

UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

Nástroj pro sledování výpočtu v rámci vývojových
diagramů

David Moravčík

Bakalářská práce
2015

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2014/2015

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **David Moravčík**
Osobní číslo: **I11133**
Studijní program: **B2646 Informační technologie**
Studijní obor: **Informační technologie**
Název tématu: **Nástroj pro sledování výpočtu v rámci vývojových diagramů**
Zadávající katedra: **Katedra informačních technologií**

Z á s a d y p r o v y p r a c o v á n í :

Primárním cílem bakalářské práce je návrh, implementace a ověření softwarového nástroje pro podporu sledování evoluce výpočtu v rámci vývojových diagramů.
Pro potřeby konstruování vývojových diagramů bude vyvinuto jednoduché (grafické) vývojové prostředí, v jehož rámci bude následně možné rovněž sledovat evoluci výpočtu.
Vyvinuté programové vybavení může potenciálně představovat prvotní verzi nástroje využitelného na cvičeních z předmětu Základy algoritmizace.
Navržené řešení bude otestováno na sadě vybraných algoritmů.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

PECINOVSKÝ, Rudolf. Návrhové vzory. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.

JONES, Meilir. Základy objektově orientovaného návrhu v UML. Vyd. 1. Praha: Grada, 2001, 367 s. ISBN 80-247-0210-X.

DAMELIO, Robert. The Basics of process mapping. 2nd ed. New York: CRC/Productivity Press, c2011, ix, 173 p. ISBN 9781563273766.

Vedoucí bakalářské práce:

prof. Ing. Antonín Kavička, Ph.D.

Katedra softwarových technologií

Datum zadání bakalářské práce:

20. prosince 2014

Termín odevzdání bakalářské práce:

11. května 2015



prof. Ing. Simeon Karamazov, Dr.
děkan



L.S.



Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 31. března 2015

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 29. 5. 2015



David Moravčík

Poděkování

Touto cestou bych chtěl poděkovat vedoucímu této bakalářské práce, panu prof. Ing. Antonínu Kavičkovi, Ph.D. za všechny návrhy, připomínky a čas, který mi a mé práci věnoval.

Anotace

Tématem práce je návrh a tvorba aplikace s grafickým uživatelským rozhraním pro tvorbu a vizualizaci evoluce vývojových diagramů. Aplikace také umožní export a import vytvořeného diagramu do souboru pro perzistentní uchování. Součástí práce je i testování aplikace na sadě vybraných algoritmů. Aplikace bude implementována v jazyce Java.

Klíčová slova

vývojový diagram, evoluce, aplikace, nástroj, Java

Title

Flowchart evolution tracking tool

Annotation

This thesis is dedicated to design and creation of application with a graphical user interface for creating and tracking evolution of flowcharts. Application will support exporting and importing flowchart to a file for persistent storage. Final part will be dedicated to testing the application on a set of algorithms. Application will be implemented using Java language.

Keywords

flowchart, evolution, application, tool, Java

Obsah

Seznam zkratek.....	10
Seznam obrázků.....	11
Seznam tabulek.....	11
1 Úvod.....	12
2 Základní pojmy.....	13
2.1 Algoritmus.....	13
2.1.1 Základní složky algoritmu.....	13
2.1.2 Algoritmus a data.....	14
2.1.3 Náročnost algoritmů.....	14
2.1.4 Zápis algoritmů.....	15
2.2 Vývojový diagram.....	16
2.2.1 Symboly vývojových diagramů.....	16
2.2.2 Spojnice.....	17
2.3 Orientovaný graf.....	18
2.4 Objektově orientované programování.....	19
2.4.1 Výhody.....	19
2.4.2 Objekt a jeho vlastnosti.....	19
2.4.3 Třída a instance.....	19
2.4.4 Viditelnost a zapouzdření.....	20
2.4.5 Vzájemné vztahy objektů.....	20
2.5 Návrhový vzor.....	21
2.5.1 Návrhový vzor Iterátor.....	21
2.5.2 Návrhový vzor Tovární metoda.....	21
2.5.3 Návrhový vzor Abstraktní továrna.....	22
2.5.4 Návrhový vzor Pozorovatel.....	22
3 Specifikace aplikace.....	23
3.1 Úvod.....	23
3.2 Přehled současných nástrojů.....	23
3.2.1 Nástroj RAPTOR.....	23
3.2.2 Nástroj JavaBlock.....	24
3.3 Model.....	25

3.4	Proměnné	26
3.5	Bloky	26
3.5.1	Blok Start	26
3.5.2	Blok Konec	26
3.5.3	Blok Proces	26
3.5.4	Blok Rozhodování	27
3.5.5	Blok Výstup	27
3.5.6	Blok Vstup	28
3.5.7	Blok Cyklus	28
3.6	Relace	28
3.7	Výrazy	29
4	Implementace	32
4.1	Platforma	32
4.1.1	Java	32
4.1.2	JavaFX	33
4.1.3	NetBeans IDE	33
4.2	Jádro aplikace	34
4.2.1	Průběh evoluce	35
4.2.2	Implementace proměnných	36
4.2.3	Implementace relací	37
4.3	Import/Export modelu	38
4.3.1	XML	38
4.3.2	Postup exportu	39
4.4	Implementace bloků	40
4.5	Implementace výrazů	40
4.5.1	Postup vyhodnocení výrazu	41
5	Testování na vybraných algoritmech	42
5.1	Algoritmus Max	42
5.2	Algoritmus Schody	43
5.3	Algoritmus Bubble Sort	44
5.4	Algoritmus pro výpočet transponované čtvercové matice	45
	Závěr	46
	Literatura	47

Příloha A – Systémové požadavky	49
Příloha B – Uživatelská příručka	50
Příloha C – Obsah přiloženého CD.....	56

Seznam zkratk

XML	Extended Markup Language
HTML	HyperText Markup Language
UML	Unified Modeling Language
OOP	Objektově Orientované Programování
IDE	Integrated Development Environment
JVM	Java Virtual Machine
GUI	Graphic User Interface
CSS	Cascading Style Sheets

Seznam obrázků

Obrázek 1 – Příklad vstupu, výstupu a spojení datového toku.....	17
Obrázek 2 – Příklad obecného vývojového diagramu.....	18
Obrázek 3 – Orientovaný graf.....	18
Obrázek 4 – Schématické znázornění objektu.....	19
Obrázek 5 – UML diagram: Iterátor.....	21
Obrázek 6 – UML diagram: Pozorovatel.....	22
Obrázek 7 – Aplikace RAPTOR.....	24
Obrázek 8 – Aplikace JavaBlock.....	25
Obrázek 9 – Schéma modelu v průběhu evoluce.....	25
Obrázek 10 – Blok Start.....	26
Obrázek 11 – Blok Konec.....	26
Obrázek 12 – Blok Proces.....	27
Obrázek 13 – Blok Rozhodování.....	27
Obrázek 14 – Blok Výstup.....	28
Obrázek 15 – Blok Vstup.....	28
Obrázek 16 – Blok Cyklus.....	28
Obrázek 17 – Zobrazení relace mezi dvěma bloky.....	29
Obrázek 18 – Prostředí NetBeans IDE ve verzi 8.0.2.....	34
Obrázek 19 – Změny stavů modelu.....	35
Obrázek 20 – UML diagram: Jádro aplikace.....	35
Obrázek 21 – Vizualizace stavu proměnných.....	37
Obrázek 22 – Vizualizace relace.....	38
Obrázek 23 – Algoritmus Max.....	42
Obrázek 24 – Algoritmus Schody.....	43
Obrázek 25 – Algoritmus Bubble Sort.....	44
Obrázek 26 – Algoritmus pro prohození prvků matice podle hlavní diagonály.....	45
Obrázek 27 – Hlavní okno aplikace.....	50
Obrázek 28 – Panel proměnných.....	51
Obrázek 29 – Dialog pro editaci proměnných.....	52
Obrázek 30 – Přidání bloku do modelu.....	53
Obrázek 31 – Dialog pro editaci bloku Proces.....	53
Obrázek 32 – Postup vytvoření relace.....	54
Obrázek 33 – Panel nástrojů.....	54

Seznam tabulek

Tabulka 1 – Přehled základních tříd složitosti.....	15
Tabulka 2 – Přehled základních symbolů podle ČSN ISO 5807.....	16
Tabulka 3 – Přehled podporovaných porovnávacích operátorů.....	27
Tabulka 4 – Přehled podporovaných aritmetických operátorů.....	29
Tabulka 5 – Příklady práce s proměnnými.....	30
Tabulka 6 – Přehled podporovaných funkcí.....	30

1 Úvod

Cílem práce je navrhnout softwarový nástroj pro vizualizaci evoluce v rámci vývojových diagramů. Za tímto účelem bude navrženo jednoduché grafické rozhraní, umožňující návrh vývojového diagramu a sledování jeho stavu při průběhu daným algoritmem. Současně je třeba umožnit interakci uživatele v průběhu evoluce a jednoduchý výstup pro výpis aktuálního stavu. Dále aplikace umožní uložit navržený vývojový diagram do souboru pro jeho trvalé uchování nebo sdílení.

Aplikace je především určena pro návrh a testování algoritmů z oblasti algoritmizace a sekvenčního programování bez nutné znalosti programovacího jazyka. Nástroj může v budoucnu sloužit i jako podpora pro výuku předmětu Základy Algoritmizace.

Účelem aplikace není být plnohodnotnou náhradou programovacího jazyka a ani tvorba samostatných programů. Aplikace se soustředí především na algoritmy z oblasti programování a práci s číselnými proměnnými. V praxi lze obecně vývojové diagramy aplikovat i na procesy, které s výpočetní technikou nesouvisí.

Aplikace bude implementována v programovacím jazyce Java od společnosti Oracle. K vývoji uživatelského rozhraní bude využit Framework JavaFX.

Součástí práce je také ověření funkce aplikace na sadě vybraných algoritmů. Modely těchto algoritmů budou součástí elektronické přílohy.

2 Základní pojmy

Tato část je věnována vybraným pojmům především z oblasti algoritmizace a objektivě orientovaného programování.

2.1 Algoritmus

Algoritmus je elementární matematický pojem. Jedná se v podstatě o návod, jak provést určitou činnost. V případě programování jde zpravidla o transformaci množiny vstupních dat na množinu dat výstupních. Aby bylo možné označit návod za algoritmus, musí mít následující vlastnosti:

1. **Elementárnost:** Skládá se z konečného počtu elementárních, snadno realizovatelných činností, které lze označit jako kroky.
2. **Determinovanost:** Po každém kroku lze určit, zda daný proces skončil, nebo kterým krokem má algoritmus pokračovat.
3. **Konečnost:** Počet opakování jednotlivých kroků algoritmu je vždy konečný. Algoritmus jako celek musí skončit po konečném množství kroků.
4. **Rezultativnost:** Algoritmus vede ke správnému výsledku.
5. **Hromadnost:** Algoritmus lze použít k řešení celé skupiny podobných úloh.

Pro označení objektu, který bude algoritmem popisovanou činnost provádět, se používá termín procesor. Tímto procesorem může být stroj, člověk, nebo například celý systém lidí a strojů. Při formulaci algoritmu je tedy vždy nutné znát procesor a formu elementárních kroků, které může návod obsahovat. (VIRIUS, 1995)

Neexistuje algoritmus, který by o jakémkoli jiném algoritmu prohlásil, zdali vede ke správnému výsledku a zdali je konečný.

2.1.1 Základní složky algoritmu

Algoritmy se skládají z tří základních konstrukcí, které se označují jako posloupnost (sekvence), cyklus (iterace) a podmíněná operace (selekce, podmínka).

Posloupnost (sekvence)

Posloupnost je tvořena jedním, nebo několika kroky, které se provedou v daném pořadí. Nemusí se nutně jednat o kroky elementární. Sekvencí může být například i vnořený algoritmus¹ tvořený sekvencemi, cykly a podmínkami.

Podmíněná operace (selekce, podmínka)

Podmínka představuje vždy větvení algoritmu. Je tvořena dvěma nebo více výběrovými složkami a podmínkou, která určí následující krok algoritmu. Podmínka je nezbytnou součástí cyklu a zajišťuje jeho konečnost.

¹ V praxi často označován jako dílčí algoritmus nebo podprogram

Cyklus (iterace)

Cyklus je částí algoritmu, která se opakuje, dokud je splněna podmínka opakování. Cyklus se vždy skládá z podmínky opakování a těla cyklu. Tělo cyklu je složeno z operací, které se opakují. Tyto operace mohou být tvořeny sekvencí, nebo dalšími tzv. vnořenými cykly.

Podmínku lze vyhodnocovat před provedením těla cyklu. V takovémto případě se jedná o cyklus s podmínkou na začátku. Tento nejběžnější typ cyklu se v závislosti na podmínce opakování nemusí nikdy provést. Dalším typem cyklu je cyklus s podmínkou na konci. U tohoto cyklu se podmínka opakování vyhodnocuje vždy po provedení těla cyklu. U tohoto cyklu se tělo vždy provede alespoň jednou. Posledním typem je cyklus s podmínkou v těle cyklu. Použití tohoto typu se zpravidla nedoporučuje, nemusí totiž být programovacím jazykem podporován². (VIRIUS, 1995)

2.1.2 Algoritmus a data

Základním účelem algoritmu je transformovat množinu vstupních dat na množinu dat výstupních. Povaha vstupních, výstupních a především vnitřních³ dat tak výrazně ovlivňuje strukturu algoritmu. Při zpracování vstupních dat tak bude posloupnosti datových položek různých druhů odpovídat posloupnost odpovídajících příkazů. Pokud se datové položky téhož druhu opakují, bude jejich zpracování zpravidla vyžadovat použití cyklu. V místě kde se mohou vyskytnout datové položky různého druhu, využijeme zejména podmíněného větvení.

Důležitou vlastností algoritmu ve vztahu k datům je jeho neměnnost. Na rozdíl od dat se algoritmus nemění a zůstává konstantní i při opakovaném použití.

2.1.3 Náročnost algoritmů

Při analýze algoritmů je důležitá nejen správnost, příp. přesnost algoritmu, ale i náročnost na procesor, pro který je algoritmus určen. V případě, že algoritmus bude realizován počítačovým programem, lze hovořit o časové náročnosti a množství operační paměti, kterou bude program vyžadovat.

Při hodnocení časové náročnosti je nutné vycházet zejména z počtu elementárních kroků, které bude procesor vykonávající algoritmus provádět. Časová náročnost algoritmu tak stoupá zejména při využívání vnořených cyklů. (VIRIUS, 1995)

Ke klasifikaci algoritmů se obvykle používá tzv. asymptotický odhad složitosti, což je rozdělení algoritmů do tříd složitostí, u kterých platí, že od určité velikosti dat, je algoritmus dané třídy vždy pomalejší než algoritmus třídy předchozí, bez ohledu na to, jestli je některý z počítačů c-násobně výkonnější (c je konstanta).

Škála v nekonečnu slouží k rozlišení jednotlivých tříd. Říká, že pokud se objem vstupních dat blíží k nekonečnu, tak neexistuje reálná konstanta taková, aby byl algoritmus z vyšší třídy rychlejší než ten z třídy přecházející. (MIČKA, 2011)

² Např. v jazyce Java vytvoříme tento typ cyklu pomocí klíčového slova `break` v jiném cyklu

³ Nekonzistentní stav dat za běhu algoritmu

Následující tabulka (Tabulka 1) obsahuje přehled nejčastějších tříd složitosti.

Tabulka 1 – Přehled základních tříd složitosti.

Asymptotická složitost	Vyjádření	Popis
konstantní	$O(1)$	Počet operací je pro libovolně velká vstupní data zhruba stejný. Typicky se jedná o nějaký jednoduchý matematický výpočet, jeden přístup k paměti, apod.
logaritmická	$O(\log n)$	Typicky ideální případ vyhledávání ve stromě s n prvky.
lineární	$O(n)$	Náročnost algoritmu se zvyšuje podobně jako velikost vstupních dat. Typicky jeden průchod polem. Některé algoritmy s touto složitostí mohou být implementovány i proudově.
lineárně logaritmická	$O(n * \log(n))$	Typická složitost rozumného řadícího algoritmu.
kvadratická	$O(n^2)$	Náročnost algoritmu roste jako druhá mocnina velikosti vstupních dat. Typicky průchod všech dvojic v poli.
polynomiální	$O(n^k), k \in R$	
exponenciální	$O(k^n), k \in R$	
faktoriálová	$O(n!)$	Typicky vyhodnocování všech možných permutací n prvků, například v tzv. brute-force algoritmech.

(HORDEJČUK, © 2008)

2.1.4 Zápis algoritmů

Každý algoritmus musí být natolik srozumitelný, aby podle něj procesor, pro který je určen dokázal postupovat. V případě počítačového programu je tímto procesorem programátor, který bude na základě tohoto algoritmu vyvářet funkční program. Pro zápis algoritmů existuje několik způsobů.

- **Slovní vyjádření:** Je například návod nebo recept, obvykle psaný formou srozumitelnou pro člověka. V oblasti programování se obvykle využívá tzv. pseudokód, který více odpovídá konstrukcím programovacích jazyků.
- **Matematický zápis:** Jednoznačný zápis pomocí matematických vzorců nebo rovnic.
- **Rozhodovací tabulky:** Vhodné především pro zápis vícenásobného větvení. Nevhodné pro zápis cyklů.
- **Programovací jazyk:** Je jazyk srozumitelný jak pro člověka, tak pro počítač. Nevýhodou bývá nižší přehlednost a nutná znalost konkrétního programovacího jazyka.
- **Vývojové diagramy:** Jsou grafickým (schématickým) znázorněním daného algoritmu. Podrobně popsány v následující kapitole.

2.2 Vývojový diagram

Vývojový diagram (angl. flowchart) je symbolický programovací jazyk sloužící pro grafické znázornění algoritmu nebo obecného procesu. Jedná se o univerzální formu zápisu nezávislou na konkrétním programovacím jazyku. Často se využívá ke komunikaci mezi programátory a analytiky a za účelem dokumentace.


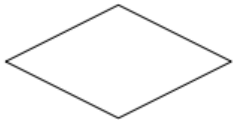

K zakreslení vývojového diagramu se používají standardizované symboly podle normy ČSN ISO 5807 Zpracování informací.


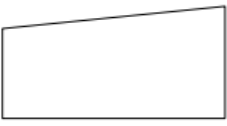

2.2.1 Symboly vývojových diagramů

Symboly vývojových diagramů představují grafické značky normou definovaného významu. Pro specifikaci funkce symbolů se do nich vpisují slovní nebo symbolické operace, příp. celé skupiny operací. Symbolika tohoto textového zápisu není normou nijak specifikována a závisí především na povaze algoritmu a procesoru, pro který je určen. Při zápisu matematických výrazů se však výrazně doporučuje vyhýbat se používání symboliky konkrétního programovacího jazyka a využívat především značky definované normou ČSN ISO 31-11 (Veličiny a jednotky – Část 11: Matematické značky a značky používané ve fyzikálních vědách a v technice). Zajistí se tím obecnost daného vývojového diagramu.

Následující tabulka (Tabulka 2) obsahuje přehled vybraných symbolů.

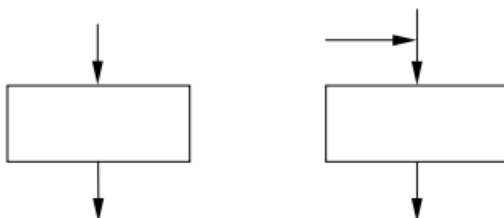
Tabulka 2 – Přehled základních symbolů podle ČSN ISO 5807.

Název	Symbol	Popis
zpracování		Symbol představuje jakýkoli druh zpracování nebo provedení dané operace příp. skupiny operací, jejímž výsledkem je transformace informace (např. měna hodnoty). Symbol může obsahovat libovolné množství vstupů, avšak vždy právě 1 výstup.
rozhodování		Symbol představuje rozhodovací nebo přepínací funkci a realizuje větvení algoritmu. Symbol zpravidla obsahuje 1 vstup a alespoň 2 alternativní výstupy. Na základě vyhodnocení podmínky uvnitř symbolu je vždy zvolen právě 1 výstup.
příprava		Symbol představuje úpravu nebo modifikaci činnosti, která upravuje postup činnosti následné (např. vyjmenování hodnot, kterých nabývá proměnná cyklu). Symbol má dva vstupy, jeden sekvenční, druhý pro návrat po provedení příslušného bloku operací a dva výstupy, jeden vstupující do daného bloku operací, druhý sekvenční, který pokračuje do další části algoritmu.

Název	Symbol	Popis
vstup a výstup dat		Symbol reprezentuje vstupně-výstupní operace s daty, tj. dodání dat pro zpracování nebo výstup dat v požadované formě. Druh vstupního nebo výstupního zařízení není jednoznačně určen. V případě počítačového programu se jedná především o zobrazení výstupu na obrazovku. Symbol má zpravidla 1 vstup a 1 výstup.
ruční vstup		Symbol představuje zařízení pro ruční (interaktivní) vstup informací jako například klávesnice, snímače, příp. různé přepínače. Symbol má vždy 1 výstup.
mezí značka		Symbol představuje vstup z vnějšího prostředí do programu nebo výstup z programu do vnějšího prostředí, např. začátek a konec programu. Symbol má vždy 1 vstup nebo 1 výstup.

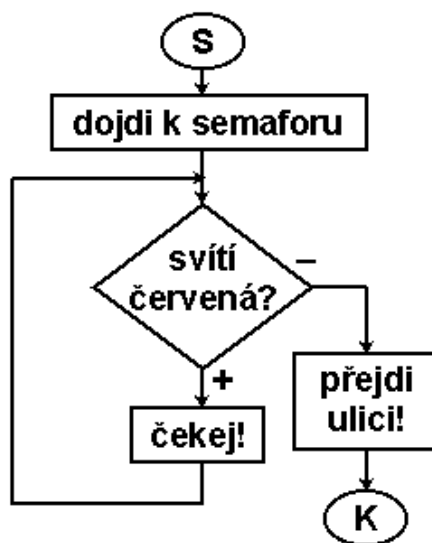
2.2.2 Spojnice

Spojnice ve tvaru svislých nebo vodorovných čar představují tok dat nebo řízení a slouží k propojení jednotlivých symbolů vývojového diagramu. Pro znázornění směru toku dat se spojnice zpravidla opatřují plnou nebo otevřenou šipkou. Spojnice se mohou podle potřeby spojovat. Pro zvýšení jasnosti by měl být u připojovaných spojníc uveden směr toku informací plnou nebo otevřenou šipkou. (viz. Obrázek 1)



Obrázek 1 – Příklad vstupu, výstupu a spojení datového toku.

Pro znázornění výstupu symbolu rozhodování s binární podmínkou lze využít symbolu plus, je-li podmínka splněna a symbolu mínus v případě opačném. (viz. Obrázek 2)



Obrázek 2 – Příklad obecného vývojového diagramu.

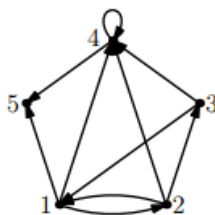
Větvení datového toku je pak možné pouze pomocí k tomu určených symbolů (např. symbol rozhodování). Tím se zajistí determinovanost daného algoritmu.

2.3 Orientovaný graf

Z hlediska datových struktur lze vývojový diagram považovat za jistý druh orientovaného grafu, kde jednotlivé symboly představují vrcholy daného grafu a spojnice představují orientované hrany.

Definice:

Orientovaný graf je dvojice (V, E) , kde V je množina vrcholů a $E \subset V \times V$ je množina hran.



Obrázek 3 – Orientovaný graf.

Na rozdíl od obecných orientovaných grafů však vývojové diagramy nepřipouští smyčky, tedy hrany vedoucí do stejného vrcholu, ze kterého vychází. (viz. Obrázek 3 – hrana z vrcholu 4 do vrcholu 4) (ČADA, a další, 2004)

2.4 Objektově orientované programování

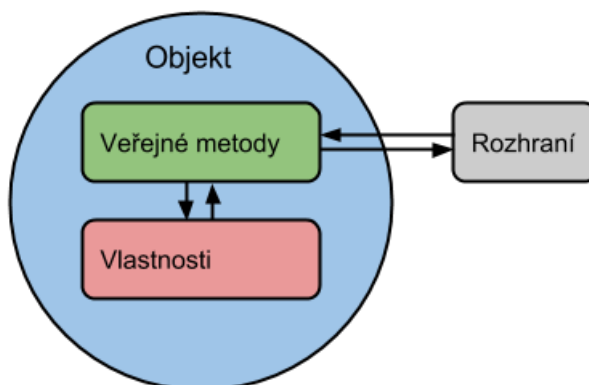
Objektově orientované programování (OOP) je programovací paradigma. Jeho cílem je tvorba malých, relativně samostatných a opakovaně použitelných jednotek, které nazýváme „objekty“.

2.4.1 Výhody

Předností OOP je především snazší a rychlejší vývoj aplikací, snazší údržba a obecně menší chybovost. Potenciální nevýhodou je správa objektů, která zabírá určité systémové prostředky. Program napsaný podle OOP bude tedy vždy pomalejší, než dokonale napsaný procedurální program. Výhody především ve snadnosti údržby a rozšiřitelnosti však veškeré nevýhody OOP značně převyšují.

2.4.2 Objekt a jeho vlastnosti

Objekt je soběstačná entita, která obsahuje jak data, tak i funkce pro práci s těmito daty. Objekt si tedy uchovává svůj vnitřní stav ve formě vlastností, které se v terminologii OOP nazývají atributy. Dále objekt obsahuje funkce, které s těmito vlastnostmi manipulují. Ty se v terminologii OOP označují jako metody.



Obrázek 4 – Schématické znázornění objektu.

2.4.3 Třída a instance

Třída je množina objektů s určitými vlastnostmi. Sama třída žádné konkrétní objekty nedefinuje, ale udává, jaké vlastnosti bude objekt dané třídy mít.

Instance je již konkrétní objekt dané třídy. Instance mají svůj vnitřní stav, nezávislý na jiných instancích dané třídy. Pokud tedy za třídu budeme považovat např. osobní automobil, pak instancemi této třídy budou jednotlivé modely různých výrobců.

Třída je tedy jistým druhem návodu, jak objekt vytvořit a jak s ním nadále pracovat. Třída je tím, co vytváří programátor, její instance (objekty) se vytváří až za běhu programu. K vytvoření instancí slouží speciální druh metod, které nazýváme konstruktory. Název konstruktorů se obvykle shoduje s názvem třídy a odvoláváme se na ně operátorem „new“.

Třída může mít i své atributy a metody, které se označují jako statické. Ty zpravidla nemohou pracovat s instancemi dané třídy, neboť žádné nemusí existovat. Vytvořené objekty však k těmto statickým vlastnostem přistupovat mohou. Výhodou těchto statických prvků je, že k nim lze přistoupit i bez vytváření instance dané třídy. (MENCL, 2012)

2.4.4 Viditelnost a zapouzdření

Některé vlastnosti objekt potřebuje jen ke svému internímu využití. Je tedy nežádoucí, aby tyto vlastnosti používal někdo (např. jiný objekt) „zvenčí“. Proto rozlišujeme vlastnosti veřejné (angl. public), které lze využívat „zvenčí“, a soukromé (angl. private), které mohou používat jen jiné metody téhož objektu.

Koncept zapouzdření spočívá v odkrytí jen těch metod a atributů, které jsou pro práci s objektem nezbytné. Interní logika daného objektu zůstává pro okolí skryta. Objekt si tedy lze představit jako černou skříňku (angl. blackbox), která má určité rozhraní přes které jí předáváme instrukce nebo data a ona je zpracovává. Tento přístup zajišťuje nižší chybovost a lepší znouvopoužitelnost dané třídy. Programátor, který s takovou třídou pracuje, pak musí znát pouze její veřejné rozhraní.

2.4.5 Vzájemné vztahy objektů

Existuje několik druhů vztahů, které mohou objekty utvářet.

- **Skládání:** Každý objekt může obsahovat další „vnořené“ objekty. Jak vnější, tak i vnitřní objekt mohou přistupovat pouze k veřejným složkám toho druhého.
- **Delegování:** Objekt může využít veřejných služeb jiných objektů.
- **Dědičnost:** Představuje stromovou strukturu, kdy objekt může dědit vlastnosti jiného objektu.
- **Polymorfismus:** Skupina objektů poskytujících stejné rozhraní, které se od sebe liší interní implementací. Příkladem mohou být třídy reprezentující geometrické tvary disponující metodou pro výpočet obsahu. V programu potom můžeme zjistit obsah daného tvaru, aniž bychom v daném okamžiku věděli, o jaký konkrétní tvar se jedná.

Dědičnost:

Dědičnost umožňuje vytvářet hierarchii mezi třídami. Třída výše v hierarchii se nazývá předek (nadtřída, angl. superclass) a třída níže v hierarchii potomek (podtřída, angl. subclass). Většina programovacích jazyků využívá pouze jednoduchou dědičnost⁴. To znamená, že třídy mohou mít nejvýše jednoho předka.

Dědičnost tedy umožňuje:

1. Doplnit předka o dodatečné vlastnosti a funkce.
2. Změnit implementaci stávajících funkcí předka.
3. Vyhnout se duplicitám v kódu a výrazně tím usnadnit jeho údržbu.

⁴ Patří mezi ně i jazyk Java

Dědičnost je také polymorfismem. Na základě rozhraní předka můžeme tedy přistupovat k třídám jeho potomků.

2.5 Návrhový vzor

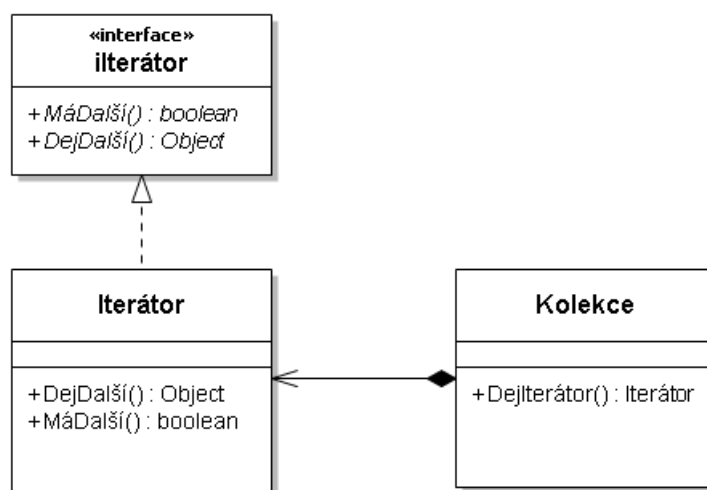
Jednou z možností tvorby objektově orientovaných aplikací je využití tzv. návrhových vzorů (angl. design pattern). Návrhový vzor v softwarovém inženýrství představuje obecné řešení problému, které se využívá při návrhu počítačových programů. Nejde přitom o knihovnu nebo část zdrojového kódu, ale o šablonu, která může být použita na řešení určitého typu problémů.

Návrhové vzory mohou výrazně urychlit vývoj softwaru s využitím prověřených postupů řešení. Efektivní návrh softwaru vyžaduje zvážení potencionálních problémů, které by mohly později během vývoje nastat. Využívání návrhových vzorů pomáhá předcházet těmto problémům a zlepšuje přehlednost kódu pro programátory, kteří tyto návrhové vzory znají. (PECINOVSKÝ, 2007)

Následuje popis vybraných návrhových vzorů využitých při tvorbě aplikace.

2.5.1 Návrhový vzor Iterátor

Iterátor zajišťuje možnost procházení prvků datových struktur bez nutné znalosti jejich implementace. Iterátor je rozhraní poskytující jednotné metody pro procházení datových struktur, které toto rozhraní implementují.



Obrázek 5 – UML diagram: Iterátor

2.5.2 Návrhový vzor Tovární metoda

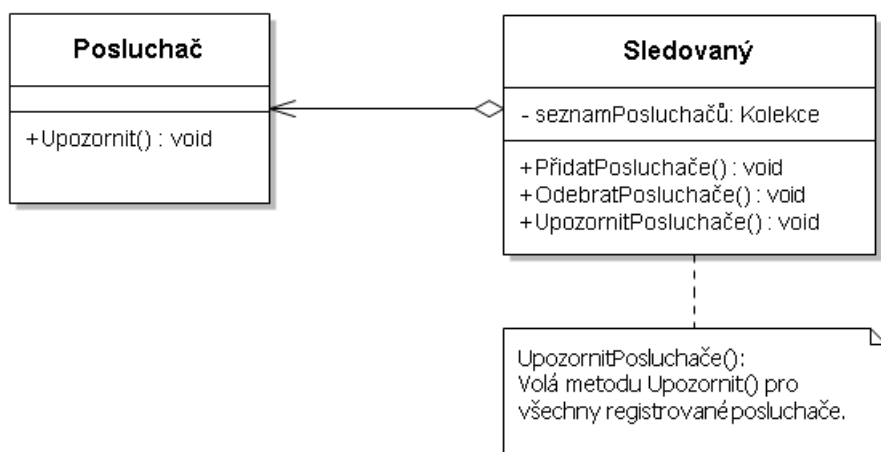
Tovární metoda je statická metoda sloužící výhradně k vytvoření instance dané třídy. Tuto instanci metoda vytvoří pomocí neveřejného konstruktoru. Tento přístup poskytuje oproti přímému využití konstrukturu několik výhod. Jednou z výhod je možnost takovouto metodu pojmenovat. Další výhodou je možnost omezit rozsah vlastností vytvářených instancí podle potřeb dané aplikace.

2.5.3 Návrhový vzor Abstraktní továrna

Abstraktní továrna je návrhový vzor, který umožňuje „obalit“ skupinu tříd využívajících tovární metody. Tím získáme jednotné rozhraní pro tvorbu instancí těchto tříd. Klient se tedy již nestará o to, které konkrétní objekty vytváří, ale využívá k tomu rozhraní abstraktní továrny. Jedná se tedy o polymorfismus, umožňující tvorbu různých typů objektů jednotným způsobem.

2.5.4 Návrhový vzor Pozorovatel

Návrhový vzor pozorovatel (angl. observer pattern) umožňuje „sledovanému“ objektu vytvářet si seznam objektů na něm závislých (posluchačů). Každá změna sledovaného objektu tak způsobí notifikaci objektů na něm závislých. Tím umožníme posluchačům na tyto změny reagovat (např. aktualizovat zobrazenou hodnotu). Tento návrhový vzor využívají zejména moderní platformy pro vývoj aplikací s grafickým uživatelským rozhraním, jakou je například i platforma JavaFX. Tento přístup umožňuje oddělit aplikační data od prezentační vrstvy.



Obrázek 6 – UML diagram: Pozorovatel

3 Specifikace aplikace

Následující kapitola je věnována specifikaci požadavků na vyvíjenou aplikaci.

3.1 Úvod

Účelem vyvíjené aplikace je poskytnout grafické prostředí pro tvorbu vývojových diagramů. Následně je třeba umožnit „spuštění“ těchto diagramů a sledování jejich evoluce. Dále je třeba poskytnout mechanismus pro uložení a načtení těchto vývojových diagramů ze souboru.

Aplikace je určena především pro podporu studia předmětu Algoritmizace, za účelem návrhu a testování vývojových diagramů bez nutných znalostí programovacího jazyka. Účelem aplikace není být plnohodnotnou náhradou programovacího jazyka nebo tvorba samostatných programů.

3.2 Přehled současných nástrojů

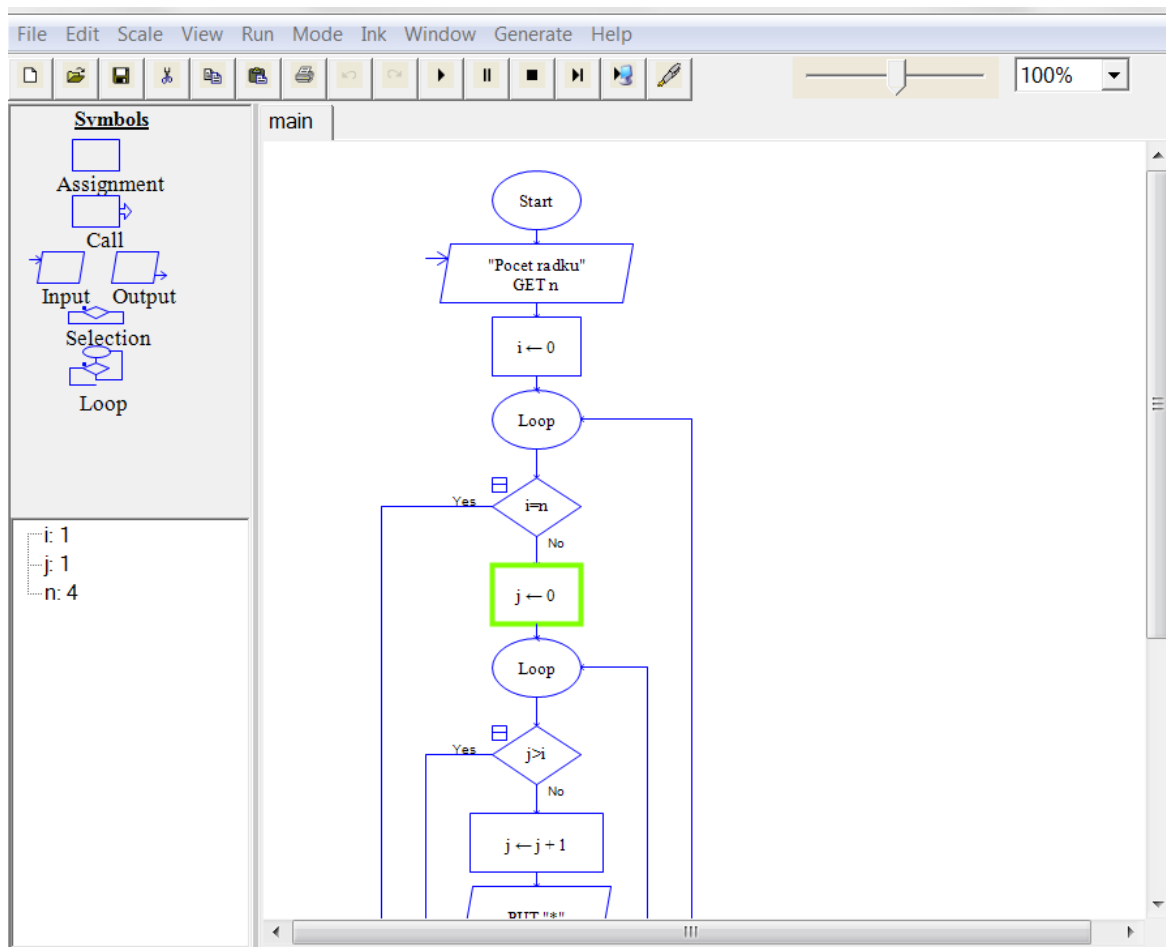
V současné době existuje několik nástrojů zabývajících se návrhem a vizualizací evoluce vývojových diagramů.

- **RAPTOR** – Volně dostupný nástroj napsaný v jazyce C#. Plně podporován pouze na systémech Windows.
- **JavaBlock** – Volně dostupný nástroj napsaný v Javě.
- **Flowcharts Interpreter** – Volně dostupný nástroj. Výsledné diagramy interpretuje v C++ kódu.
- **Free DFD** – Volně dostupný nástroj pro Windows a UNIX (Linux).

3.2.1 Nástroj RAPTOR

RAPTOR je volně šiřitelný nástroj původně vyvinutý pro oddělení počítačových věd Americké letecké akademie (United States Air Force Academy). Program disponuje třemi režimy. Základní režim s globálním prostorem pro všechny proměnné, pokročilý režim s možností tvorby procedur s vlastním prostorem pro proměnné a objektově orientovaný režim umožňující návrh hierarchie tříd pomocí UML diagramů. Kromě textového výstupu disponuje RAPTOR i metodami pro práci s grafikou. Nástroj disponuje rozsáhlou dokumentací v anglickém jazyce. (WILSON, a další, 2011)

Nestandardní vlastností programu je především způsob, jakým jsou implementovány cykly. Ve většině programovacích jazyků cyklus probíhá, dokud je splněna podmínka opakování. V nástroji RAPTOR se však cykly splněním podmínky ukončují.

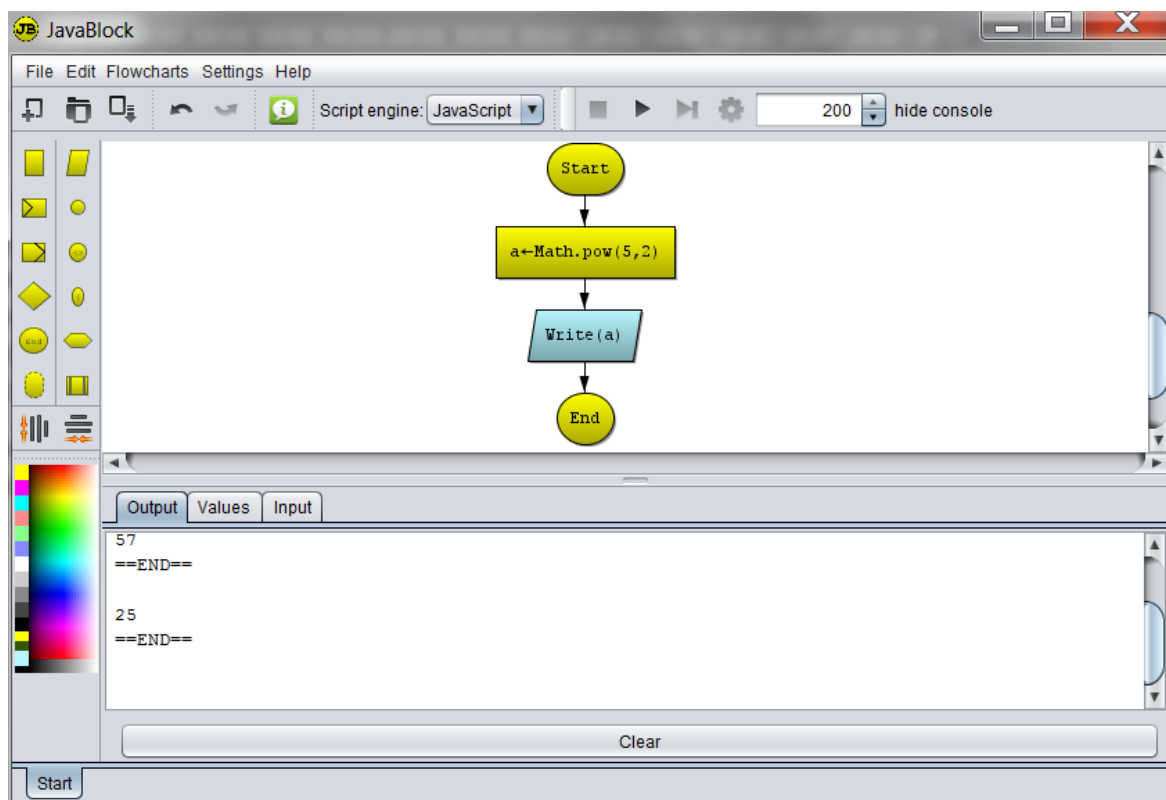


Obrázek 7 – Aplikace RAPTOR

3.2.2 Nástroj JavaBlock

JavaBlock je volně šiřitelný nástroj podporující kromě návrhu a evoluce vývojových diagramů také generování zdrojových kódů v jazyce JavaScript a Python. Program poskytuje možnost rozšíření funkcí prostřednictvím přidavných modulů (pluginů). Nástroj vyžaduje alespoň základní znalosti jazyka JavaScript nebo Python pro tvorbu výrazů a práci s proměnnými.

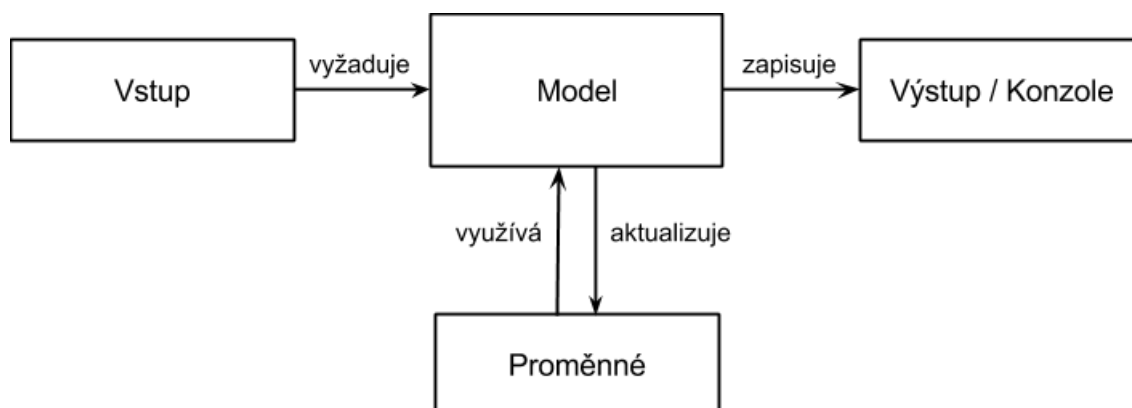
Výhodou nástroje jsou především široké možnosti grafického návrhu vývojového diagramu. Potencionální nevýhodou je absence návodů v anglickém nebo českém jazyce. (KONIECZNY, 2011)



Obrázek 8 – Aplikace JavaBlock

3.3 Model

Model představuje samotný vývojový diagram. Zahrnuje jak strukturu daného vývojového diagramu, tak i definované proměnné a jejich stav. Strukturu vývojového diagramu představují symboly, ze kterých je tvořen, a jejich vzájemné vztahy (relace). Funkcí modelu je také zajistit samotnou evoluci a její vizuální prezentaci. Model také musí disponovat funkcí pro export a import ze souboru.



Obrázek 9 – Schéma modelu v průběhu evoluce

3.4 Proměnné

Proměnné slouží k uchování stavu modelu v průběhu evoluce. Aplikace se s ohledem na své zaměření omezí pouze na číselné proměnné. Všechny proměnné budou v rámci konkrétního modelu globální.

Podporovány budou následující typy proměnných:

- **Číslo:** Obecný typ pro uchování jednoho reálného čísla.
- **Pole:** Jednorozměrné pole reálných čísel.
- **Matice:** Matice reálných čísel typu $m \times n$.

Pole a matice jsou indexovány řadou kladných přirozených čísel počínající hodnotou nula.

3.5 Bloky

Představují základní stavební prvky modelu. Každý blok představuje jeden krok daného algoritmu. Úkolem každého bloku je provedení požadované operace (např. změny hodnoty proměnné nebo vyhodnocení podmínky) a předat řízení bloku následujícímu. Výjimkou je blok pro ukončení algoritmu, který evoluci zastaví. Grafická podoba bloků vychází z normy ČSN ISO 5807 Zpracování informací.

3.5.1 Blok Start

Blok Start určuje bod počátku evoluce. Po spuštění evoluce předává řízení bloku následujícímu. Blok má pouze jeden výstup. V každém modelu musí být právě jeden blok Start.



Obrázek 10 – Blok Start

3.5.2 Blok Konec

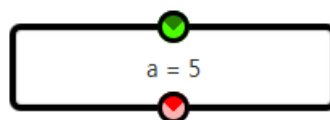
Blok Konec určuje konec evoluce. Blok má pouze jeden vstup. V každém modelu musí být alespoň jeden blok End, na rozdíl od bloku Start jich může být i více.



Obrázek 11 – Blok Konec

3.5.3 Blok Proces

Úkolem tohoto bloku je změnit hodnotu proměnné na základě matematického výrazu. Poté blok předává řízení bloku následujícímu. Blok má jeden vstup a jeden výstup.

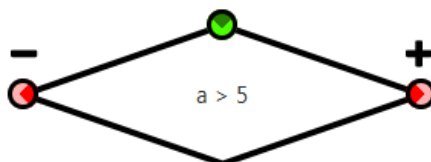


Obrázek 12 – Blok Proces

Na obrázku (Obrázek 12) je příklad bloku Proces, který do proměnné „a“ uloží hodnotu 5.

3.5.4 Blok Rozhodování

Úkolem tohoto bloku je předat řízení jednomu z následujících bloků na základě vyhodnocení binární⁵ podmínky. Blok má jeden vstup a dva výstupy. Výstup označený symbolem plus je použit, je-li podmínka splněna. V opačném případě je použit výstup označen symbolem mínus.



Obrázek 13 – Blok Rozhodování

Následující tabulka (Tabulka 3) obsahuje seznam porovnávacích operátorů podporovaných blokem Rozhodování a podmínkou v bloku Cyklus.

Tabulka 3 – Přehled podporovaných porovnávacích operátorů

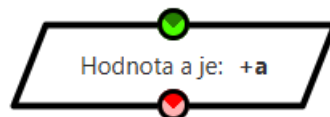
operátor	popis funkce
$a == b$	Podmínka je splněna, je-li hodnota „a“ rovna hodnotě „b“.
$a != b$	Podmínka je splněna, není-li hodnota „a“ rovna hodnotě „b“.
$a \leq b$	Podmínka je splněna, je-li hodnota „a“ menší nebo rovna hodnotě „b“
$a \geq b$	Podmínka je splněna, je-li hodnota „a“ větší nebo rovna hodnotě „b“
$a > b$	Podmínka je splněna, je-li hodnota „a“ větší než hodnota „b“
$a < b$	Podmínka je splněna, je-li hodnota „a“ menší než hodnota „b“

a a b může být reálné číslo, proměnná nebo výraz (viz. podkapitola 3.7 Výrazy).

3.5.5 Blok Výstup

Blok Výstup slouží pro výpis textového řetězce na obrazovku. Dále umožňuje výpis obsahu zadané proměnné. Po provedení výpisu blok předá řízení bloku následujícímu. Blok má jeden vstup a jeden výstup.

⁵ Výsledkem binární podmínky je hodnota pravda (angl. true) nebo nepravda (angl. false)

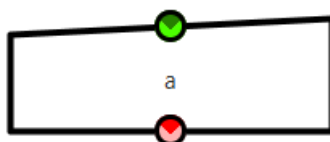


Obrázek 14 – Blok Výstup

Na obrázku (Obrázek 14) je příklad bloku Výstup, který vypíše textový řetězec „Hodnota a je“ a k němu připojí aktuální hodnotu proměnné „a“.

3.5.6 Blok Vstup

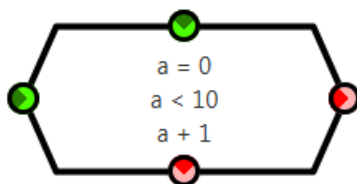
Blok Vstup realizuje interaktivní vstup uživatele v průběhu evoluce. Před předáním řízení bloku následujícímu vyzve uživatele k zadání hodnoty nastavené proměnné. Blok má jeden vstup a jeden výstup.



Obrázek 15 – Blok Vstup

3.5.7 Blok Cyklus

Blok Cyklus realizuje cyklus s podmínkou na začátku. Blok je v podstatě kombinací podmínky a přiřazení hodnoty. Blok má dva vstupy a dva výstupy. Vrchní vstup slouží k inicializaci řídicí proměnné na základě nastaveného matematického výrazu. Postranní vstup inkrementuje řídicí proměnnou o nastavenou hodnotu. V případě, že je podmínka splněna, je zvolen spodní výstup a cyklus pokračuje. V opačném případě je použit postranní výstup a cyklus je ukončen.

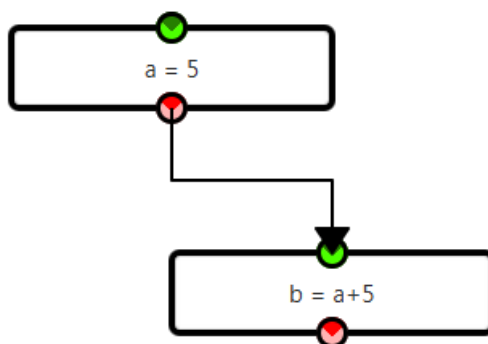


Obrázek 16 – Blok Cyklus

Na obrázku (Obrázek 16) je příklad blok Cyklus, jehož řídicí proměnná je proměnná „a“ inicializovaná na hodnotu 0. Cyklus pokračuje, dokud je splněna podmínka, že a je menší než 10. V každé iteraci je k hodnotě proměnné „a“ přičtena hodnota 1.

3.6 Relace

Relace představují spojnice vývojového diagramu. Realizují vztah předchůdce – následník mezi dvěma bloky daného modelu. Z každého výstupu musí existovat právě jedna relace. Tím se zajistí determinovanost daného algoritmu. Počet relací vstupujících do bloku není omezen. Dále nejsou přípustné relace vstupující do stejného bloku, ze kterého vystupují.



Obrázek 17 – Zobrazení relace mezi dvěma bloky

3.7 Výrazy

Některé bloky umožňují zadání matematických výrazů. Tyto výrazy se vyhodnocují v průběhu evoluce a to vždy před tím, než je předáno řízení následujícímu bloku. Výsledkem výrazu je vždy jedno reálné číslo. Výraz je vždy vyhodnocen před provedením akce daného bloku (např. v bloku Rozhodování jsou nejprve vyhodnoceny výrazy a jejich výsledná hodnota se poté porovná pomocí logických operátorů).

Tabulka 4 – Přehled podporovaných aritmetických operátorů

+	součet
-	rozdíl
*	součin
/	podíl
%	modulo (zbytek po celočíselném dělení)
^	umocnění

Při vyhodnocování výrazů je dodrženo pořadí základních aritmetických operací. Toto chování lze ovlivnit použitím kulatých závorek. Výraz uzavřený v závorkách má vždy nejvyšší prioritu.

Př.

- $2 + 5 * 3 = 17$
- $(2 + 5) * 3 = 21$

Ve výrazech lze využívat i definované proměnné zapsáním jejich názvu. V případě proměnných typu pole a matice uvádíme index do hranatých závorek za název proměnné.

Tabulka 5 – Příklady práce s proměnnými

výraz	popis
$a + 1$	Výsledkem výrazu je hodnota proměnné „a“ navýšená o hodnotu 1.
$p[1]$	Výsledkem výrazu je hodnota proměnné „p“ typu pole na indexu 1.
$p[a - 1]$	Výsledkem výrazu je hodnota proměnné „p“ typu pole na indexu s hodnotou proměnné „a“ sniženou o hodnotu 1.
$m[a][0]$	Výsledkem výrazu je hodnota proměnné „m“ typu matice v řádku s hodnotou proměnné „a“ a sloupci s hodnotou 0.

Ve výrazech lze využít i několik jednoparametrických funkcí. Názvy funkcí se zapisují velkými písmeny. Za názvem funkce obvykle následují kulaté závorky obsahující výraz, na jehož výsledek bude daná funkce použita. Výsledkem funkce je vždy reálné číslo. Následující tabulka (Tabulka 6) obsahuje seznam podporovaných funkcí.

Tabulka 6 – Přehled podporovaných funkcí

syntaxe	popis funkce
ABS(výraz)	Výsledkem funkce je absolutní hodnota výrazu v závorkách.
ROUND(výraz)	Zaokrouhlí výraz v závorkách na nejbližší celé číslo.
FLOOR(výraz)	Zaokrouhlí výraz v závorkách dolů na celé číslo.
CEIL(výraz)	Zaokrouhlí výraz v závorkách nahoru na celé číslo.
SIN(výraz)	Výsledkem je sinus daného výrazu. (výraz je v radiánech)
COS(výraz)	Výsledkem je kosinus daného výrazu. (výraz je v radiánech)
TAN(výraz)	Výsledkem je tangens daného výrazu. (výraz je v radiánech)
ASIN(výraz)	Výsledkem je arkus sinus daného výrazu v radiánech.
ACOS(výraz)	Výsledkem je arkus kosinus daného výrazu v radiánech.
ATAN(výraz)	Výsledkem je arkus tangens daného výrazu v radiánech.
SIZE(pole)	Výsledkem je počet prvků dané proměnné typu pole. Do parametru v závorkách se uvádí pouze název proměnné.
ROWS(matice)	Výsledkem je počet řádků dané proměnné typu matice. Do parametru v závorkách se uvádí pouze název proměnné.
COLS(matice)	Výsledkem je počet sloupců dané proměnné typu matice. Do parametru v závorkách se uvádí pouze název proměnné.
RAND	Výsledkem je náhodné reálné číslo mezi hodnotami 0 a 1. Tato funkce nemá žádný parametr.

Z hlediska priority operací se vždy nejprve vyhodnocuje výraz v parametru dané funkce. Poté se na výslednou hodnotu výrazu aplikuje daná funkce. Vyhodnocování dále pokračuje podle pořadí základních aritmetických operací, dokud není výraz nahrazen jedním reálným číslem.

Kromě funkcí lze využít i několik konstant. Názvy konstant se zapisují velkými písmeny bez parametrů.

Podporované konstanty:

- PI – Ludolfovo číslo (π) s přesností na 15 desetinných míst.
- E – Eulerovo číslo s přesností na 15 desetinných míst.

4 Implementace

Následující kapitola se věnuje popisu implementace a struktury vyvíjené aplikace. V úvodu bude představena zvolená platforma, nad kterou je aplikace implementována. Dále zde budou rozvedeny jednotlivé moduly a způsob jejich spolupráce.

4.1 Platforma

Aplikace je postavena na platformě Java SE⁶ ve verzi 8. K vývoji grafického uživatelského rozhraní je využit Framework JavaFX, který je součástí platformy Java SE. Aplikace je vytvořena pomocí open source vývojového prostředí NetBeans IDE.

4.1.1 Java

Java je jedním z nejrozšířenějších programovacích jazyků na světě. Jeho hlavní předností je snadná přenositelnost na různé systémy disponující platformou Java bez nutnosti program kompilovat zvlášť pro každý takovýto systém.

Jazyk Java vznikl v roce 1995 pod společností Sun Microsystems. Syntaxe jazyka vycházela zejména z jazyků C a C++, které byly v této době nejrozšířenější. Hlavním příslibem bylo heslo „Write Once, Run Anywhere“ (napiš jednou, spust' kdekoliv) symbolizující univerzálnost jazyka pro všechny podporované platformy. V roce 2009 byla společnost Sun Microsystems koupena společností Oracle Corporation která s vývojem Javy pokračuje. Současná hlavní verze platformy je verze 8 vydaná 18. Března 2014.

Java na rozdíl od svých předchůdců používá automatickou správu paměti. Programátor tedy rozhoduje pouze o tom, kdy bude daný objekt vytvořen, a musí zajistit, že na tento objekt bude neustále odkazovat alespoň jedna reference. Objekt, na který neodkazuje žádná reference, je považován za nedostupný a je pomocí tzv. garbage collectoru z paměti odstraněn. V jazyce Java tedy neexistuje destruktorka. Destruktorka je programová konstrukce podobná konstruktoru sloužící k uvolnění paměti při odstraňování objektů, jakou můžeme najít například v jazyce C++. Tento přístup výrazně snižuje možnost výskytu chyb způsobujících únik paměti.

Implementace platformy Java je v současnosti rozdělena do dvou distribucí. První distribucí je Java Runtime Environment (JRE), která obsahuje část platformy Java SE potřebnou pro běh programů napsaných v jazyce Java (tj. JVM). Tato distribuce je určena především pro koncové uživatele. Druhou distribucí je Java Development Kit (JDK) určená pro softwarové vývojáře, která navíc obsahuje vývojové nástroje jako je Java kompilátor, Javadoc a debugger.

Při kompilaci programu v jazyce Java se místo strojového kódu pro konkrétní systém vytváří pouze tzv. mezikód (bajtkód). Tento formát je nezávislý na architektuře počítače nebo zařízení. Program pak lze spustit na libovolném zařízení, které má k dispozici tzv. virtuální stroj Javy (JVM – Java Virtual Machine). JVM byl později doplněn technologií

⁶ SE = standardní edice (angl. Standard Edition)

JIT (Just In Time compilation), která mezikód za běhu programu dynamicky kompilují do strojového kódu pro daný systém, a přidává řadu dalších optimalizací. To umožňuje programům napsaným v jazyce Java se výkonem vyrovnat programům kompilovaným přímo do strojového kódu.

Výsledkem kompilace programu v jazyce Java je soubor s příponou „jar“. Tento soubor je archiv obsahující zkompilevané třídy a jejich metadata společně s dalšími zdroji, které tyto třídy využívají (např. obrázky, zvuky, styly atd.). Archiv může být spustitelnou aplikací, nebo ho lze využít jako softwarovou knihovnu pro jiný program. Pro spuštění souboru s příponou „jar“ musí být na klientovi nainstalována platforma JRE ve verzi stejné nebo vyšší, než je verze JDK kterou byl daný archiv zkompileván. Platforma JRE je zpětně kompatibilní.

4.1.2 JavaFX

JavaFX je softwarová platforma postavená na bázi platformy Java sloužící k tvorbě aplikací s grafickým uživatelským rozhraním. Platforma JavaFX vznikla za účelem nahradit zastaralý Swing jako standardní knihovnu pro tvorbu GUI aplikací. Platforma byla navržena s důrazem na oddělení aplikační logiky od uživatelského rozhraní. Tím se zajistí vyšší přehlednost a flexibilita zdrojového kódu.

Ke specifikaci vzhledu jednotlivých elementů využívá platforma JavaFX standard CSS ve verzi 2.1 s rozšířeními pro podporu specifických vlastností platformy JavaFX. Pro odlišení vlastností určených pro elementy JavaFX se v CSS používá přípona „-fx-“.

Za účelem návrhu GUI byl vytvořen značkovací jazyk FXML na bázi jazyka XML. FXML poskytuje praktickou alternativu k tvorbě GUI přímo v kódu aplikační logiky. Výhodou FXML je zejména jeho hierarchické uspořádání, které více odpovídá struktuře objektů reprezentujících jednotlivé grafické prvky uživatelského rozhraní. (BROWN, 2011)

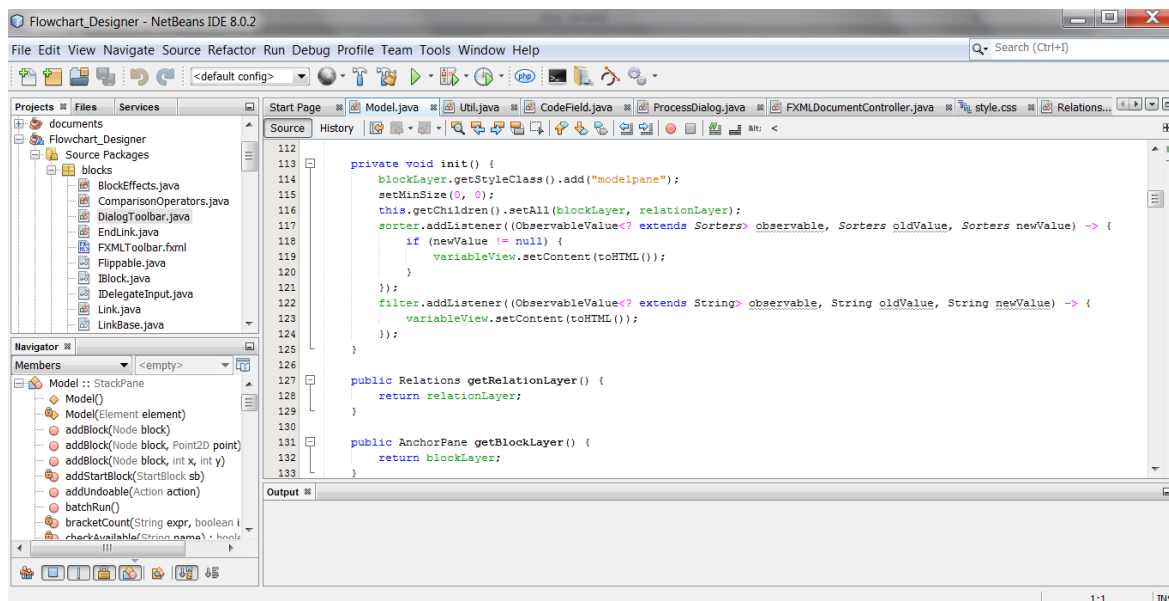
Od verze Java SE 7u6 je knihovna JavaFX součástí platformy Java SE.

4.1.3 NetBeans IDE

NetBeans IDE je svobodné, zdarma distribuované integrované vývojové prostředí, které v současné době vlastní a sponzoruje firma Oracle Corporation. Prostedí NetBeans je vytvořeno v jazyce Java. Díky své modulární softwarové architektuře umožňuje prostředí NetBeans programování v celé řadě programovacích jazyků (patří mezi ně např. Java, C++, C#, PHP, ...) a rozšíření svých funkcí prostřednictvím tzv. pluginů.

Alternativy:

- Eclipse – Open source prostředí, původně vyvíjené spol. IBM.
- IntelliJ IDEA – Komerční prostředí od společnosti JetBrains.
- JDeveloper – Volně šiřitelné prostředí vyvíjené spol. Oracle Corporation.



Obrázek 18 – Prostředí NetBeans IDE ve verzi 8.0.2

Prostředí NetBeans IDE je dostupné pod svobodnou licencí GNU GPL v2, nebo pod licencí CDDL (Common Development and Distribution License).

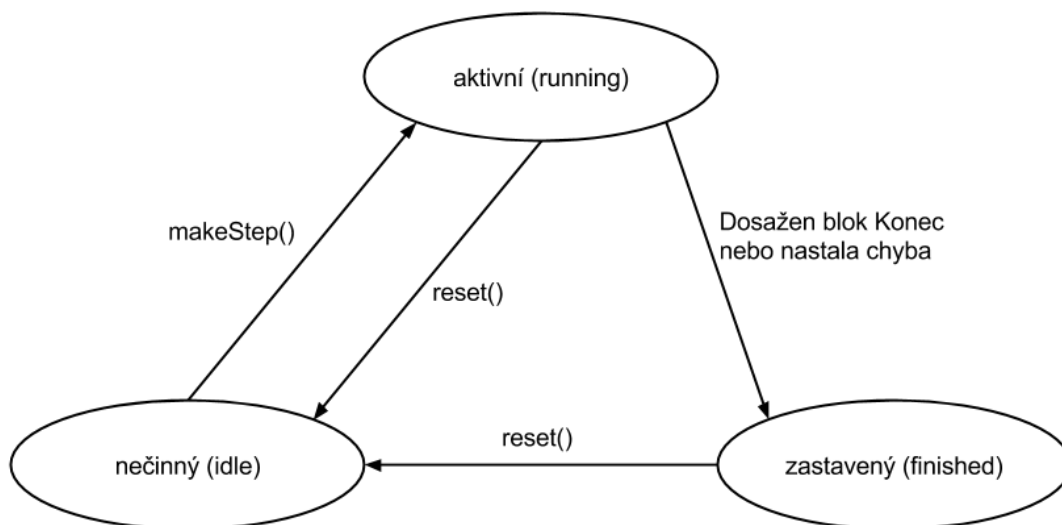
4.2 Jádru aplikace

Jádrem aplikace je třída *Model*, která se nachází v balíčku *core*. Instance třídy *Model* představují funkční vývojový diagram společně s jeho proměnnými. Třída *Model* zajišťuje posun evoluce daného vývojového diagramu.

Model obsahuje seznam všech bloků tvořících vývojový diagram a především reference na počáteční blok Start a blok aktuální v průběhu evoluce. Model si také uchovává historii změn vývojového diagramu nebo proměnných a umožňuje tyto změny zvrátit.

Model se v okamžiku může nacházet v jednom z 3 stavů.

- **nečinný (idle)** – Aktuálně neprobíhá evoluce a lze přidávat, odebírat a měnit bloky, nebo editovat deklarované proměnné. S tohoto stavu lze spustit evoluci.
- **aktivní (running)** – Aktuálně probíhá evoluce a nelze provádět jakékoli změny modelu nebo proměnných.
- **zastavený (finished)** – Evoluce skončila dosažením bloku Konec nebo chybou za běhu. Před prováděním jakýchkoliv změn v modelu nebo proměnných je nutno provést reset.

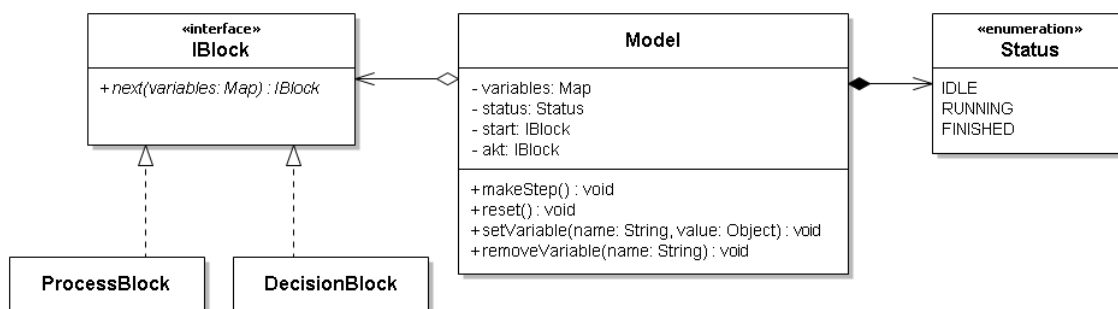


Obrázek 19 – Změny stavů modelu

4.2.1 Průběh evoluce

Evoluci modelu spustíme pomocí metody *makeStep* v třídě *Model*. Pokud se model nacházel ve stavu *nečinný*, je přepnut do stavu *aktivní* a jako aktuální blok je nastaven blok Start daného modelu. Použitím stejné metody následně posouváme aktuální blok na blok následující, který získáme použitím metody *next* aktuálního bloku. Tímto způsobem bude evoluce pokračovat, dokud není dosažen blok Konec, nebo při provádění metody *next* nenastane chyba.

Aktuální blok je v průběhu evoluce označen zeleně. V případě, že při provádění metody *next* nastala chyba, je daný blok označen červeně.



Obrázek 20 – UML diagram: Jádru aplikace

Aplikace nedokáže zajistit, že jakýkoliv modelovaný algoritmus bude konečný. Evoluci lze tedy kdykoliv přerušit použitím metody *reset*. Metoda *reset* přepne model do stavu *nečinný* a vrátí hodnoty proměnných do počátečního stavu.

4.2.2 Implementace proměnných

Veškeré proměnné daného modelu jsou uloženy v asociativním poli s názvem *variables* typu *HashMap*, které se nachází ve třídě *Model*. Asociativní pole je typ pole, jehož prvky nejsou identifikovány pomocí číselných indexů, ale pomocí klíčů. V jazyce Java může být tímto klíčem jakýkoli objekt. V případě pole *variables* je tímto řetězec typu *String*, který reprezentuje název dané proměnné. Všechny proměnné musí být před použitím v modelu definovány.

Proměnné se nastavují pomocí metody *setVariable* ve třídě *Model*. Prvním parametrem této metody je jméno proměnné, která se má přidat nebo modifikovat. Druhým parametrem je pak hodnota dané proměnné.

Jméno proměnné musí začínat malým nebo velkým písmenem a dále může pokračovat písmeny, číslicemi nebo znakem "_" (podtržítko). Jméno proměnné se nesmí shodovat s názvy funkcí nebo konstant (viz. podkapitola 3.7) nebo s klíčovými slovy jazyka Javascript.

Hodnotou proměnné pak může být typ *double*, reprezentující jedno reálné číslo nebo jednorozměrné pole typu *double*, reprezentující pole reálných čísel, příp. dvojrozměrné pole typu *double* reprezentující matici reálných čísel.

Příklad: Vytvoření proměnné „a“ typu reálné číslo s hodnotou 15,5.

```
Model model = new Model();  
model.setVariable("a", 15.5d);
```

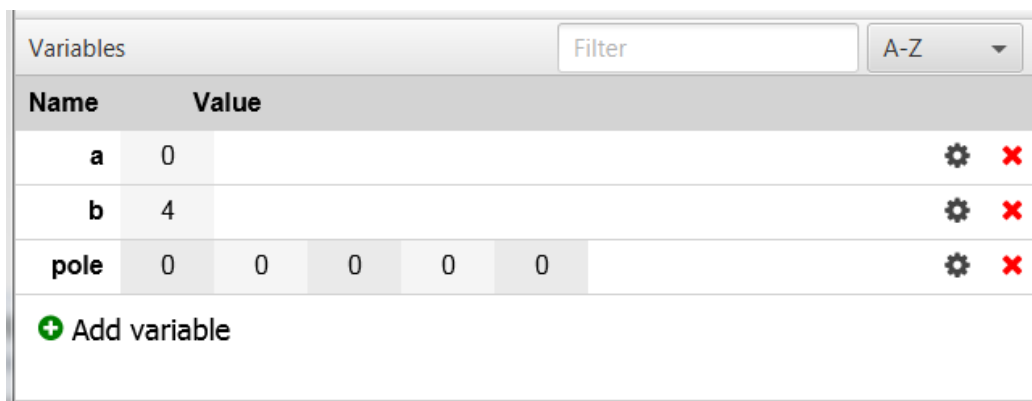
Odstranění proměnné lze provést obdobným způsobem pomocí metody *removeVariable* ve třídě *Model*. Metoda *removeVariable* přijímá pouze jediný parametr a tím je název nežádoucí proměnné.

Příklad: Odstranění vytvořené proměnné s názvem „a“.

```
model.removeVariable("a");
```

Veškeré změny v poli proměnných v době, kdy se model nachází ve stavu *nečinný*, jsou evidovány a lze je podle potřeby vrátit zpět. To platí i pro přidávání a odebírání proměnných.

Aktuální stav proměnných lze sledovat pomocí tabulky v levém dolním rohu uživatelského rozhraní. Proměnné lze seřadit podle názvu nebo typu, nebo filtrovat zadáním části názvu. Za zobrazení tabulky zodpovídá třída *VariableView* z balíčku *variableview*. Třída *Model* tuto tabulku aktualizuje při jakýchkoli změnách pole proměnných a při každém kroku během evoluce.



The screenshot shows a window titled 'Variables'. It has a 'Filter' input field and a dropdown menu set to 'A-Z'. Below is a table with columns 'Name' and 'Value'. The table contains three rows: 'a' with value '0', 'b' with value '4', and 'pole' with five '0's. Each row has a gear icon and a red 'X' icon to its right. At the bottom, there is a button with a green plus icon and the text 'Add variable'.

Name	Value		
a	0	⚙️	❌
b	4	⚙️	❌
pole	0 0 0 0 0	⚙️	❌

+ Add variable

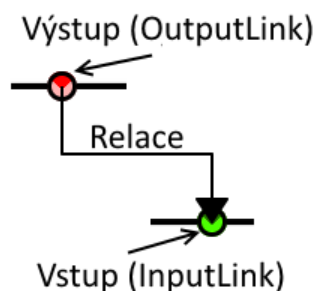
Obrázek 21 – Vizualizace stavu proměnných

V průběhu evoluce je výchozí stav proměnných uložen v asociativním poli *initVar*. Při použití metody *reset* je tak obnoven výchozí stav všech proměnných tak, jak byly definovány uživatelem.

4.2.3 Implementace relací

Každá relace je tvořena třemi objekty. Prvním je instance třídy *OutputLink*, reprezentující výstup z předchozího bloku. Druhým je instance třídy *InputLink*, reprezentující vstup do bloku následujícího. Posledním objektem je instance třídy *Relation*, která zajišťuje vizuální reprezentaci relace a předává řízení následujícímu bloku v průběhu evoluce.

Každý model obsahuje instanci třídy *Relations*, která uchovává seznam všech relací a zajišťuje jejich vytváření. Relace se vytváří pomocí dvou metod ve třídě *Relations*. Metoda *initRelation* přijímá jako parametr objekt typu *OutputLink* a určuje zdroj vytvářené relace. Metoda *finishRelation* přijímá jako parametr objekt typu *InputLink*, reprezentující cíl dané relace, a relaci následně vytvoří. Důvodem tohoto přístupu je způsob, jakým se relace vytváří pomocí uživatelského rozhraní. Kliknutím na výstup bloku, který má být zdrojem budoucí relace, se zavolá metoda *initRelation* a předá se jí daný výstup. Následným kliknutím na vstup cílového bloku se zavolá metoda *finishRelation* s daným vstupem jako parametr, která relaci vytvoří.



Obrázek 22 – Vizuální reprezentace relace

Pro každý výstup může existovat nejvýše jedna relace. V případě vytvoření relace z výstupu, ze kterého již relace existuje, je předchozí relace odstraněna. Počet relací do jednoho vstupu není omezen. Pro spuštění evoluce musí mít každý výstup definovanou relaci. Tímto je zajištěna determinovanost modelovaného algoritmu.

4.3 Import/Export modelu

Pro trvalé uchování modelu i po ukončení aplikace je implementována možnost serializovat model do souboru a následně jej ze souboru opět načíst. Výsledkem serializace je soubor s příponou „fmd“ ve formátu XML.

4.3.1 XML

XML (Extensible Markup Language) je flexibilní značkovací jazyk který byl vyvinut a standardizován konsorciem W3C⁷. Jazyk XML je určen především pro výměnu dat mezi aplikacemi a publikování dokumentů. Data ve formátu XML jsou uložena v čitelné textové podobě a organizována do hierarchické struktury.

Základem struktury XML jsou tzv. elementy. Každý element je definován pomocí počáteční a koncové značky. Počáteční značka je tvořena názvem elementu uzavřeným ve špičatých závorkách. Název elementu nesmí obsahovat mezery.

Příklad počáteční značky:

```
<Element>
```

Koncová značka vypadá stejně, jen je před názvem elementu, který ukončuje navíc znak lomítka (/).

Příklad koncové značky:

```
</Element>
```

Cokoli se nachází mezi těmito značkami, je považováno za potomka daného elementu. Potomkem můžou být textová data nebo jiný element. Každý element XML dokumentu musí mít jak počáteční tak koncovou značku. Výjimkou jsou prázdné elementy, které lze zapsat zkráceně přidáním znaku lomítka (/) do počáteční značky před ukončovací závorku.

⁷ W3C (World Wide Web Consortium) – Mezinárodní konsorcium pro webové standardy.

Příklad prázdného elementu:

```
<PrázdnýElement />
```

Elementy se stejným názvem a předkem se mohou opakovat. Výjimkou je tzv. kořenový element, tedy element, který nemá žádného předka. Kořenový element je přímým nebo nepřímým předkem všech elementů v daném dokumentu.

Každý element může mít definované atributy. Atributy se zapisují do počáteční značky elementu. Syntaxe zápisu atributů je název atributu, následovaný znakem rovno (=), následovaným hodnotou atributu v uvozovkách. Jednotlivé atributy s mezi sebou oddělují mezerami. Pro názvy atributů platí stejná pravidla jako pro názvy elementů.

Příklad prázdného elementu s atributy:

```
<Element atribut1="0.5" atribut2="textová hodnota" />
```

Na začátku XML dokumentu se nachází speciální element nazývaný XML deklarace. Ten zpravidla obsahuje atribut *version*, definující verzi standardu XML, a atribut *encoding* udávající použité kódování.

Syntaxe XML deklarace:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Příklad validního XML dokumentu:

```
<?xml version="1.0" encoding="UTF-8" ?>
<KořenovýElement>
  <Element atribut="prázdný element" />
  <Element>
    <Potomek atribut="hodnota" />
  </Element>
  <Element atribut="40" />
</KořenovýElement>
```

Jazyk XML rozlišuje velká a malá písmena.

4.3.2 Postup exportu

Export je realizován pomocí knihovny DOM. Knihovna DOM vyvádí stromovou objektovou reprezentaci dokumentu XML v paměti. Tato struktura se následně pomocí metody *transform* z třídy *Transformer* zapíše do souboru XML.

Strukturu vytváří třída *Model* pomocí metody *toXml*. Kořenovým elementem je element *Model* s atributy určujícími šířku a výšku pracovní plochy. Potomkem elementu *Model* je element *Variables*, obsahující elementy *variable* pro jednotlivé proměnné, a element *Blocks* obsahující elementy pro specifické bloky modelu. Každý blok implementuje metodu *serializeXML*, která je zodpovědná za vytvoření struktury pro daný blok.

Import probíhá obdobným způsobem, jen je v tomto případě použita statická metoda *createFromXML* ve třídě *Model*, která na základě dodaného souboru vytvoří nový model.

4.4 Implementace bloků

Každý blok implementuje rozhraní *IBlock* v balíčku *blocks*. Rozhraní *IBlock* definuje především metodu *next*, která slouží pro provedení akce daného bloku a předání řízení bloku následujícímu. Dále rozhraní definuje metodu *serializeXML* sloužící pro serializaci implementujícího bloku a metody nezbytné pro vizualizaci evoluce a ověření validity modelu.

Implementace jednotlivých bloků jsou potom organizovány do balíčků s názvem daného bloku. Např. implementace bloku *Proces* je v balíčku *blocks.process*. Tyto balíčky zpravidla obsahují následující položky.

- FXML.fxml – Definuje vzhled bloku.
- [název bloku]Block.java – Definuje chování bloku. Implementuje rozhraní *IBlock*.
- FXMLDialog.fxml – Definuje vzhled dialogu pro editaci bloku.
- [název bloku]Dialog.java – Definuje chování dialogu.

Součástí bloku jsou také jeho vstupy a výstupy. Při zavolání metody *next* tak blok na základě své funkce ve vývojovém diagramu zvolí jeden z výstupů (instance třídy *OutputLink*). Zavoláním metody *getNextBlock* zvoleného výstupu získá referenci na následující blok, kterou následně předá modelu. Tento postup se v průběhu evoluce neustále opakuje.

Každý blok kromě bloku *Start* zahrnuje kontextové menu sloužící k odstranění, duplikování, příp. k editaci daného bloku. Veškeré provedené změny jsou logovány a lze je vrátit zpět.

4.5 Implementace výrazů

Aritmetické výrazy jsou vyhodnocovány pomocí skriptovacího enginu Nashorn, který je součástí platformy Java SE 8. Výrazy jsou implementovány v jazyce JavaScript. K překladu zjednodušených výrazů do kódu jazyka JavaScript slouží třída *Translator* v balíčku *core*. Ta zajišťuje především překlad podporovaných funkcí ve výrazech (Tabulka 6) na funkce jazyka JavaScript prostřednictvím výčtu *Translations*. Dále také nahrazuje symbol $^$ funkcí *Math.pow(x, n)*, která provádí mocninu x^n .

Dodatečné funkce lze definovat v souboru *jsFunctions.js* v balíčku *core*. Tyto funkce lze poté využít při vyhodnocování výrazů. Funkce se definují v jazyce JavaScript. Pro zpřístupnění funkce uživateli v editoru výrazů je nutné vytvořit položku ve výčtu *Translations* definující syntaxi funkce a její překlad.

Kontrola syntaxe výrazů je řešena metodou *checkExpression* ve třídě *Model*. Kontrola probíhá na základě aktuálně existujících proměnných a jejich typů. Parametrem metody je kontrolovaný výraz. Kontrola se provádí při editaci bloku, nebo při změně proměnných modelu. V případě nedostatku je vystavena výjimka *InvalidExpressionException*.

4.5.1 Postup vyhodnocení výrazu

Vyhodnocování výrazů probíhá v metodě *next* pro daný blok. Následující ukázka je pro implementaci bloku *Proces*.

1. Získání instance skriptovacího engine pomocí statické metody v třídě *Util* z balíčku *core*.

```
ScriptEngine engine = Util.createExpressionEngine();
```

2. Nastavení proměnných engine na základě parametru *variables*.

```
for (Map.Entry<String, Object> entry : variables.entrySet()) {  
    engine.put(entry.getKey(), entry.getValue());  
}
```

3. Přeložení a vyhodnocení výrazu.

```
String parsedExpression = Translator.parse(expression.toString());  
engine.eval(parsedExpression);
```

4. Uložení proměnných zpět do asociativního pole *variables*.

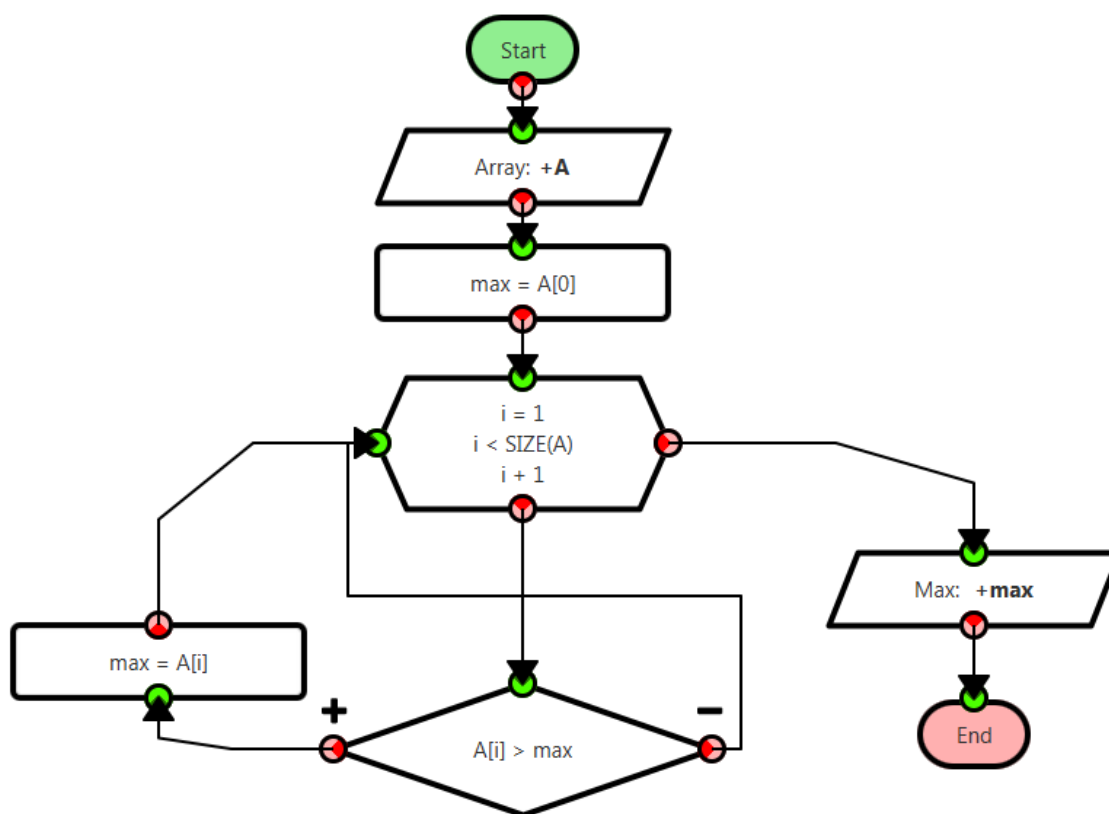
```
for (Map.Entry<String, Object> entry : variables.entrySet()) {  
    setVariable(entry, engine.get(entry.getKey()));  
}
```

5 Testování na vybraných algoritmech

Následující kapitola je věnována testování aplikace. K tomuto účelu bylo zvoleno několik algoritmů z oblasti algoritmizace a programování. Každý algoritmus je navržen a testován přímo v aplikaci pro sadu přípustných vstupních dat. Modely následujících algoritmů budou uloženy v importovatelných souborech na přiloženém CD.

5.1 Algoritmus Max

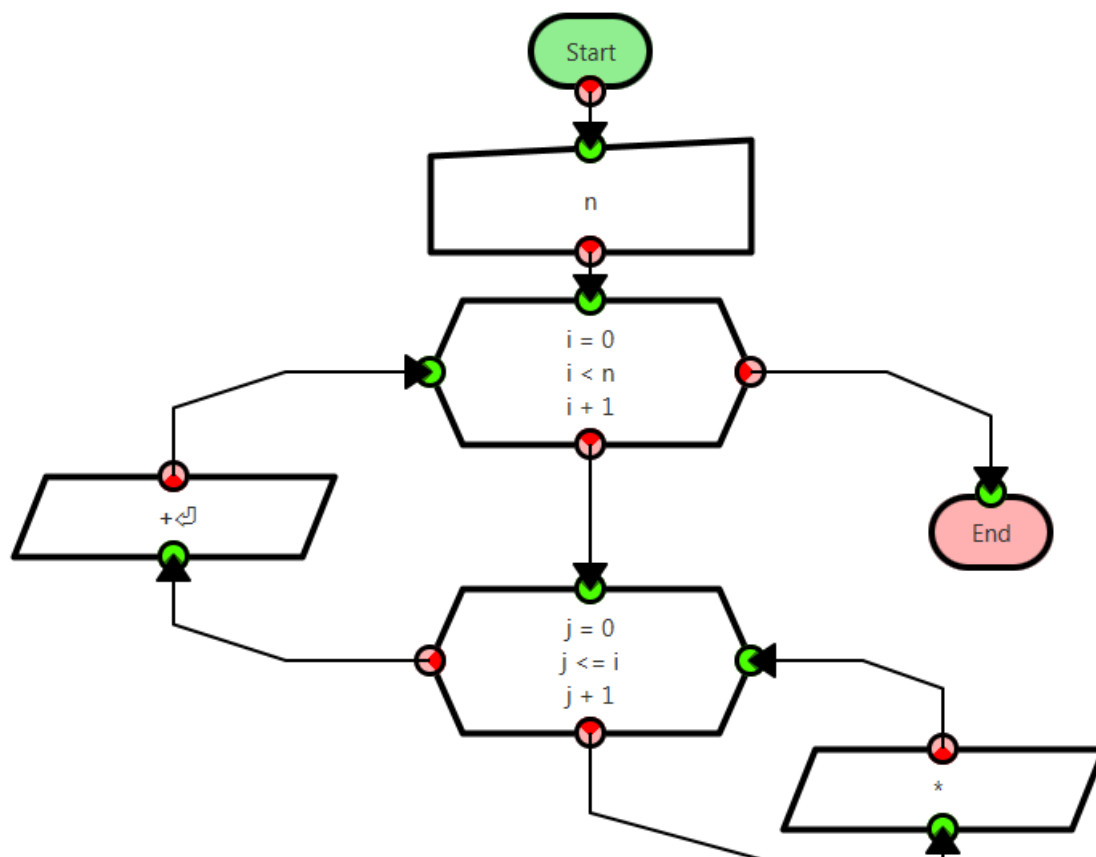
Jednoduchý algoritmus pro nalezení maximální hodnoty v poli. Proměnná A je typu pole s libovolným počtem hodnot. Algoritmus nejprve vypíše obsah pole A . Proměnná max je inicializována na první hodnotu v poli. Následným cyklem se prochází všechny prvky pole, a pokud $A[i] > max$, je hodnota proměnné max nahrazena hodnotou $A[i]$. Po skončení cyklu je hodnota max vypsána na obrazovku.



Obrázek 23 – Algoritmus Max

5.2 Algoritmus Schody

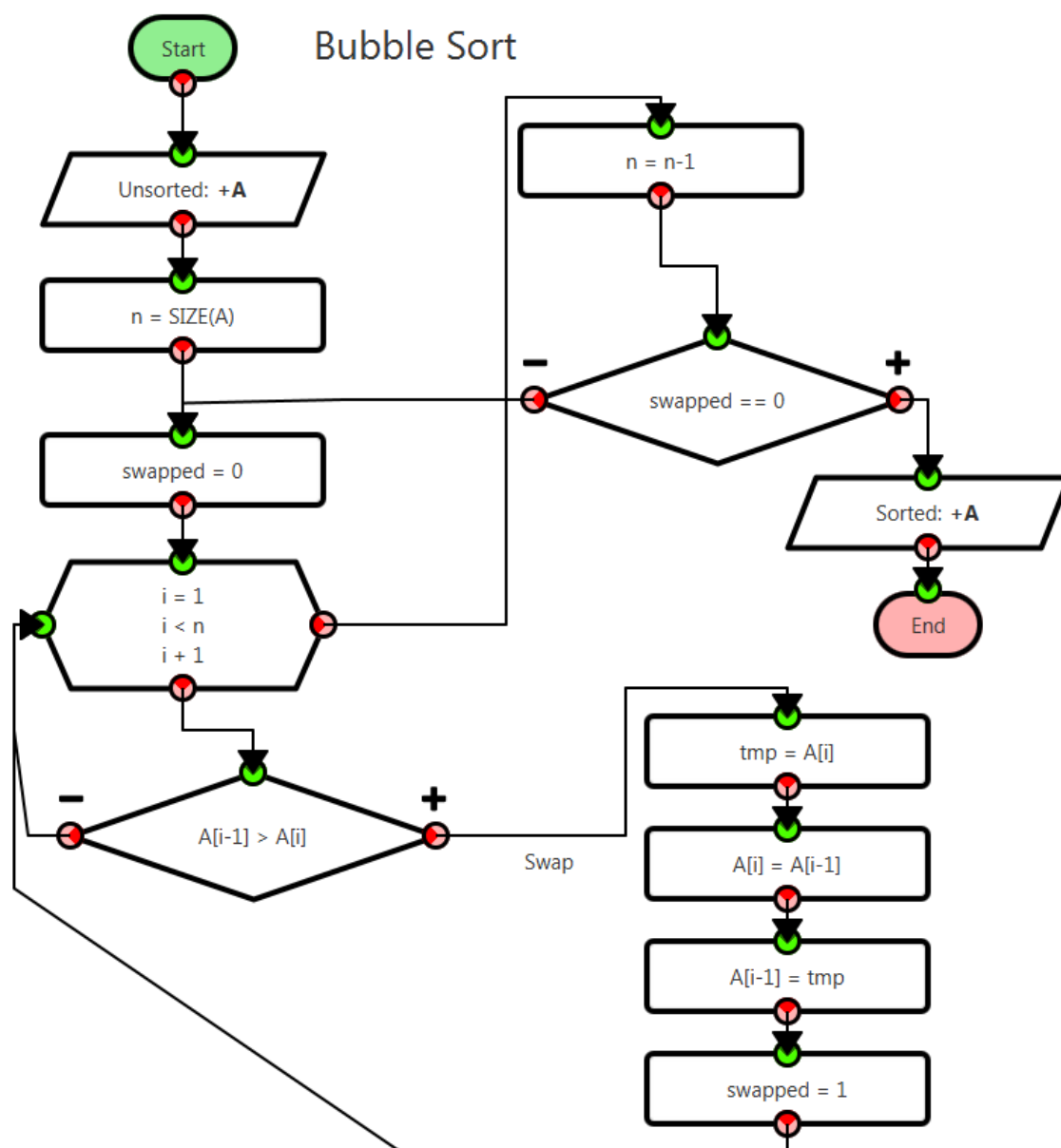
Algoritmus *Schody* demonstruje použití vnořeného cyklu. Algoritmus vypíše na výstup zadaný počet řádků s tím, že na každém následujícím řádku bude o 1 symbol více, než na řádku předcházejícím. Na počátku je uživatel vyzván k zadání počtu řádků, který se uloží do proměnné n . Následuje inicializace vnějšího cyklu s řídicí proměnnou i , který je proveden n -krát. Vnitřní cyklus s řídicí proměnnou j je vždy proveden $i + 1$ krát a při každé iteraci vypíše znak "*". Při ukončení vnitřního cyklu je provedeno odřádkování.



Obrázek 24 – Algoritmus Schody

5.3 Algoritmus Bubble Sort

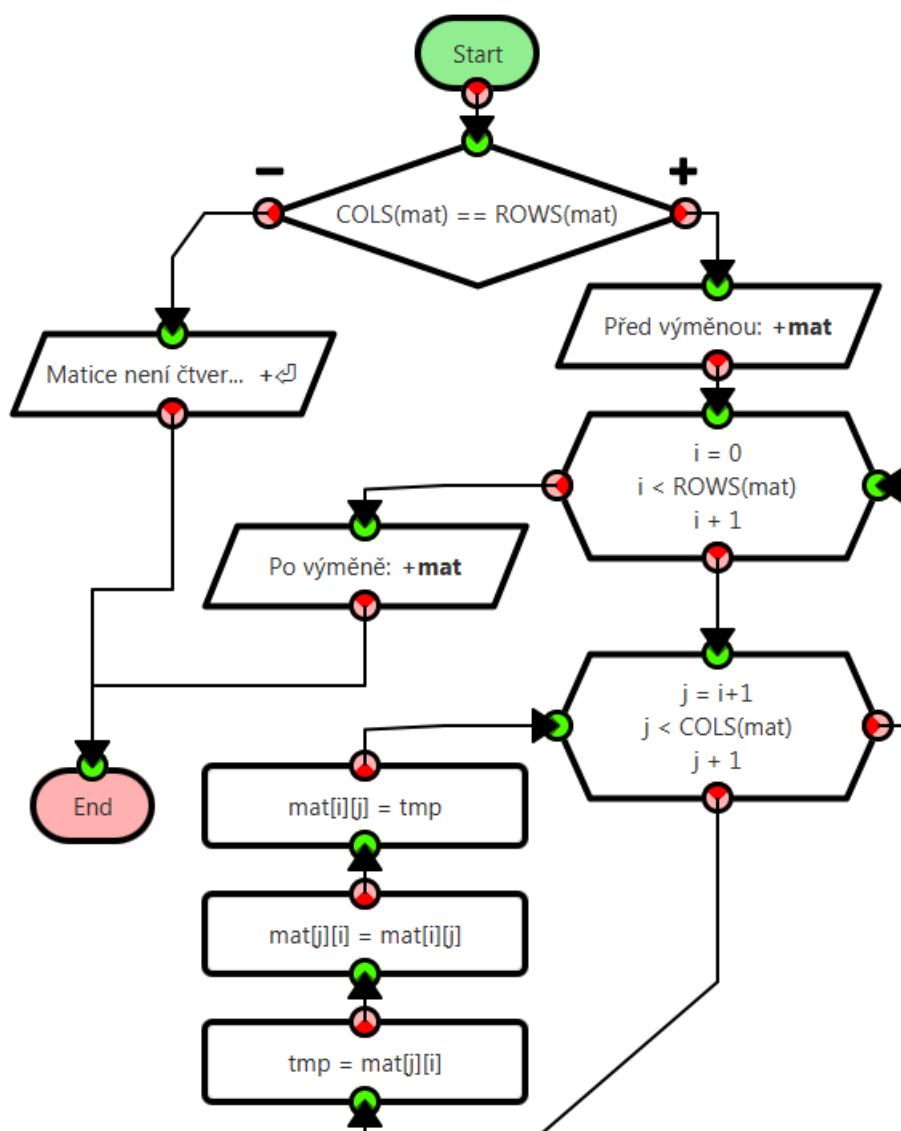
Bubble Sort patří do skupiny třídících algoritmů. Algoritmus seřadí libovolné pole A od nejnížší hodnoty po nejvyšší. Na začátku vypíše algoritmus počáteční stav pole A , uloží velikost pole do proměnné n a nastaví proměnnou *swapped* na hodnotu 0. V cyklu poté porovnává postupně vždy 2 po sobě následující prvky v poli. Pokud má předchozí prvek vyšší hodnot než prvek následující, jsou pomocí pomocné proměnné *tmp* prohozeny a proměnná *swapped* reprezentující záměnu prvků je nastavena na hodnotu 1. Po ukončení cyklu je proměnná n snížena o hodnotu 1 a cyklus se opakuje, dokud během procházení pole dochází k záměně prvků. Nakonec algoritmus vypíše seřazené pole na výstup.



Obrázek 25 – Algoritmus Bubble Sort

5.4 Algoritmus pro výpočet transponované čtvercové matice

Algoritmus nejprve ověří, je-li matice *mat* čtvercová. Pokud není čtvercová, vypíše chybovou zprávu a skončí. V opačném případě vypíše počáteční stav matice *mat* a inicializuje vnější cyklus pro procházení řádků matice. Vnitřní cyklus prochází pouze prvky nad hlavní diagonálou a provádí jejich záměnu s příslušnými prvky pod hlavní diagonálou za pomoci pomocné proměnné *tmp*. Na závěr algoritmus vypíše stav matice *mat* po transpozici.



Obrázek 26 – Algoritmus pro prohození prvků matice podle hlavní diagonály

Závěr

Existuje celá řada nástrojů pro tvorbu vývojových diagramů. Valná většina z nich však u vizuálního návrhu diagramu končí. Tyto aplikace jsou především určeny pro tvorbu dokumentace a obecných algoritmů nejen z oblasti výpočetní techniky. Možnost testovat vytvořené algoritmy v nich však chybí.

Podle zadání byla navržena a implementována aplikace pro tvorbu vývojových diagramů a sledování jejich evoluce. Za tímto účelem bylo navrženo jednoduché grafické uživatelské rozhraní umožňující návrh vývojových diagramů a jejich testování. Grafické rozhraní také umožňuje sledování stavu proměnných po celou dobu průběhu evoluce.

Za účelem práce s proměnnými byl vyvinut systém pro dynamické vyhodnocování aritmetických výrazů v průběhu evoluce. Aplikace byla navržena především jako nástroj pro podporu studia předmětu Základy Algoritmizace a s předpokladem minimálních znalostí programovacích jazyků ze strany uživatelů.

Kromě požadovaných funkcí pro import a export vytvořeného diagramu umožňuje aplikace uložit navržený diagram jako obrázek pro dokumentační účely. Z hlediska evoluce umožňuje aplikace postupovat manuálně po jednotlivých krocích, automaticky s nastavitelným intervalem, nebo rychlý průběh bez vizualizace. Kromě jednoduchých číselných proměnných umožňuje aplikace i práci s jednorozměrnými poli a maticemi.

Přestože byly požadavky na aplikaci splněny, existuje zde široký prostor na další vývoj a vylepšení aplikace. V budoucnu by například mohla být přidána možnost vytvářet proměnné v průběhu evoluce nebo alespoň měnit v průběhu velikost polí a matic. Dalším krokem pro zlepšení přehlednosti diagramu by mohla být možnost nahradit dlouhé hrany spojovací značkou. Dalším technickým zlepšením by byl přechod na nativní knihovnu pro dialogová okna, která byla do platformy JavaFX začleněna teprve nedávno.

Literatura

ČSN ISO 5807. *Zpracování informací: Dokumentační symboly a konvence pro vývojové diagramy toku dat, programu a systému, síťové diagramy a diagramy zdrojů systému*. Praha: Český normalizační institut, 1996.

Java (programming language). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-04-26]. Dostupné z: [http://en.wikipedia.org/wiki/Java_\(programming_language\)](http://en.wikipedia.org/wiki/Java_(programming_language))

JavaFX. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-04-26]. Dostupné z: <http://en.wikipedia.org/wiki/JavaFX>

NetBeans. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-04-27]. Dostupné z: <http://cs.wikipedia.org/wiki/NetBeans>

Extensible Markup Language. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-04-29]. Dostupné z: http://cs.wikipedia.org/wiki/Extensible_Markup_Language

BROWN, Greg. 2011. Introducing FXML: A Markup Language for JavaFX. *FX Experience*. [Online] 15. 8 2011. [Citace: 26. 4 2015.] <http://fxexperience.com/wp-content/uploads/2011/08/Introducing-FXML.pdf>.

ČADA, R., KAISER, T. a RYJÁČEK, Z. 2004. *Diskrétní matematika*. Plzeň : Západočeská univerzita, 2004.

DAMELIO, Robert. ©2011. *The Basics of process mapping. 2nd ed.* New York : CRC/Productivity Press, ©2011. ISBN 9781563273766.

HORDĚJČUK, Vojtěch. © 2008. Asymptotická složitost. *ing. Vojtěch Hordějčuk - Digitální Wiki*. [Online] © 2008. [Citace: 20. 4 2015.] <http://voho.cz/wiki/asymptoticka-slozitost/>.

JONES, Meilir. 2001. *Základy objektově orientovaného návrhu v UML. Vyd. 1.* Praha : Grada, 2001. ISBN 80-247-0210-X.

KONIECZNY, Jakub. 2011. JavaBlock. *SourceForge*. [Online] 2011. [Citace: 4. 5 2015.] <http://javablock.sourceforge.net/?show=main>.

MENCL, Michal. 2012. Základní principy a pojmy objektově orientovaného programování. *Péhápků.cz*. [Online] 2012. [Citace: 22. 4 2015.] <http://pehapko.cz/oop/uvod>.

MIČKA, Pavel. 2011. Asymptotická složitost. *Algoritmy.net*. [Online] 2011. [Citace: 20. 4 2015.] <http://www.algoritmy.net/article/102/Asymptoticka-slozitost>.

PECINOVSKÝ, Rudolf. 2007. *Návrhové vzory. Vyd. 1.* Brno : Computer Press, 2007. ISBN 978-80-251-1582-4.

VIRIUS, Miroslav. 1995. *Základy algoritmizace.* Praha : ČVUT, 1995. ISBN 80-01-01346-4.

WILSON, Terry, a další. 2011. RAPTOR - Flowchart Interpreter. *Martin Carlisle's Home Page.* [Online] 2011. [Citace: 4. 5 2015.] <http://raptor.martincarlisle.com/>.

Příloha A – Systémové požadavky

Doporučené systémové požadavky:

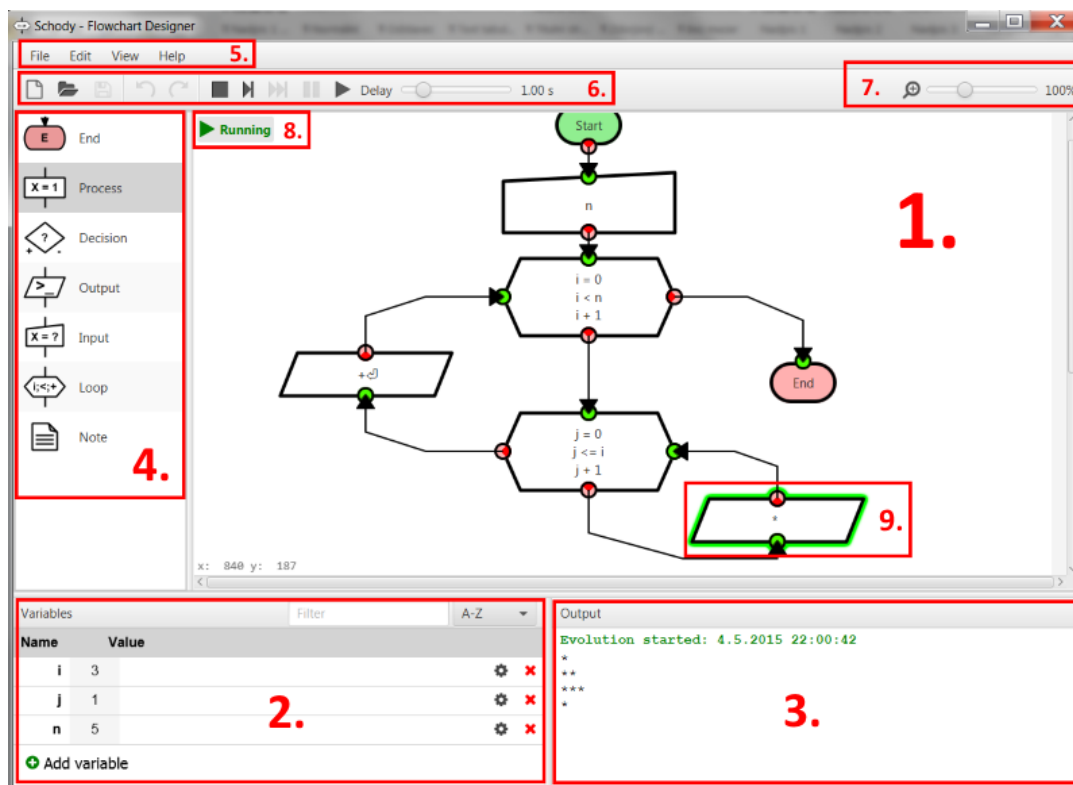
- Nainstalované prostředí JRE ve verzi 8u40 a vyšší,
- 512MB RAM,
- 2+ MB na pevném disku,
- obrazovka s rozlišením alespoň 1024 x 768 px,
- myš a klávesnice.

Příloha B – Uživatelská příručka

1. Instalace

Program nevyžaduje instalaci. Pro spuštění stačí rozbalit přiložený archiv a spustit soubor „Flowchart Designer.jar“. Pro spuštění je nezbytné mít nainstalováno prostředí JRE ve verzi alespoň 8u40.

2. Hlavní okno



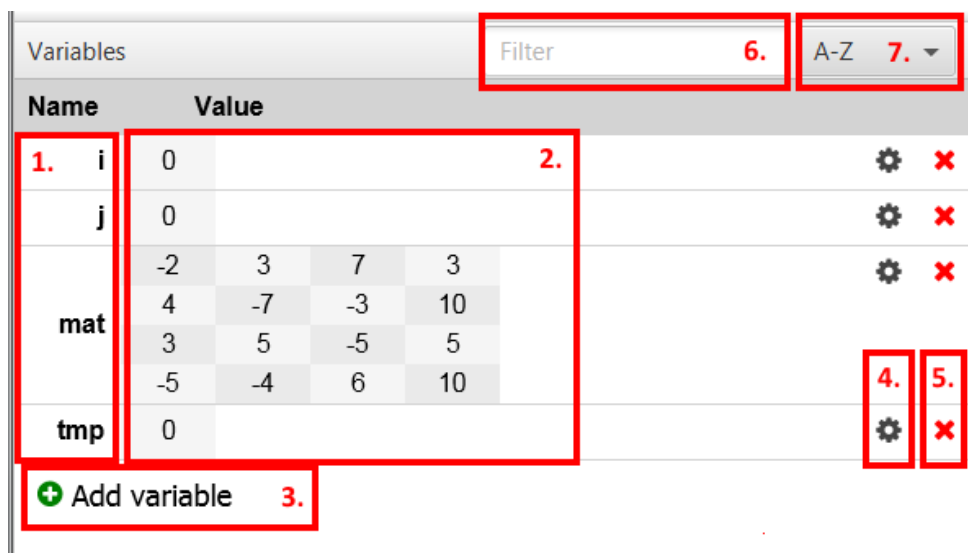
Obrázek 27 – Hlavní okno aplikace

Obrázek 27 – popis:

1. Pracovní plocha.
2. Panel proměnných.
3. Panel výstupu – Zobrazuje konzolový výstup z bloku Výstup (Output).
4. Panel bloků – Všechny dostupné bloky pro přidání do modelu.
5. Hlavní menu.
6. Panel nástrojů – Slouží především k řízení evoluce.
7. Změnit měřítko.
8. Indikátor stavu – V průběhu evoluce zobrazuje aktuální stav.
9. Aktuální blok – V průběhu evoluce je aktuální blok zvýrazněn zelenou barvou.

3. Panel proměnných

Panel proměnných se nachází v levém dolním rohu hlavního okna. Na tomto panelu se zobrazuje aktuální stav všech definovaných proměnných. Všechny proměnné musí být před použitím v modelu (v aritmetických výrazech) definovány.



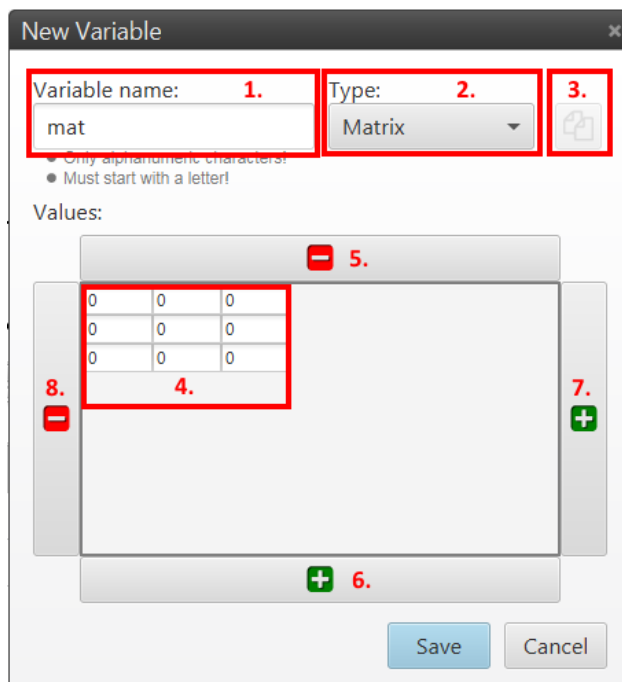
Obrázek 28 – Panel proměnných

Obrázek 28 – popis:

1. Názvy proměnných,
2. aktuální hodnoty proměnných,
3. přidat novou proměnnou,
4. modifikovat existující proměnnou,
5. smazat proměnnou,
6. filtrovat proměnné podle části názvu,
7. seřadit proměnné.

4. Editace a vytváření proměnných

Kliknutím na volbu přidání nebo modifikace existující proměnné se spustí dialog pro editaci proměnných.



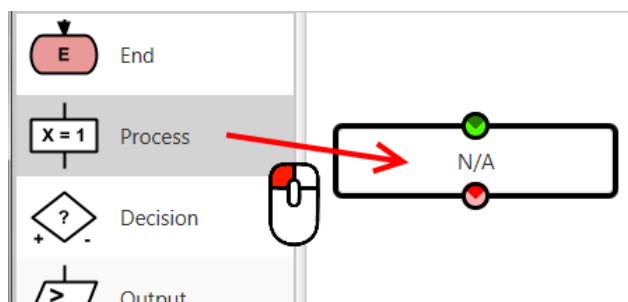
Obrázek 29 – Dialog pro editaci proměnných

Obrázek 29 – popis:

1. **Název proměnné** – dostupné jen při vytváření nové proměnné. Název musí být unikátní, začínat velkým nebo malým písmenem a obsahovat pouze písmena, číslice nebo znak podtržítko (_).
2. **Typ proměnné** – Podporované jsou reálné číslo (Number), pole čísel (Array) a matice (Matrix).
3. **Duplikovat proměnnou** – Dostupné je při editaci existující proměnné. Přepne dialog do režimu vytvoření proměnné a zachová aktuální hodnoty a typ.
4. **Hodnoty proměnné** – Nastavení počátečních hodnot proměnné. Výchozí hodnotou je hodnota 0.
5. **Odebrat řádky** – Přidá řádky do proměnné typu matice.
6. **Přidat řádky** – Odebere řádky z proměnné typu matice.
7. **Přidat sloupce** – Přidá prvky do proměnné typu pole nebo sloupce do proměnné typu matice.
8. **Odebrat sloupce** – Odebere prvky z proměnné typu pole nebo sloupce z proměnné typu matice.

5. Přidávání bloků

Požadovaný blok přidáme dvojitým kliknutím levého tlačítka myši na daný blok v panelu bloků v levé části hlavního okna. Alternativním způsobem je tažení myši z panelu bloků na pracovní plochu při držení levého tlačítka.

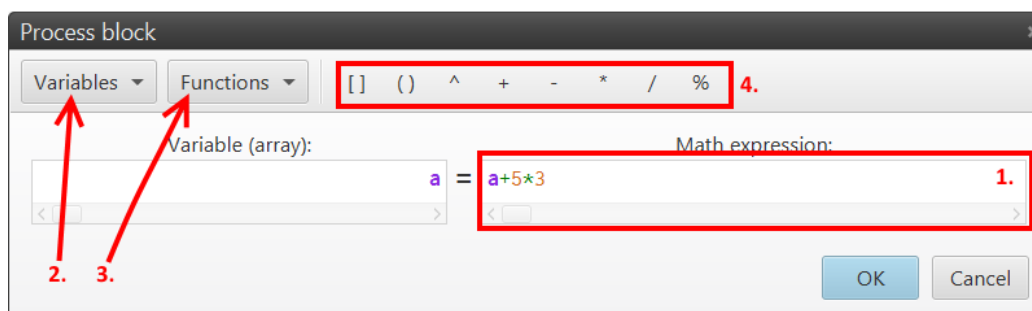


Obrázek 30 – Přidání bloku do modelu

Pro přesunutí vloženého bloku po pracovní ploše najedeme na daný blok kurzorem a tažením myši při držení levého tlačítka blok přesouváme.

6. Editace bloků

Kliknutím pravého tlačítka myši na blok na pracovní ploše se otevře kontextové menu umožňující blok editovat, duplikovat, smazat a příp. změnit jeho orientaci. Zvolením možnosti editovat se otevře dialogové okno pro editaci daného bloku. V závislosti na typu bloku se mohou dialogová okna lišit.



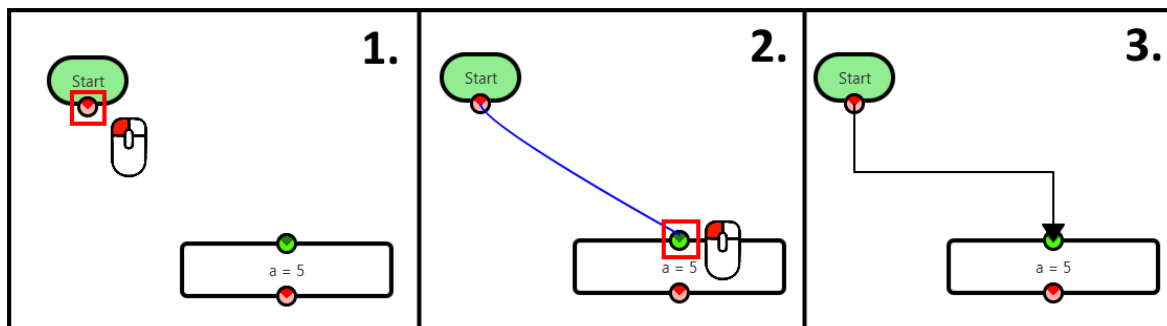
Obrázek 31 – Dialog pro editaci bloku Proces

Obrázek 31 – Popis:

1. **Editor výrazů** – Slouží pro zadání aritmetických výrazů. Podporuje barevné rozlišení syntaxe a spárování závorek.
2. **Definované proměnné** – Umožňuje vložit definované proměnné do editoru výrazů nebo definovat nové proměnné.
3. **Dostupné funkce** – Obsahuje všechny podporované funkce, které lze vložit do editoru výrazů.
4. **Operátory** – Tlačítka pro rychlé vložení operátorů do editoru výrazů.

7. Vytváření relací (spojnic)

Relace se vytvářejí kliknutím levého tlačítka myši na výstup bloku, ze kterého má daná relace vést, a následným kliknutím levého tlačítka myši na vstup bloku cílového.



Obrázek 32 – Postup vytvoření relace

Obrázek 32 – popis:

1. Tvorbu relace zahájíme kliknutím levého tlačítka myši na libovolný výstup.
2. Tvorbu dokončíme kliknutím levého tlačítka na libovolný vstup. Kliknutím pravého tlačítka kdekoli operaci přerušíme.
3. Dokončená relace.

Z každého výstupu může vést nejvýše 1 relace. Vytvoření další předchozí relaci odstraní.

8. Panel nástrojů



Obrázek 33 – Panel nástrojů

Obrázek 33 – popis tlačítek:

1. Vytvoří nový (prázdný) model.
2. Otevře dialog pro import uloženého modelu.
3. Uloží aktuální model do souboru.
4. Vrací zpět provedené změny v modelu.
5. Obnovuje vrácené změny.
6. Resetuje model v průběhu evoluce do výchozího stavu.
7. Posune evoluci o 1 krok. Alternativou v průběhu evoluce je klávesa Enter.
8. Provede evoluci najednou a bez vizualizace. Pouze zobrazí konečný stav a výstup.
9. Pozastaví automatický posun.
10. Spustí automatický posun evoluce.
11. Posuvník nastavuje prodlevu v sekundách mezi jednotlivými kroky při automatickém posunu.

9. Pracovní plocha

Pracovní plocha je prostor určený pro sestavení modelu. Pro posouvání pohledu po pracovní ploše lze využít posuvníky po stranách nebo pomocí tažení myši po pracovní ploše při držení levého tlačítka. Velikost pracovní plochy lze změnit pomocí volby *Resize* v nabídce *Edit*.

10. Hlavní menu

- File
 - New Model – Vytvoří nový (prázdný) model.
 - Save – Uloží aktuální model do souboru.
 - Save As – Umožňuje změnit soubor pro uložení modelu.
 - Save as Image – Uloží model jako obrázek ve formátu PNG.
 - Open – Otevře dialog pro import uloženého modelu.
 - Exit – Ukončí aplikaci.
- Edit
 - Resize – Změní velikost pracovní plochy.
 - Undo – Vrací zpět provedené změny v modelu.
 - Redo – Obnovuje vrácené změny.
 - Snap To Grid – Aktivuje nebo deaktivuje přichytávání bloků na mřížku při přesouvání.
- View
 - Zoom In – Zvětší měřítko zobrazení modelu.
 - Zoom Out – Zmenší měřítko zobrazení modelu.
 - Zoom 100% – Obnoví měřítko na výchozí hodnotu.
 - Find Start – Zobrazí blok Start v modelu.
 - Auto-Scroll – Aktivuje nebo deaktivuje automatické posouvání pohledu na aktuální blok v průběhu evoluce.

Příloha C – Obsah přiloženého CD

1. Adresář: Zdrojové kódy

Obsahuje projekt pro platformu NetBeans IDE 8. Samotné zdrojové soubory jsou uloženy v podadresáři *src*. Adresář také obsahuje podadresář *lib* kde jsou uloženy externí knihovny nezbytné pro kompilaci programu. Pro kompilaci je vyžadována platforma Java SE (JDK) ve verzi 8u40 a vyšší.

2. Adresář: Vzorové příklady

Obsahuje importovatelné soubory vzorových algoritmů, včetně algoritmů z kapitoly 5.

3. Archiv: Aplikace.zip

Obsahuje spustitelný soubor „Flowchart Designer.jar“ společně s adresářem *lib* obsahujícím externí knihovny. Pro správné fungování aplikace je nezbytné, aby se soubor „Flowchart Designer.jar“ a adresář *lib* nacházely ve stejném adresáři.