiedzialnego za to bloczka. Poniżej znajdują się dwa obszary. Po lewej stronie jest dodatkowa paleta z bloczkami reprezentującymi pola metod oraz parametry konstruktora, a po prawej jest przestrzeń robocza konstruktora w której należy zamieszczać bloczki, które mają się wykonywać w ramach jego wywołania. Nowe konstruktory mogą

być dodawane poprzez użycie przycisku „Dodaj nowy kostruktor klasy”. Poniżej znajdują się utworzone metody których logika działania niemal nie odbiega od konstruktorów. Posiadają jedynie dodatkowe pole na określenie ich nazwy. (Opisana struktura tworzenia klas dotyczy języka C# pozostałe języki mogą delikatnie się różnić)

Wszystkie elementy utworzonych klas są dodatkowo reprezentowane poprzez bloczki w głównej plecie używane do tworzenia obiektów odwoływania się do pól oraz metod klas

Rysunek 7 Struktura kreatora kodu

3.5. Generowanie kodu

Aplikacja został wyposażona z generator kodu który przechodząc po strukturze bloczków zapisanej w JSON tworzy kod który jest widoczny w prawej sekcji aplikacji. Kady bloczek dodany prze użytkownika jest interpretowany jako fragment kodu wyświetlany po prawej stronie. Działanie generatora jest w pełni automatyczne w wykonywane w czasie rzeczywisty dlatego każda wprowadzona miana jest natychmiast widoczna w postaci nowego kodu

Rysunek 8 Przykład generatora kodu

Poniżej został zamieszczony fragment generatora kodu odpowiadający za tworzenie instrukcji warunkowej „if”. W jego strukturze użyto dwóch kluczowych funkcji. Pierwszą z nich jest „traverse”, która pozwala rekurencyjnie przechodzić po elementach zagnieżdżonych w elementach obecnie przetwarzanych. Na przykład strukturze if mogą wystąpić 3 zagnieżdżenia w warunku, sekcji jeśli warunek spełniony oraz sekcji jeśli warunek nie spełniony. Z tego powodu w tych miejscach znajduje się funkcja „traverse”. Do poprawnego działania należy jej przekazać następujące parametry:

- json: Cała struktura JSON opisująca klasy, pola, metody itd.
- obj: Struktura komponentów w metodzie (część JSON-a, która jest przetwarzana).
- classObject: Obiekt reprezentujący całą strukturę klasy.
- level: Liczba tabulatorów od lewej strony (poziom zagnieżdżenia kodu).
- addSemicolon: Flaga określająca, czy po elementach w komponencie powinien być dodany znak ;.
- adder: Wypełniacz, używany do dodawania dodatkowych znaków między elementami (np. ,).

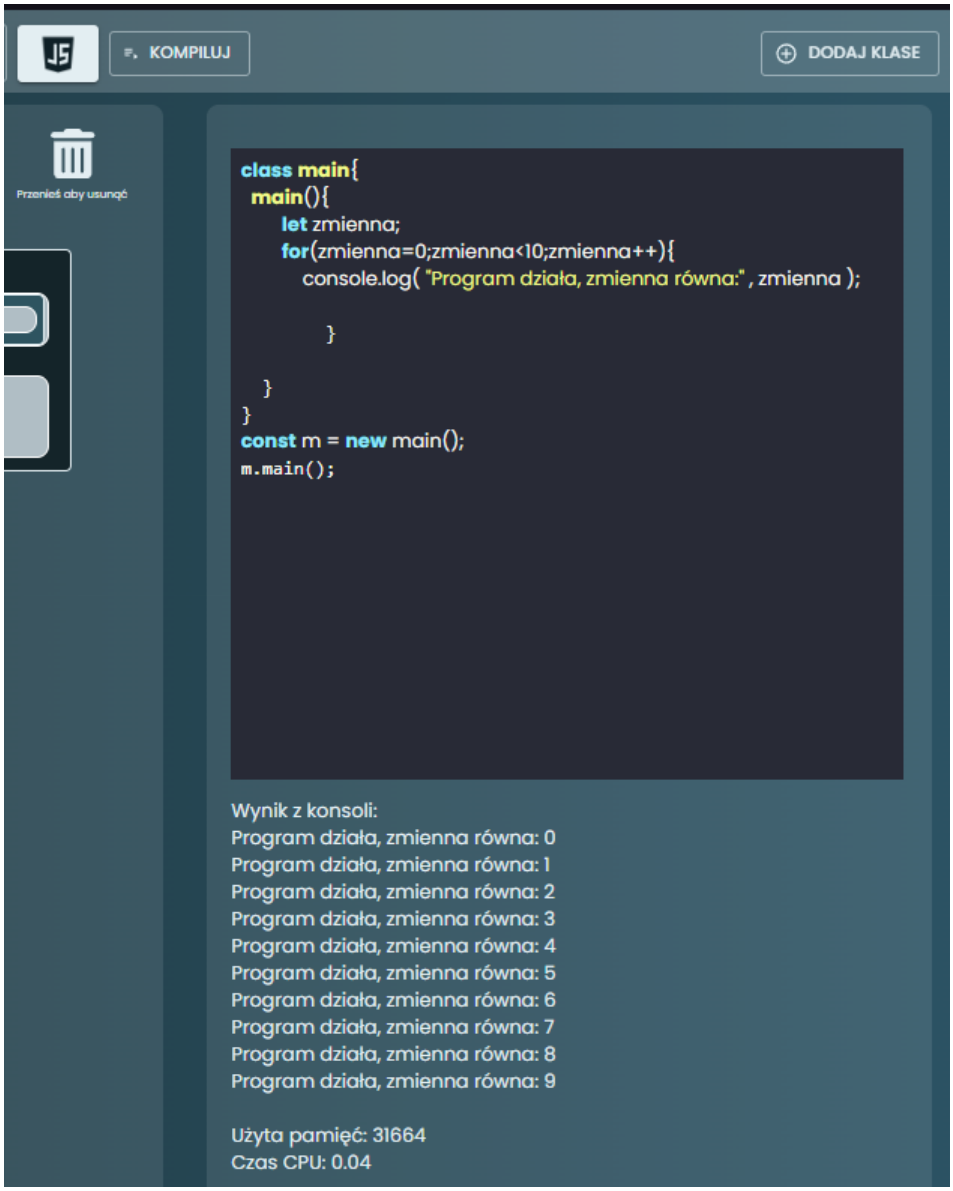
Drugą używana funkcją jest „generateTabs”. Jej zastosowanie jest czysto estetyczne i pozwala zadbać o generowanie poprawnych wcięć w kodzie dzięki czemu cała struktura jest bardziej czytelna. Jako parametr przyjmuje „level” określający liczbę wcięć do wygenerowania.

```
case ifElseBlock:
          result += generateTabs(level) + "if(";
          result += traverse(
            json,
            element.children[0],
            classObject,
            0,
            false,
            ""
          );
          result += "){\n";
          result += traverse(
            json,
            element.children[1],
            classObject,
            level + 1,
            true,
            ""
          );
          result += "\n" + generateTabs(level) + "}else{\n";
          result += traverse(
            json,
            element.children[2],
            classObject,
            level + 1,
            true,
```

```
            ""
        );
        result += "\n" + generateTabs(level) + "}\n";
        break;
```

3.6. Kompilacja kodu

Aplikacja poza generowaniem kodu pozwala również na jego kompilacje i uruchomienie. W tym celu użytkownik aplikacji musi użyć przycisku kompiluj zlokalizowanego w górnej części ekranu. Wtedy następuje wygenerowanie kodu obejmującego wszystkie utworzone klasy i przeslanie poprzez HTTP do serwisu JDoodle który udostępnia możliwość kompilacji i uruchamiania kodów. Serwis w odpowiedzi odsyła użytkownikowi wynik uruchomienia programu który jest wyświetlany poniżej kodu. W przypadku gdy kod zawiera błąd zostanie wyświetlony tekst z opisem błędu kompilacji wygenerowany prze kompilator.

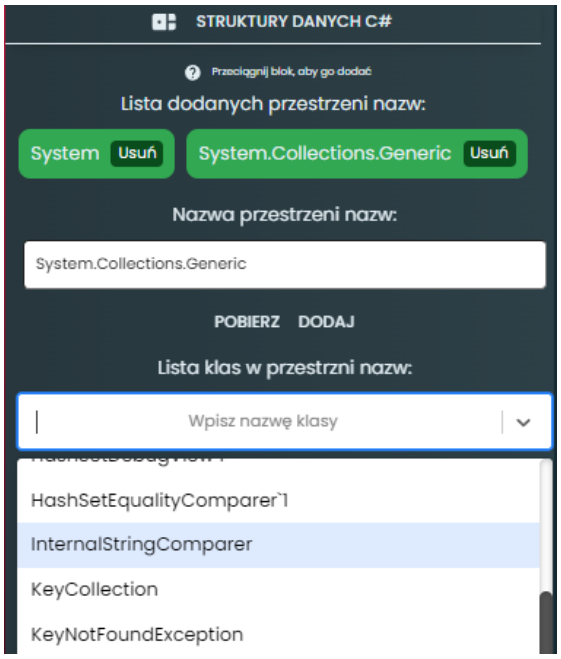


Rysunek 9 Przykład działania kompilatora

3.7. Pobieranie standardowych klas przez refleksje

W celu stworzenia uniwersalnego narzędzia aplikacja daje możliwość wykorzystywania klas standardowych dla danego języku programowania. Został w tym celu zastosowany mechanizm refleksji.

Z poziomu języka C# w palecie jest dostępna sekcja „Struktury Danych C#". Pozwala ona na dostęp do dowolnego obiektu. Na początku użytkownik musi podać nazwę przestrzeni nazw w której dany obiekt się znajduje i użyć przycisku „Pobierz". W tym momencie kompilator online wykonuje kod C# który zwraca listę wszystkich klas dostępnych w określonej przestrzeni nazw. Aplikacje prezentuje klasy na rozwijanej liście, z której użytkownik może wybrać klasę z której chce skorzystać. Użytkownik może również przyciskiem „Dodaj" określić które przestrzeni nazw mają być używane w projekcie podczas generowania kodu. Używane przesyczenie nazw są widoczne powyżej pola do wyszukiwania



Rysunek 10 Wyszukiwanie standardowej klasy C#

Po wybraniu klasy na kompilatorze online ponownie jest wykonywany kod, ale tym razem zwraca ona listę metod dostępnych dla danej klasie. Na podstawie tej listy są generowane bloczki które użytkownik może przeciągać na przestrzeń roboczą i w ten sposób korzystać z standardowych klas języka.

Rysunek 11 Dostęp do metod standardowej klasy C#

Wspomniany wcześniej kod C# odpowiedzialny za pobieranie nazw z przestrzeni nazw działa w następujący sposób. Prze wysłaniem kodu do kompilatora do zmiennej „namespaceName" jest przypisywana wprowadzona przez użytkowana nazwa. Następnie jest wywoływana funkcja „PrintClassesAndMethodsInNamespace", a w niej są pobierane wszystkie załadowane do domeny aplikacji zestawy (assemblies) i przypisuje je do tablicy assemblies. Następnie jest wykonywana filtracja w celu uzyskania klas które należą do zadanej przestrzeni nazw i sortowanie ich alfabetycznie według nazwy. W kolejnym kroku jest tworzony słownik (result) do przechowywania nazw klas i ich metod. Jeśli znaleziono klasy, iteruje przez każdą z nich, pobiera jej metody publiczne (zarówno statyczne, jak i instancje), sortuje je i dodaje do listy metod w słowniku. Na sam koniec jest wykonywana konwersja do formatu JSON i wypisanie go na konsoli.

```
class Program {
  static void Main(string[] args) {
    string namespaceName = "${nameSpace}";

    if (!string.IsNullOrEmpty(namespaceName)) {
      PrintClassesAndMethodsInNamespace(namespaceName);
    }
  }

  static void PrintClassesAndMethodsInNamespace(string namespaceName) {

    Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();

    var types = assemblies.SelectMany(a =>a.GetTypes()).Where(t =>t.IsClass &&
t.Namespace == namespaceName).OrderBy(t =>t.Name);

    var result = new Dictionary < string,
    List < string >> ();

    if (types.Any()) {
      foreach(var type in types) {
        var methods = type.GetMethods(BindingFlags.Public |
BindingFlags.Instance | BindingFlags.Static).OrderBy(m =>m.Name).Select(m
=>m.Name).ToList();

        result[type.Name] = methods;
      }

      string jsonResult = ConvertToJson(result);
      Console.WriteLine(jsonResult);
    }
  }
}
```

Działanie drugiego kodu jest niemal identyczne. Wyróżnia go jedynie to, że użytkownik podaje nazwę przestrzeni nazw i nazwę klasy i w rezultacie jest zwracany struktura JSON z metodami wskazanej klasy.

Dla języka JavaScript refleksja jest dostępna z sekcji „Metody klas javaScript”. Sekcja ta posiada pole które umożliwia wyszukiwanie klas wbudowanych w JavaScript. Po wybraniu klasy, kod odpowiedzialny za refleksję pobiera metody i tworzy bloczki dla tych metod oraz wyświetla je gotowe do użycia.

Rysunek 12 Dostęp do metod standardowej klasy JS

Refleksja w JavaScript opiera się o obiekt global który zawiera wszystkie zadeklarowane klasy i zmienne globalne. Funkcja „setCurrentMethodsFromReflection” na samym początku wyszukuje nazwę klasy w obiekcie global. Następnie tworzy deklarację dla metod statycznych oraz pobiera wszystkie metody statyczne dla wybranej klasy. W kolejnym kroku funkcja tworzy deklaracje dla zwykłych metod oraz pobiera zwykłe metody.

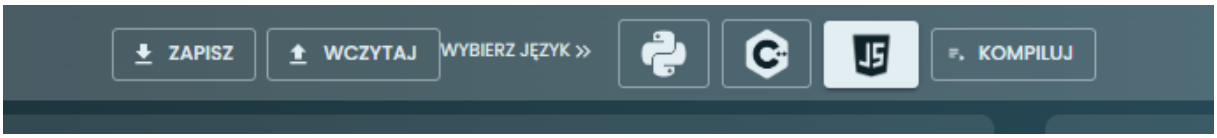
```
setCurrentMethodsFromReflection(state, action) {
  state.blockTypes.currentMethodsFromReflection = [];
  state.currentClassName = {
    value: action.payload.name,
    label: action.payload.name,
```

```
};
const foundClass = global[action.payload.name];
if (!foundClass) return;
const staticMethodNames = Object.getOwnPropertyNames(foundClass);
if (staticMethodNames.length > 0) {
  const declaration = {
    id: action.payload.name + ";js;reflection;declaration",
    texts: [action.payload.name],
    styleClass: "bg-color-js-second-variant",
    structureJS: action.payload.name + "? ",
    moveText: action.payload.name,
    disableMainDroppable: false,
    appendBeforeTraverseInJSGenerator: true,
    disableComma: true,
  };
  state.blockTypes.currentMethodsFromReflection.push(declaration);

  staticMethodNames.forEach((methodName) => {
    const method = {
      id: methodName + ";js;reflection;staticMethod",
      texts: [methodName],
      styleClass: "bg-color-js-second-variant",
      structureJS: "." + methodName + "( ? )",
      moveText: methodName,
      disableMainDroppable: false,
      appendBeforeTraverseInJSGenerator: true,
    };

    state.blockTypes.currentMethodsFromReflection.push(method);
  });
}
const proto = foundClass.prototype;
if (proto === undefined) return;

const methodNames = Object.getOwnPropertyNames(proto);
if (methodNames.length === 1 && methodNames.includes("constructor"))
  return;

const declaration = {
  id: action.payload.name + ";js;reflection;declaration2",
  texts: ["", action.payload.name],
  styleClass: "bg-color-js-first-variant",
  structureJS: "new " + action.payload.name + "( ? )",
  moveText: action.payload.name,
  disableMainDroppable: false,
  appendBeforeTraverseInJSGenerator: false,
  reflect: true,
};
state.blockTypes.currentMethodsFromReflection.push(declaration);
```

```
    methodNames.forEach((methodName) => {
      let descriptor = Object.getOwnPropertyDescriptor(proto, methodName);
      if (
        !descriptor ||
        (typeof descriptor.value !== "function" &&
          typeof descriptor.get !== "function" &&
          typeof descriptor.set !== "function") ||
        methodName === "constructor"
      )
        return;
      const method = {
        id: methodName + ";js;reflection;method",
        texts: ["", methodName],
        styleClass: "bg-color-js-first-variant",
        structureJS: "." + methodName + "( ? )",
        moveText: methodName,
        disableMainDroppable: false,
      };

      state.blockTypes.currentMethodsFromReflection.push(method);
    });
  },
```

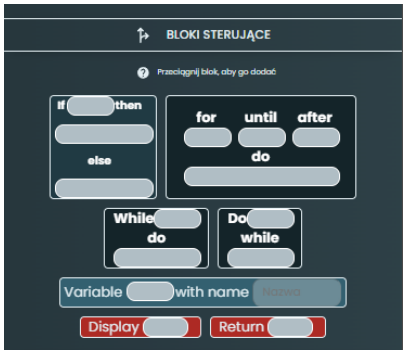
4. Instrukcja korzystania

Po uruchomieniu aplikacji użytkownikowi ukazuje się interfejs z pustą przestrzenią roboczą i domyślnie wybranym językiem JavaScript. Na tym etapie można wczytać utworzony i pobrany projekt używając przycisków „Zapisz” i „Wczytaj” oraz zmienić język programowania przyciskami obok. Każda zmiana języka skutkuje skalowaniem postępów z powodu występowania bloczków, które z powodu struktury obsługiwanych języków nie są między nimi kompatybilne. Aby uniknąć przypadkowej zmiany przed jej końcową realizacją użytkownik musi zatwierdzić to w oknie dialogowym.

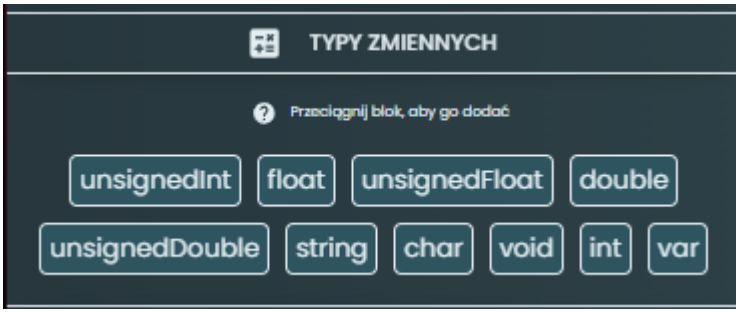


Rysunek 13 Menu górne aplikacji

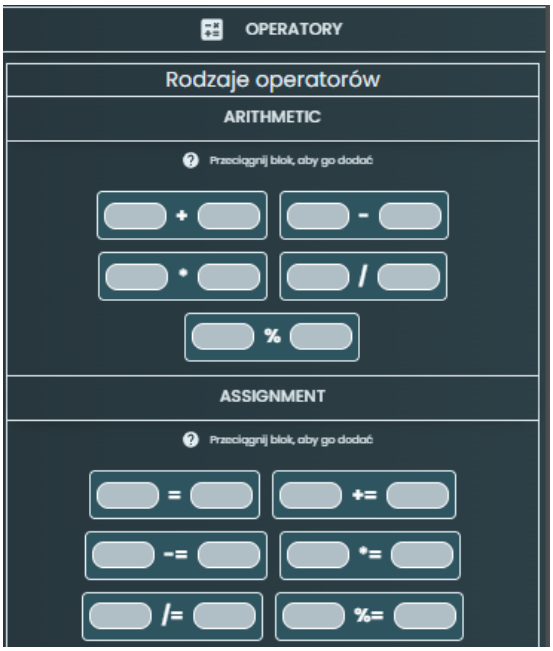
Teraz użytkownik może przystąpić do tworzenia swojego kodu z użyciem bloczków dostępnych w palecie po lewej stronie. Są tam dostępne różnorodne loczki dające ogromne możliwości, dzięki przede wszystkim zastosowaniu mechanizmu refleksji celem dostępu do standardowych klas danego języka (poniżej przykładowe listy bloczków).



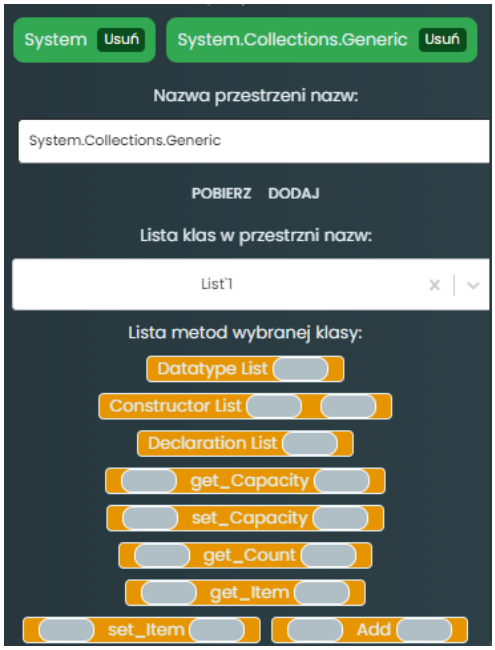
Rysunek 16 Lista bloczków sterujących C#



Rysunek 17 Lista typów zmiennych C#



Rysunek 15 Lista bloczków operatorów

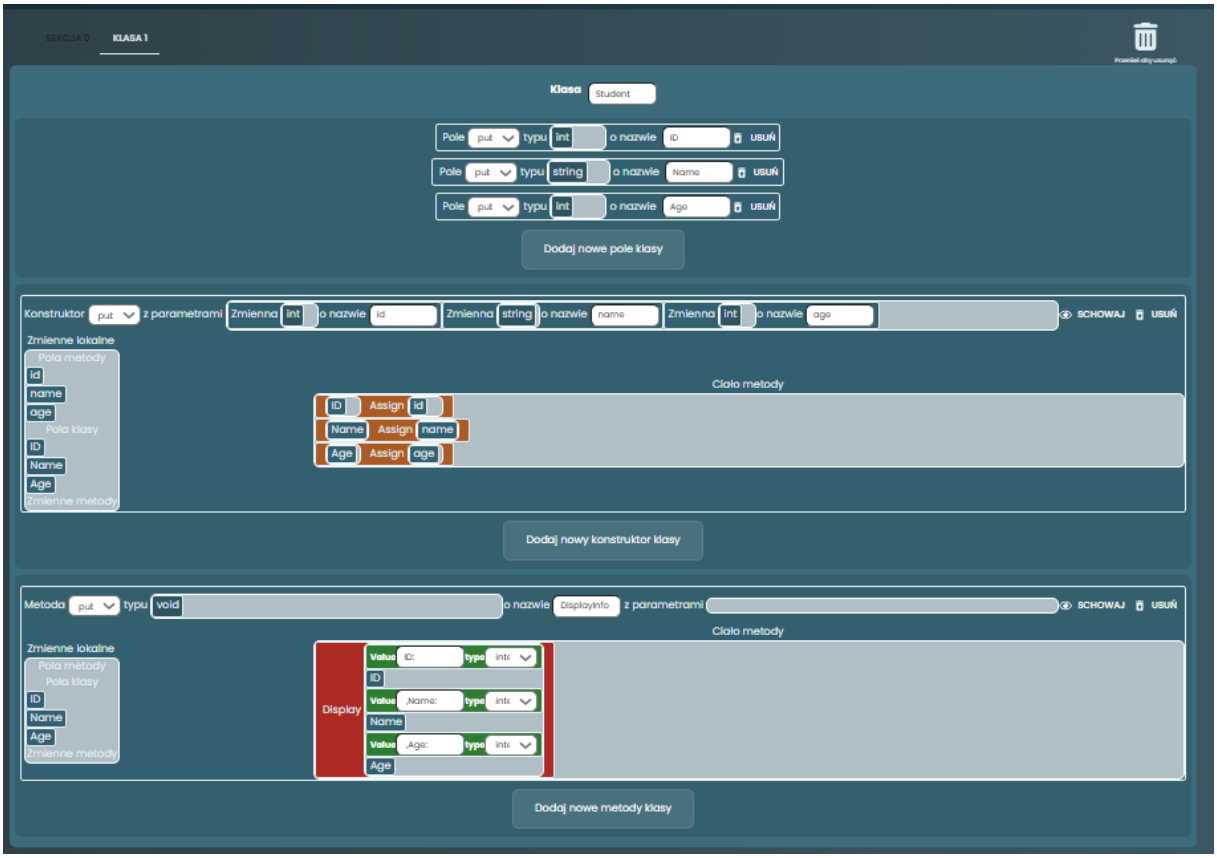


Rysunek 14 Lista bloczków metod klasy z refleksji

Na każdym etapie pracy z aplikacją użytkownik może stworzyc swoją klase używając przycisku w prawym górnym rogu „Dodaj klase”, który utworzy nową sekcje kodu odpowiedzialna na budowanie elemntów utowrzonej klasy.

Rysunek 18 Dodawanie nowej klasy

Z poziomu nowej zakładki użytkownik może tworzyć nowe pola, konstruktory oraz metody. Te dwa ostatnie mają dodatkowe pole podpisane „Ciało metody”. Jest to przestrzeń w której można budować ich kod. Po ich prawej natomiast znajdują 3 rodzaje bloczków: pola klasy, parametry metody oraz zmienne lokalne utworzone w metodzie.



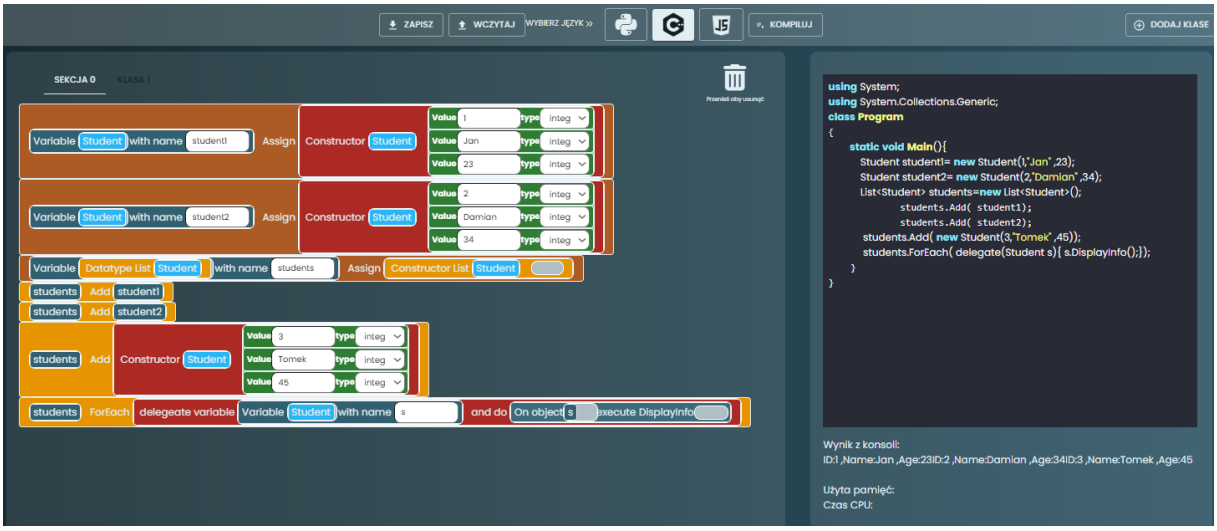
Rysunek 19 Budowanie klasy

Każda klasa dodana przez użytkownika jest reprezentowana w palecie w zakładce „Klasy”. Są tam generowane bloczki pozwalające uzyskać dostęp do pól oraz wywoływać metody. Dodatkowo znajdują się tam też elementy koloru niebieskiego reprezentujące typ danych jakim jest klasa.

Rysunek 20 Dostęp do pól i metod klasy użytkownika

Mając przygotowane odpowiednie klasy można przystąpić do tworzenia doku main od którego rozpoczyna się wykonywanie programu. Na poniższej grafice został przedstawiony przykładowy

program tworzący obiekty studentów, a następnie dodający je do listy i wyświetlający wszystkie jej elementy. W trakcie tworzenia kodu w czasie rzeczywisty stan generowanego kodu po prawej stronie jest odświeżany. Pole wyświetla zawsze kod sekcji która jest obecnie wybrana. Ostatnią rzeczą jaka pozostaje użytkownikowi jest użycie przycisku „Kompiluj”, aby całość kodu main i dodanych klas został przesłana do kompilatora online i wykonana, a wynik wyświetlony pod przestrzenią kodu.



Rysunek 21Budowanie kodu main dla C#