Środowisko do tworzenia schematów blokowych i generacji kodu

Mateusz Buczek, Wojciech Kwiecień
Promotor: dr inż. Joanna Strug

Politechnika Krakowska
Wydział Inżynierii Elektrycznej i Komputerowej
Kraków 2012

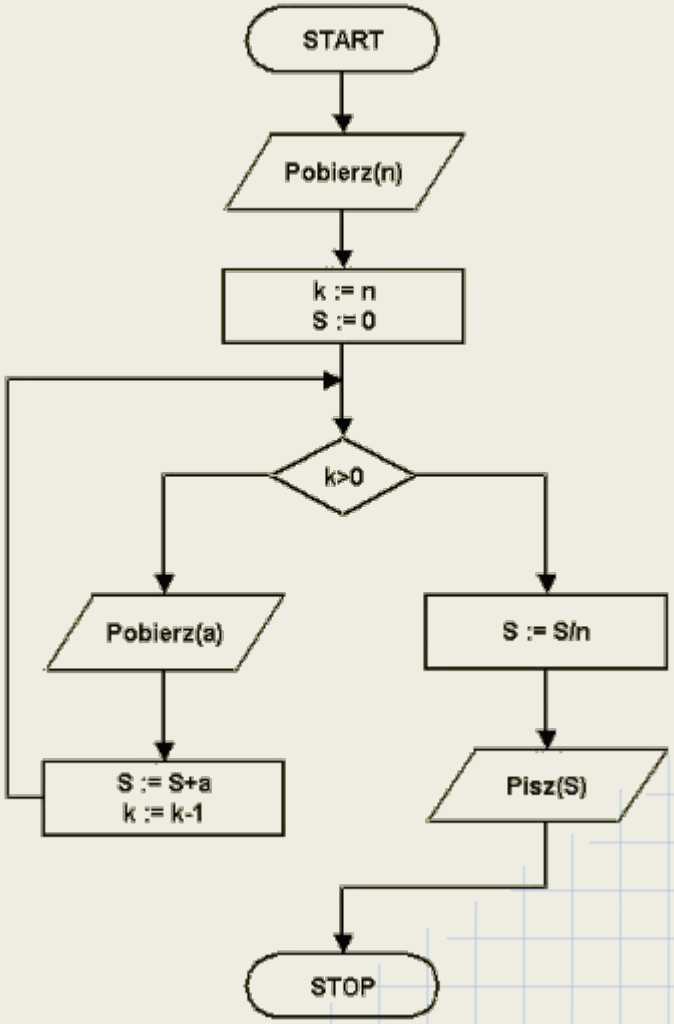
Cel i zakres pracy

Zaprojektowanie i implementacja aplikacji typu *desktop* umożliwiającej tworzenie schematów blokowych i generowanie kodu źródłowego:

- edycja podstawowych elementów strukturalnych schematu (bloki i połączenia) za pomocą GUI (*Wojciech Kwiecień*)
- przechowywanie utworzonych schematów w formie plików (*Wojciech Kwiecień*)
- możliwość uruchomienia (symulacji) schematu (*Mateusz Buczek*)
- generowanie kodu w języku C++ (*Mateusz Buczek*)

Schemat blokowy

- metoda graficznej reprezentacji algorytmu
  - **blok** - figura reprezentująca określony rodzaj lub grupę operacji: start, stop, przetwarzanie, warunek, wejście, wyjście
  - **połączenie** - linia wskazująca przepływ sterowania między blokami
- poglądowo przedstawia sekwencję, rozgałęzienie, zapętlenie - ułatwia analizę
- efektywne wykorzystanie wymaga narzędzi



Przegląd istniejących rozwiązań (1)

JavaBlock

- niekomercyjna aplikacja autorstwa Jakuba Koniecznego
- dostępna na wiele platform (Java)
- obsługa std. bloków: start, stop, wyprowadzenie/wprowadzenie danych, rozgałęzienie, obliczenia
- umożliwia generowanie liniowej grafiki w oparciu o język LOGO ("grafika żółwia")
- nie umożliwia generowania kodu w języku C++

Elbox - Laboratorium Informatyki (ELI)

- komercyjny pakiet programów dydaktycznych firmy ELBOX
- ostatnia wersja wydana w 1997 r. tylko dla systemu Windows
- obsługa std. bloków + specjalne bloki do obsługi tablic i procedur
- składnia instrukcji inspirowana językiem Pascal
- nie umożliwia generowania kodu w języku C++

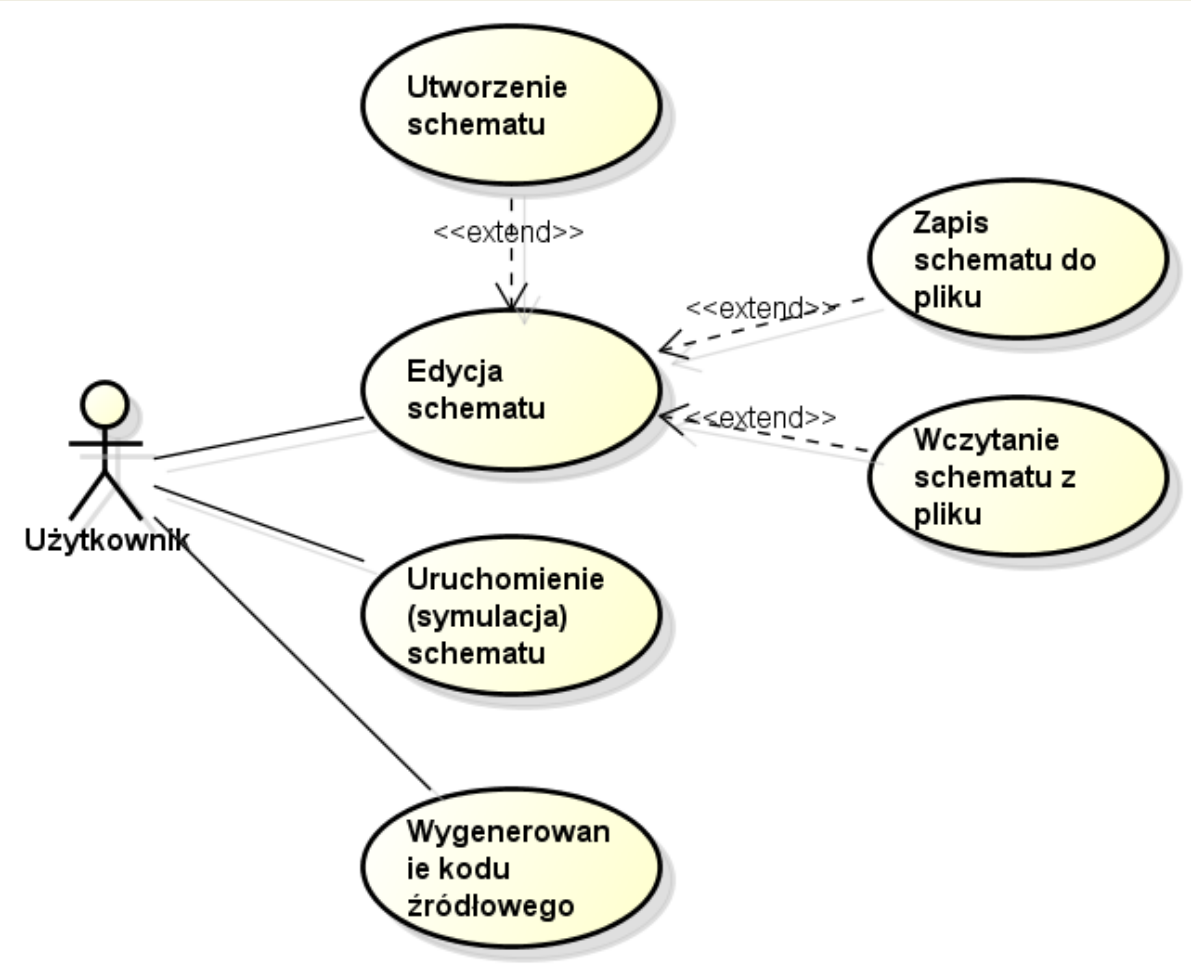
Przegląd istniejących rozwiązań (2)

Microsoft Visual Programming Language (VPL)

- część pakietu Microsoft Robotics Studio
- tylko dla systemów Windows
- model oparty na przepływie danych (*dataflow*)
  - bloki reprezentują aktywności/procesy połączone za pomocą zestawu wejść i wyjść
  - połączenia odpowiadają kierunkom przepływu danych (a nie kierunkowi przepływu sterowania)
  - procesy wykonują się w momencie pojawienia się na wejściu komunikatów (niekoniecznie sekwencyjnie)
  - konieczność reprezentacji wszystkich konstrukcji jako procesów przetwarzających komunikaty (np. przypisanie zmiennej)

Wymagania (1)

Wymagania funkcjonalne



Wymagania (2)

Wymagania niefunkcjonalne

- Przejrzysty i intuicyjny interfejs użytkownika
  - dostęp do wszystkich funkcjonalności
  - obszar roboczy zapewniający komfort edycji
  - respektowanie standardów i konwencji stosowanych w podobnych aplikacjach desktopowych
- Niezależność od systemu operacyjnego
  - wymagane jedynie standardowe środowisko JRE
- Łatwość instalacji i uruchamiania
  - brak instalacji
  - jeden plik wykonywalny

Wykorzystane technologie

Java

- wykorzystana jako język implementacyjny i środowisko wykonawcze
- przenośność między różnymi systemami operacyjnymi
- dostępność odpowiednich bibliotek (np. Swing, JDiagram, JUnit)

Swing

- biblioteka do obsługi GUI

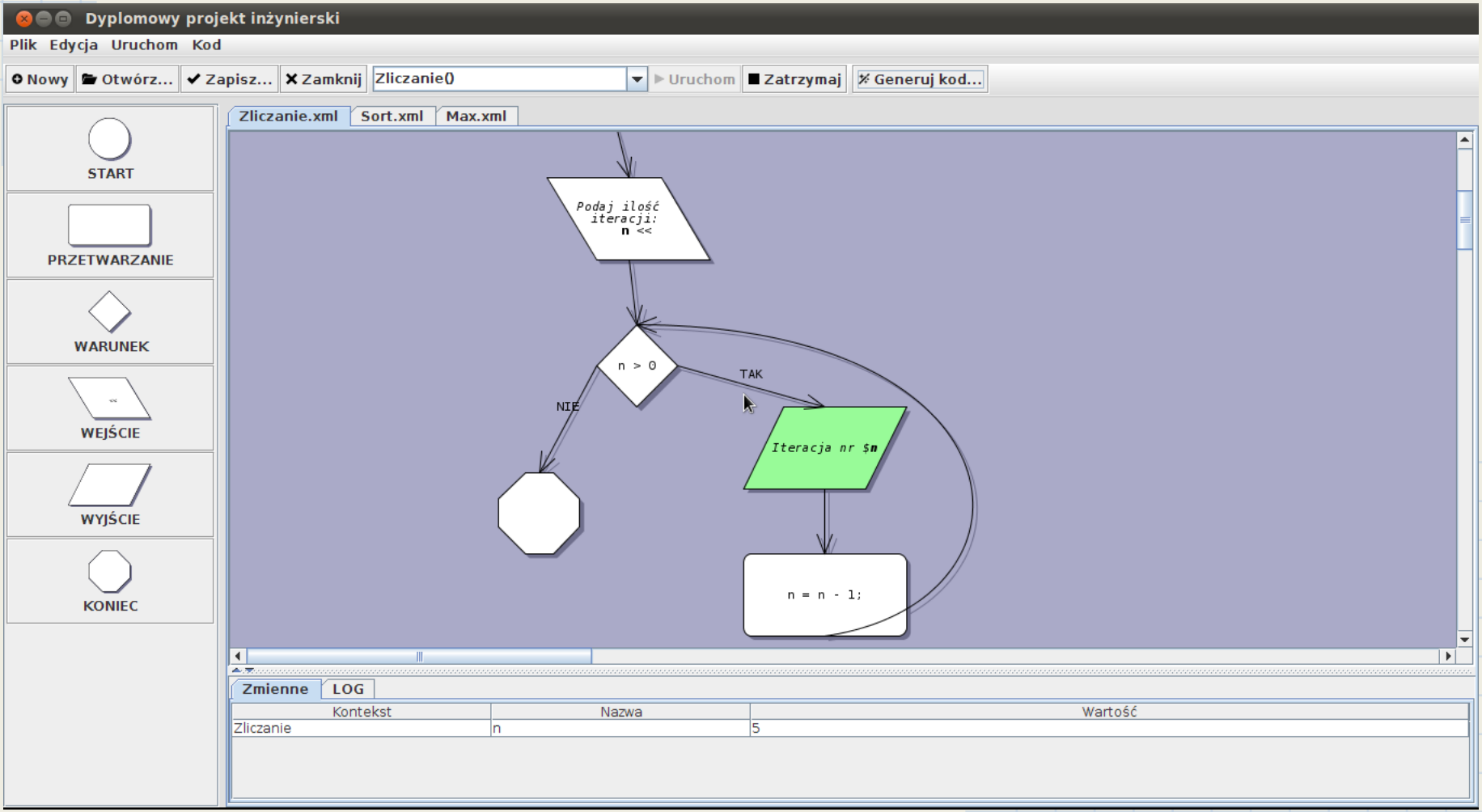
JDiagram

- biblioteka wspomagająca konstruowanie grafów, diagramów i wykresów w oparciu o komponent Swing
- wykorzystana w edytorze schematów blokowych

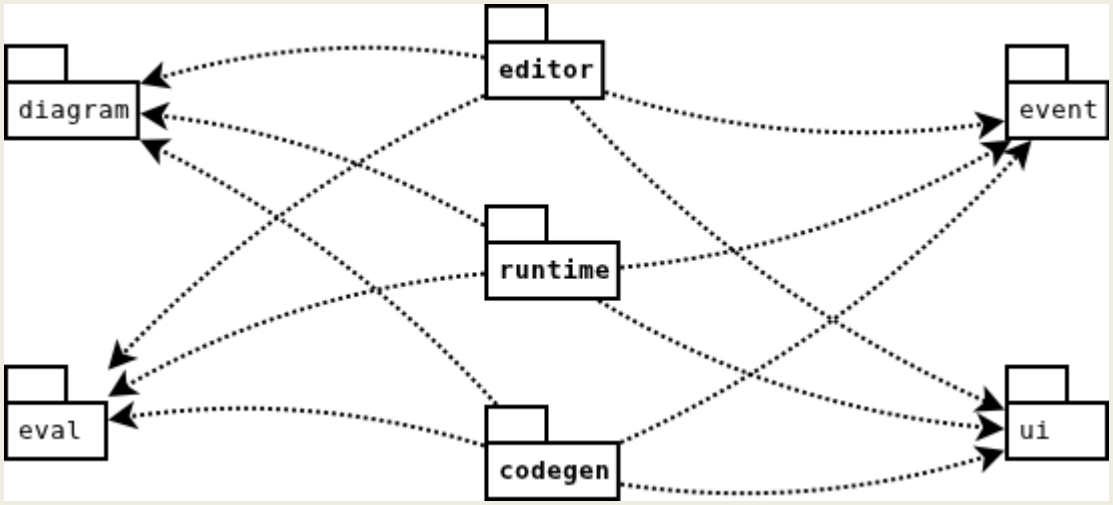
JUnit

- biblioteka to tworzenia testów jednostkowych
- wykorzystana do stworzenia testów dla kluczowych komponentów aplikacji

Interfejs aplikacji



Architektura (1)



- architektura modułowa: każdy pakiet realizuje pewną funkcjonalność:
  - *editor* - tworzenie i edycja schematów, obsługa plików
  - *runtime* - uruchamianie (symulowanie) schematów
  - *codegen* - generowanie kodu źródłowego
  - *diagram* - definicje elementów schematu
  - *eval* - ewaluacja instrukcji w blokach, definicje typów
  - *event* - komunikacja między modułami za pomocą zdarzeń
  - *ui* - komponenty interfejsu użytkownika

Architektura (2)

- moduły *editor*, *runtime* i *codegen* reprezentowane są w głównej klasie aplikacji jako komponenty *DiagramEditor*, *DiagramRunner* i *CodeGeneration*
- komponenty są niezależne od siebie - komunikacja za pomocą zdarzeń (wzorzec projektowy *obserwator*), np.:

```
    // eventBus: instancja EventManager - wspólna dla wszystkich
komponentów

    // DiagramEditor: wywołanie zdarzenia po zamknięciu schematu
    eventBus.triggerEvent(new DiagramClosedEvent(diagram));

    // DiagramRunner: reakcja na zamknięcie schematu
    eventBus.addHandler(DiagramClosedEvent.class,
        new EventHandler<DiagramClosedEvent>() {
            public void handle(DiagramClosedEvent e) {
                // Obsługa zdarzenia...
            }
        }
    );
```

Elementy schematu blokowego - *diagram* (1)

- typ bloku - klasa dziedzicząca z *diagram.BaseNode* definiującą jego atrybuty oraz reprezentację graficzną
- **START**
  - początek sekwencji połączonych bloków ~ początek podprogramu
  - posiada unikalną *nazwę* i opcjonalne *parametry wywołania*
  - pojedynczy schemat może zawierać kilka bloków START
- **KONIEC**
  - koniec sekwencji połączonych bloków ~ koniec podprogramu
  - umożliwia *zwrócenie wartości zmiennej* do miejsca wywołania
- **PRZETWARZANIE**
  - blok obliczeń realizujący podane *instrukcje* (definicje zmiennych, op. arytmetyczne, wywołania bloków START itd.)
- **WARUNEK**
  - blok warunkowy - uzależnienie ścieżki wykonania od zadanego *wyrażenia warunkowego*

Elementy schematu blokowego - *diagram* (2)

- **WEJŚCIE**
  - wprowadzenie danych przez użytkownika
  - wprowadzona wartość *przypisywana jest do podanej zmiennej*
- **WYJŚCIE**
  - wyświetlenie komunikatów i wyników w czasie wykonania
- składnia instrukcji bazuje na *JavaScript*
- deklaracje zmiennych i parametrów bloków START wymagają podania typu
  - `int` - typ całkowity
  - `real` - typ zmiennoprzecinkowy
  - `string` - typ znakowy
  - `bool` - typ logiczny
  - *T*[*n*] - typ tablicowy o *n* elementach typu *T*

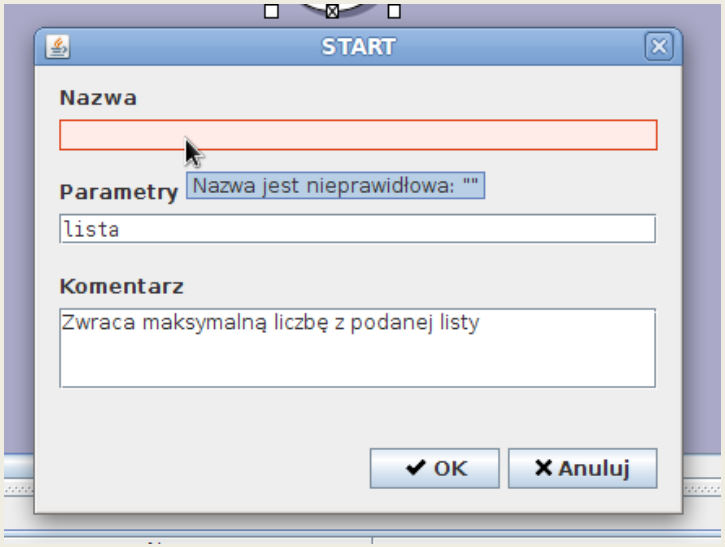
```
var i:int; i = 42;
var t:real[2]; t = [1.5, 0.0];
var tt:string[2][2];
```

Edytor schematów - *editor* (1)

- zarządzanie zakładkami zawierającymi schematy: otwieranie, zamykanie; zapis i odczyt schematu do/z pliku; dodawanie, usuwanie i edytowanie elementów schematu
- funkcje zdefiniowane w postaci akcji - klas dziedziczących z *AbstractAction* biblioteki *Swing* (np. *SaveDiagramAction*)
  - każda akcja posiada swoją nazwę i ikonę
  - jedną akcję można podpiąć do kilku komponentów (przycisków, pozycji menu itp.)
- **Zapis i odczyt schematu do/z pliku**
  - pliki w formacie XML
  - zapisywane są informacje o położeniu i stanie każdego elem. schematu; dla bloków - również dane wprowadzane przez użytkownika
- **Eksport**
  - zapis schematu do pliku graficznego (JPEG, PNG, GIF) lub pliku PDF
- **Cofnij/wykonaj ponownie**
  - każda akcja edycyjna użytkownika jest zapamiętywana

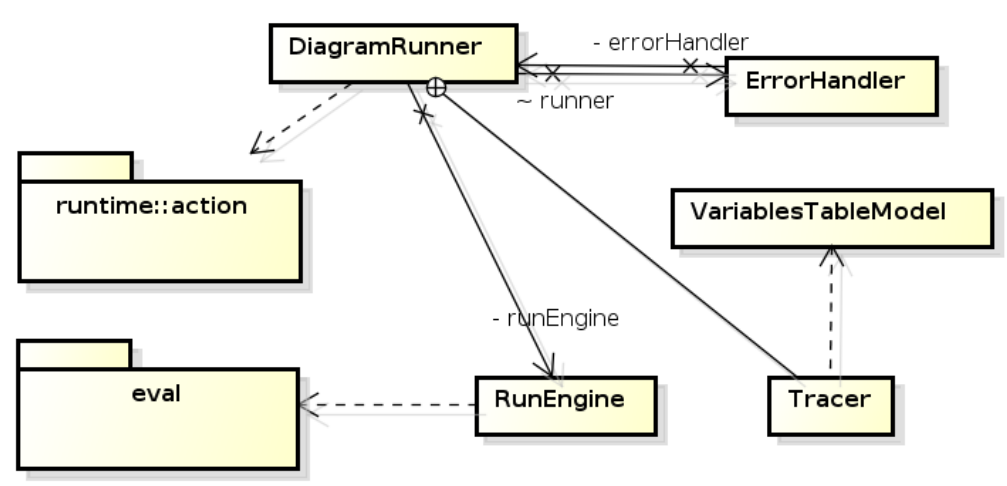
Edytor schematów - *editor* (2)

- **Edycja bloku**
  - każdy rodzaj bloku posiada dedykowany formularz edycji - możliwość modyfikacji atrybutów



  - wartości są sprawdzane pod kątem poprawności

Uruchamianie schematów - *runtime* (1)

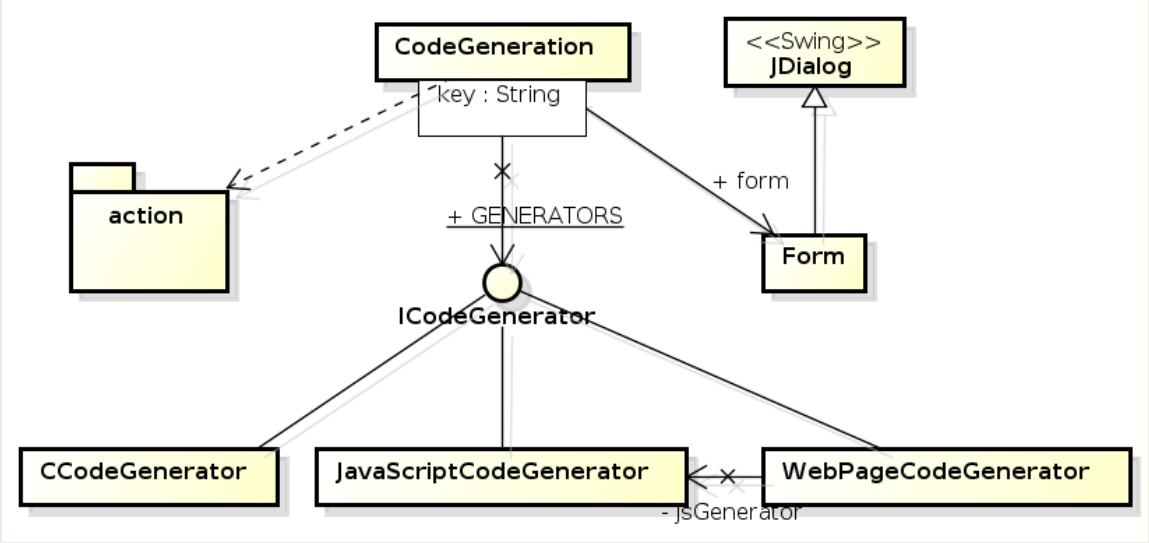


1. "Kompilacja" schematu do kodu *JavaScript* [*eval.Compiler*]
   - każda sekwencja START-KONIEC = funkcja
2. Ewaluacja wybranego (przez użytkownika) bloku START [*eval.EvalEngine*]
   - wywołanie wygenerowanej dla niego funkcji JavaScript
   - wykorzystanie *Java Scripting API* - wbudowany silnik do wykonywania kodu JavaScript z poziomu Javy
   - ewaluacja i wykonanie w osobnym wątku (*RunnerThread*) - nie blokujemy UI

Uruchamianie schematów - *runtime* (2)

3. Wykonanie poszczególnych bloków
   - środowisko/kontekst wykonania (*eval.Environment*) - zbiór dostępnych w czasie wykonania zmiennych i funkcji
     - komunikacja Java <=> JavaScript
   - *$TRACE* - implementacja interfejsu *ITracer*
     - możliwość "wstrzyknięcia" kodu przed i po wykonaniu pojedynczego bloku
     - `$TRACE.before(`*`idx-bloku`*`)`
       zaznaczenie kolorem aktualnie wykonywanego bloku
     - `$TRACE.after(`*`idx-bloku`*`)`
       aktualizacja tabeli z bieżącymi wartościami zmiennych (panel *Zmienne*);
       chwilowe uśpienie wątku wykonania (opóźnienie)

Generowanie kodu - *codegen* (1)



- generator kodu - klasa implementująca interfejs *ICodeGenerator*
  - przeciążone wersje metody *generate* dla każdego typu bloku - każda powinna zwrócić reprezentację bloku w danym języku
  - *generate(Diagram)* - końcowy kod odpowiadający danemy schematowi
- zaimplementowane 3 generatory: C++, JavaScript i HTML
  - dzięki zdef. wspólnego interfejsu można łatwo dodawać kolejne

Generowanie kodu - *codegen* (2)

Generator kodu C++

- każdej sekwencja START-KONIEC => funkcja o nazwie i argumentach jak zdef. w START oraz typie zwracanym zgodnym z KONIEC
- ciało każdej funkcji składa się z kodu wygenerowanego z bloków między START i KONIEC
  - każdy blok odwiedzany jest raz, zgodnie z algorytmem *DFS*
  - WARUNEK => `if (warunek) { ... } [else { ... }]`
  - WEJŚCIE => `std::cin >> zmienna;`
  - WYJŚCIE => `std::cout << "komunikat";`
  - KONIEC => `return zmienna;`
  - przepływ sterowania między blokami => instrukcja *switch* - kod każdego bloku emitowany jako przypadek *case* z indeksem bloku
    - niediomatyczny kod dla pętli (nie wykorzystuje *for* lub *while*)
    - nie wymaga wykrywania cykli i obsługi przypadków tzw. grafów nieredukowalnych (pętle o kilku wejściach)

Generowanie kodu - *codegen* (3)

Generator kodu C++ (cd.)

- C++ jest statycznie typowany - wymaga wygenerowania typów dla zmiennych, argumentów i typów zwracanych funkcji
- mapowanie typów użytych w schemacie na typy C++:

Typ użyty na schemacie	Typ C++
int	*int*
real	*float*
string	*std::string*
bool	*bool*
T[N] (tablica *N* elementów typu *T*)	*std::vector<T'>(N)* (wektor *N* elementów typu *T'*)

- jeśli istnieje blok START o nazwie *Main*, jego wywołanie *Main()* zostanie umieszczone wewnątrz funkcji *main* programu

Testowanie (1)

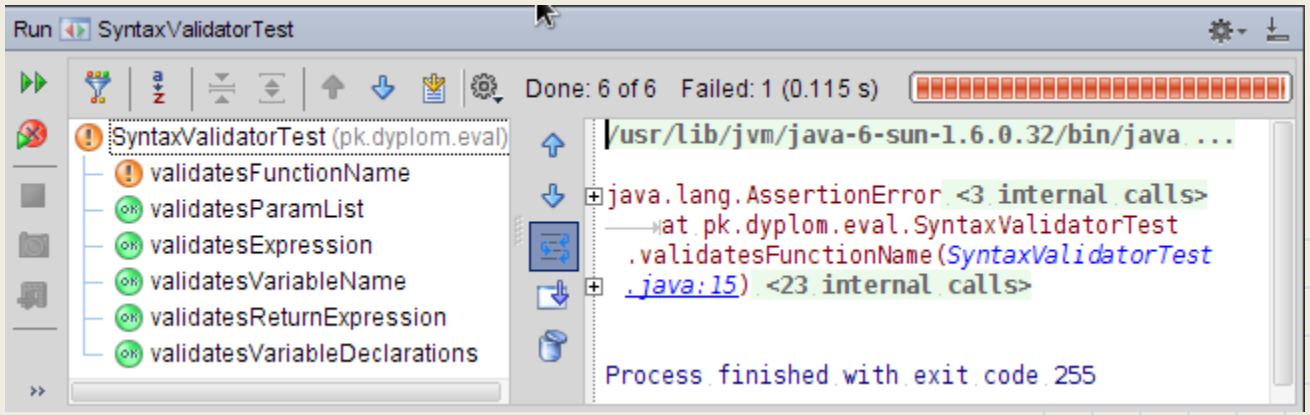
- **Testy manualne**
  - komponenty ściśle współpracujące z GUI lub opierające się na interakcji z użytkownikiem
    - głównie edycja schematów
  - testowanie funkcjonalne (całościowe)
  - sprawdzenie ergonomii i użyteczności (*usability*) interfejsu

- **Testy automatyczne**
  - testy jednostkowe (*unit tests*) dla kluczowych klas (generatory kodu, walidator składni itp.)
  - wykorzystują bibliotekę *JUnit*

Testowanie (2)

- **Testy automatyczne (cd.)**

```
public class SyntaxValidatorTest {
    SyntaxValidator validator = new SyntaxValidator();
    @Test public void validatesFunctionName() {
        assertTrue(validator.isValidFunctionName("foo_bar"));
        assertFalse(validator.isValidFunctionName("2foo"));
        // ...
    }
    // Pozostale przypadki testowe...
}
```



Konfiguracja

- konfiguracja przechowywana w pliku tekstowym (*config/default.properties*)
- prosty format typu *klucz-wartość*:
  - *nazwaZmiennej = wartość*
  - *# oznacza komentarz*

```
# Jezyk (plik *.properties w config/)
lang = pl
# Opoznienie przy wykonywaniu diagramu [int, ms]
runtime.stepDelay = 800
# Kolor aktualnie wykonywanego bloku [hex]
runtime.currentNodeColor = #98FB98
# Kolor bloku zawierajacego blad [hex]
runtime.errorNodeColor = #FE6960
```

Podsumowanie (1)

- Aplikacja pozwala na tworzenie i testowanie schematów blokowych oraz generację kodu.
- Schemat budowany jest ze std. bloków: start, koniec, przetwarzanie, warunek, wejście, wyjście
  - blok START umożliwia definiowanie parametryzowanych podprogramów
  - składnia instrukcji bazuje na JavaScript
  - system typów oferuje 4 typy podstawowe i typ tablicowy
- Schemat przed wykonaniem przekształcany jest do kodu JavaScript
  - podczas wykonania można śledzić wartości zmiennych
- Zaimplementowano 3 generatory kodu: generator C++, JavaScript i HTML
  - dzięki zdef. wspólnego interfejsu dla generatorów można dodawać kolejne
- W celu weryfikacji poprawności wykorzystano zarówno testy manualne jak i automatyczne testy jednostkowe.

Podsumowanie (2)

- Potencjalna możliwość wykorzystania aplikacji
  - pomoc dydaktyczna do nauczania podstaw algorytmiki i programowania
  - narzędzie do szybkiego prototypowania prostych progamów konsolowych w C++

- Możliwości rozwoju:
  - dodanie kolejnych generatorów kodu
  - dodanie panelu umożliwiającego konfigurację poprzez interfejs użytkownika
  - możliwość wstawiania predefiniowanych fragmentów schematu - np. układu oznaczającego pętlę