Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Nástroj pro tvorbu a interpretaci vývojových diagramů

Plzeň 2022 Marek Zábran

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Marek ZÁBRAN**
Osobní číslo:	**A19N0049P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Nástroj pro tvorbu a interpretaci vývojových diagramů**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s vývojovými diagramy a dalšími metodami pro vizualizaci algoritmů.
2. Seznamte se s existujícími nástroji pro vizualizaci algoritmů.
3. Porovnejte existující nástroje a srovnejte jejich funkcionalitu s požadavky pro výuku programování na ZČU.
4. Navrhněte způsob implementace požadované funkcionality do vybraného nebo nově implementovaného nástroje.
5. Navrženou funkcionalitu implementujte.
6. Otestujte vytvořenou implementaci jak po funkční stránce, tak z hlediska UX.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Richard Lipka, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **7. září 2021**
Termín odevzdání diplomové práce: **19. května 2022**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 7. září 2021

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. května 2022

Marek Zábran

Abstract

In the teaching of algorithmization, algorithms are traditionally visualized using flowcharts. This thesis describes and evaluates flowcharts and similar visualisation methods used to teach algorithmization, especially their more practical usage as an interpretable programming language. Already existing tools for creating and interpreting flowcharts are researched and compared.

This research is further used to specify suitable properties for a custom version of such a tool, that could be used for education at UWB and potentially elsewhere. The tool is fully implemented, and testing shows that it is capable of the intended functionality. Additional features may also be easily realized due to the implemented structure.

Abstrakt

V rámci výuky algoritmizace se algoritmy tradičně vizualizují pomocí vývojových diagramů. V tomto textu jsou popsány a ohodnoceny vývojové diagramy a podobné metody vizualizace pro právě tento účel, a zvláště pak možnosti jejich praktičtějšího použití v podobě interpretovatelného programovacího jazyka. Pro tyto účely jsou prozkoumány a porovnány již existující nástroje pro tvorbu a interpretaci vývojových diagramů.

Tento výzkum je dále použit pro specifikaci vhodných vlastností vlastní verze takového nástroje tak, aby jej bylo možné použít pro výuku na ZČU, případně i jinde. Nástroj byl implementován a testování ukazuje, že je možné jej použít k určenému použití. Nástroj je navíc uzpůsoben tak, aby jej bylo možné dále rozšiřovat.

Obsah

1 Úvod

Vývojové diagramy se používají jako nástroj pro pochopení a uspořádání procesů již od roku 1921 [22]. Od tohoto roku si prošly značným vývojem a z původně ilustračního nástroje pouze pro proces výroby se stal nástroj použitelný pro popis libovolného algoritmu, zvláště pak těch počítačových [23]. I přesto se vývojové diagramy používají pro popis složitějších algoritmů jen velmi výjimečně a většinou se místo nich používá pseudokód, který je prostorově skladnější a flexibilní v tom smyslu, že nemá žádná přesně daná pravidla. To ale také výrazně znesnadňuje jeho pochopení čtenářem, zvláště pokud tento pseudokód používá struktury a vlastností, které jsou specifické pro nějaký programovací jazyk, který čtenář nezná.

Zatímco zkušený programátor se s tímto problémem nejspíš dokáže vypořádat, pro začátečníka se jedná o problém, který bez jasného vysvětlení nebo reinterpretace zpravidla není schopen překonat[1]. Výuka programování je většinou vedena ve specifickém programovacím jazyku, což vede studenty k vyvinutí tendencí a očekávání specifických pro daný programovací jazyk[2]. Kvůli tomu je následně pro studenty obtížné řešit problémy pomocí algoritmů odstíněných od programovacích jazyků, případně pracovat v jiném programovacím jazyce, a tedy i práce s pseudokódem. Tento problém bývá někdy řešen tím, že výuka programování začíná nejprve výukou algoritmizace, ve které jsou algoritmy znázorněny právě vývojovými diagramy. Na rozdíl od kódu programu není ovšem typicky možné spustit vývojové diagramy jako program, což znamená, že si student nemůže na základě jen vývojového diagramu interaktivně vyzkoušet, jestli jeho pochopení algoritmu je validní.

Tato práce se zabývá možností tvorby, překladu a interpretace vývojových diagramů na úrovni desktopové aplikace a případné možnosti použití takového nástroje ve výuce, zvláště pak k výuce na Západočeské Univerzitě v Plzni.

V kapitole 2 je vysvětlen smysl vývojových diagramů, možné alternativy, jejich běžné použití a zvláště jak vypadá současná mezinárodně uznávaná norma pro vývojové diagramy.

[1] Tento závěr je činěn na základě dvouleté praxe jak jako cvičící základů programování na ZČU, tak jako tutor programování v SSC a na BootCampu ZČU FAV.

[2] Například student, jehož první programovací jazyk je Python, dbá se zbytečnou precizností na odsazení kódu, ale typicky má velké problémy se psaním a zvláště navazováním závorek.

V kapitole 3 jsou představeny existující nástroje pro tvorbu a interpretaci vývojových diagramů, včetně jejich pozitivních a negativních stránek, a přiblíženo je také použití tohoto typu nástrojů v rámci výuky programování v České republice.

V kapitole 4 jsou shrnuty možné funkce a vlastnosti, které nástroj pro interpretaci vývojových diagramů může mít. Tyto vlastnosti jsou rámcově kategorizovány dle užitečnosti ve výuce a přiblížena je také výuka programování v rámci Západočeské Univerzity v Plzni. Kromě toho kapitola také vysvětluje důvody pro některá rozhodnutí, ke kterým došlo při vývoji aplikace v kapitole 6.

V kapitole 5 jsou představeny nástroje a grafické knihovny pro vizualizaci vývojových diagramů a grafů v rámci programovacího jazyka *C#*. Kapitola shrnuje jejich pozitivní a negativní stránky a vysvětluje dilema, které nejprve vedlo ke zvolení knihovny *MSAGL*, jen aby až po samotné implementaci byla odstraněna a nahrazena vlastním systémem.

V kapitole 6 je představen implementovaný nástroj pro interpretaci vývojových diagramů, včetně všech plánovaných funkcionalit, architektury a přesného popisu, jakým jsou vývojové diagramy interpretovány.

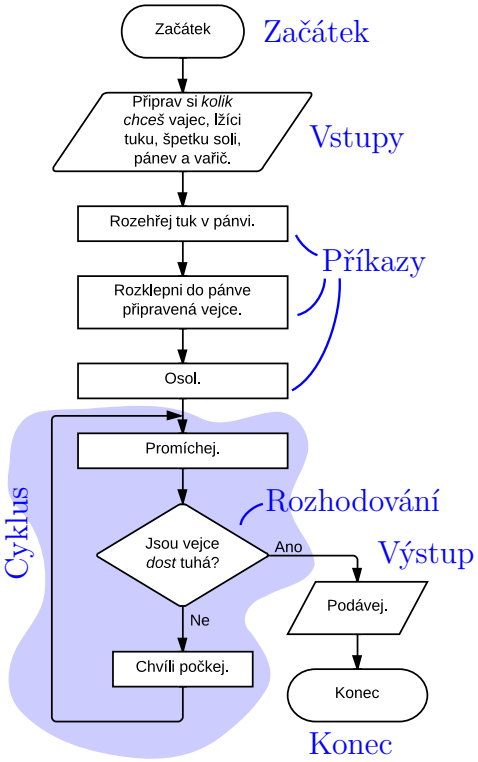
V kapitole 7 je popsán přesný vývoj aplikace již od stádia návrhu, včetně způsobů testování aplikace v dané vývojové fázi, výsledků a diskuzí nad stavem aplikace, které vedly k rozhodnutím ve vývoji aplikace.

V kapitole 8 je pak vysvětleno, jak byla implementovaná aplikace testována, včetně diskuzí a závěrů z těchto dat usouditelných, a také jakým způsobem se aplikace může dále rozšiřovat.

2 Vývojové diagramy

Vývojový diagram obecně je typ grafu, který popisuje proces nebo pracovní postup [3]. V rámci tohoto díla je vývojový diagram považován zjednodušeně za vizuální popis, který je izomorfní k počítačovému algoritmu, který popisuje. Základními stavebními prvky vývojového diagramu jsou hrany a vrcholy s různými vlastnostmi, vývojové diagramy lze tedy chápat jako rozšíření grafů z teorie grafů. V rámci vývojového diagramu se vyskytují pouze orientované hrany, které určují směr, kterým algoritmus postupuje. Tento směr se typicky znázorňuje shora dolů s výjimkou v případě, že je ve vývojovém diagramu znázorněn cyklus. Vývojový diagram je rovněž, až na větvení způsobená podmínkou anebo cyklem, grafem lineárním (sekvenčním). Vrcholy grafu nabývají různých tvarů a obsahu a znázorňují jednotlivé akce, ke kterým v rámci algoritmu dochází. Jedná se zvláště o přesuny dat, volání podprogramů, vstup a výstup, podmínkové rozhodování a cykly.

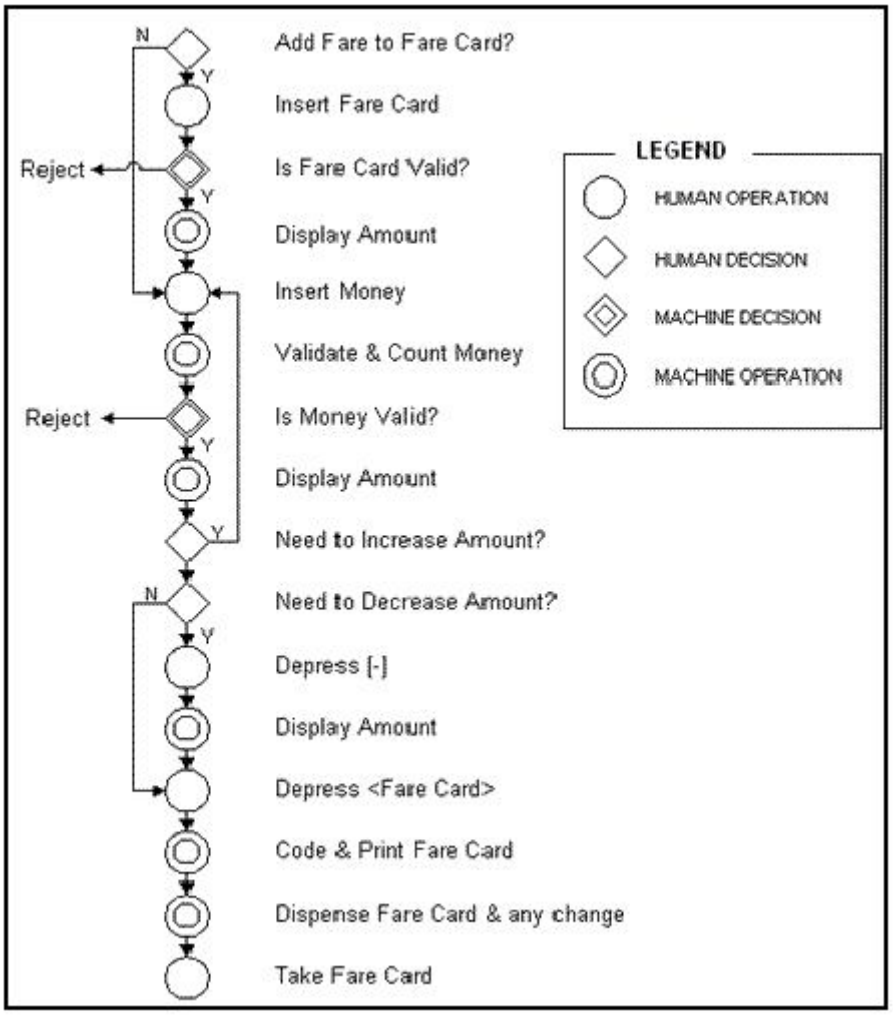
Pro ilustraci může sloužit například Obr. 2.1.



Obrázek 2.1: Vývojový diagram popisující algoritmus přípravy míchaných vajíček. Popisuje různé typy vrcholů. Pod výrazem *příkaz* si můžeme představit volání podprogramu. [29]

2.1 Použití a historie vývojových diagramů

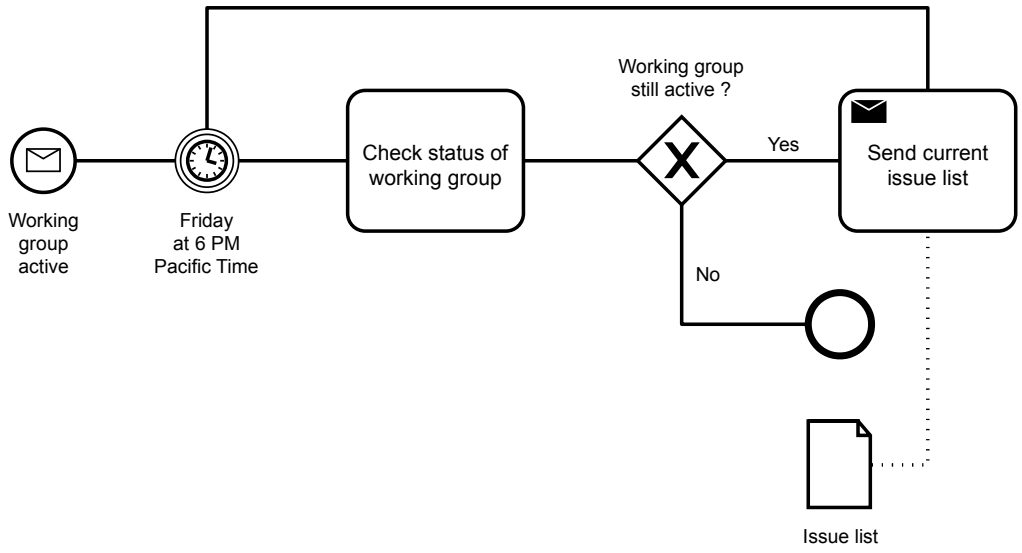
První typ grafu, který je možné zařadit do skupiny vývojových diagramů, byl popsán v roce 1921 Frankem a Lillianou Gilbrethovými a šlo o diagram pracovního procesu (*flow process chart*) [22]. Již tento typ grafu obsahoval všechny nutné části vývojového diagramu, jak je možné vidět na Obr. 2.2. Vývojové diagramy pro popis (počítačových) algoritmů vyvinul až Herman Goldshine a John von Neuman, a to nejspíš v roce 1947 - původní článek ovšem nevyšel a byl vytištěn až zpětně v roce 1963, poprvé tak byly popsány až v roce 1949 [23].



Obrázek 2.2: Příklad diagramu pracovního procesu. [2]

Vývojové diagramy byly hojně používány pro popis počítačových algoritmů zvláště v 50. a 60. letech, kdy koncept digitálního počítače (architektura Turing/von Neuman) přestával být pouhým konceptem. Později s rozvojem počítačů ovšem přestal být oblíbený a většina programátorů (tehdy spíše matematiků) začala raději zapisovat algoritmy přímo v kódu programovacích jazyků, který bylo možné přímo použít pro naprogramování programu, a tedy praktické vyzkoušení správnosti algoritmu. Vývojové diagramy se následně v programování používaly spíše pro vysvětlení algoritmizace[1] pro začínající programátory a zápis jednoduchých algoritmů. Vývojové diagramy jsou totiž v porovnání s programovacími jazyky a pseudokódy z principu náročné na množství místa a trvá déle je nakreslit.

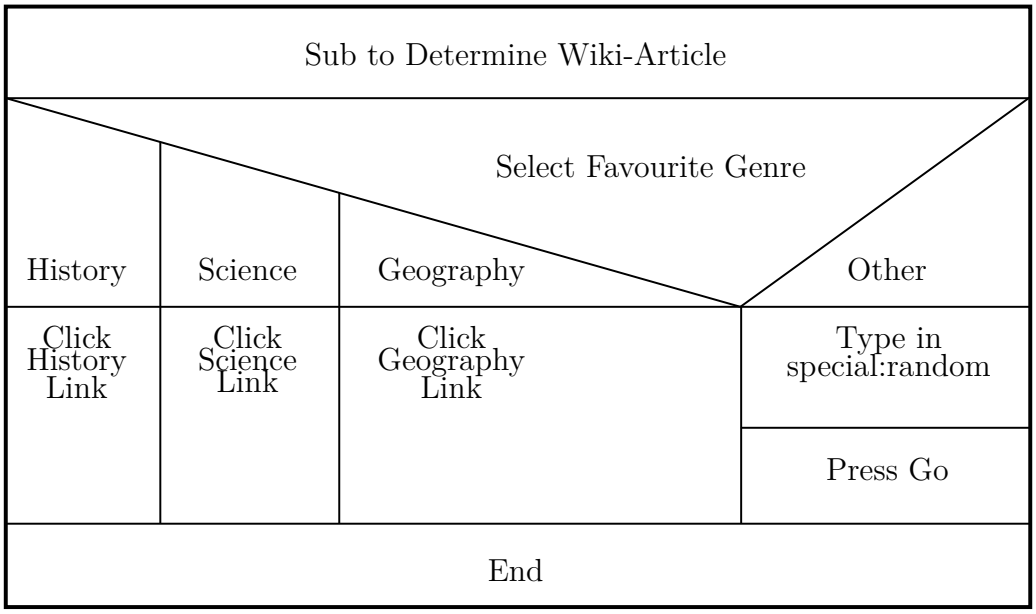
Vývojové diagramy se dnes mimo výuku základů programování používají nejčastěji pro popis pracovního postupu a výrazně vypomáhají s jeho ilustrací pro stakeholdery. Někdy se pro tento účel také používá specializovaný vývojový diagram pro business pod názvem *BPMN* (*Business Process Model and Notation*/*Business Process Modeling Notation*)[14]. Příklad BPMN je na Obr. 2.3.



Obrázek 2.3: Příklad vývojového diagramu typu BPMN. [42]

[1] Algoritmizací se rozumí postup vytvoření algoritmu řešící zadaný problém. Zpravidla vzniká rozkladem problému na podproblémy, které je možné řešit kombinací elementárních příkazů a jejich složení na celkový algoritmus.

Z hlediska potřeb informatiky a hlavně softwarového inženýrství také došlo ke specializaci vývojových diagramů. Například v 70. letech došlo k pokusu používat strukturogramy (Nassi-Schneidermanovi diagramy), které jsou skladnější než běžné vývojové diagramy díky absenci hran[2] a vhodné zvláště pro strukturové programování [49]. Nikdy ovšem nebyly výrazně rozšířené (s výjimkou Německa, kde se vyskytuje v několika učebnicích programování. Například [11].), dnes se s nimi jde setkat v rámci aplikace Microsoft Office Visio. Příklad je možné vidět na Obr. 2.4.



Obrázek 2.4: Příklad strukturogramu, který reprezentuje switch. [41]

Dalším pokusem o vylepšení vývojových diagramů jsou například v Rusku (původně ještě v 80. letech v Sovětském Svazu v rámci Sovětského vesmírného programu) vyvinutá technika DRAKON [31], která se dočkala i interaktivně a vizualizačně interpretačního nástroje. Víc o DRAKONu v sekci 3.2.1.

Jako následovníka vývojových diagramů lze označit také část UML diagramů – diagramy aktivit svou strukturou odpovídají vývojovým diagramům a například stavové a sekvenční diagramy lze z principu také zařadit mezi grafy procesů, které také mají s vývojovými diagramy mnoho společného.

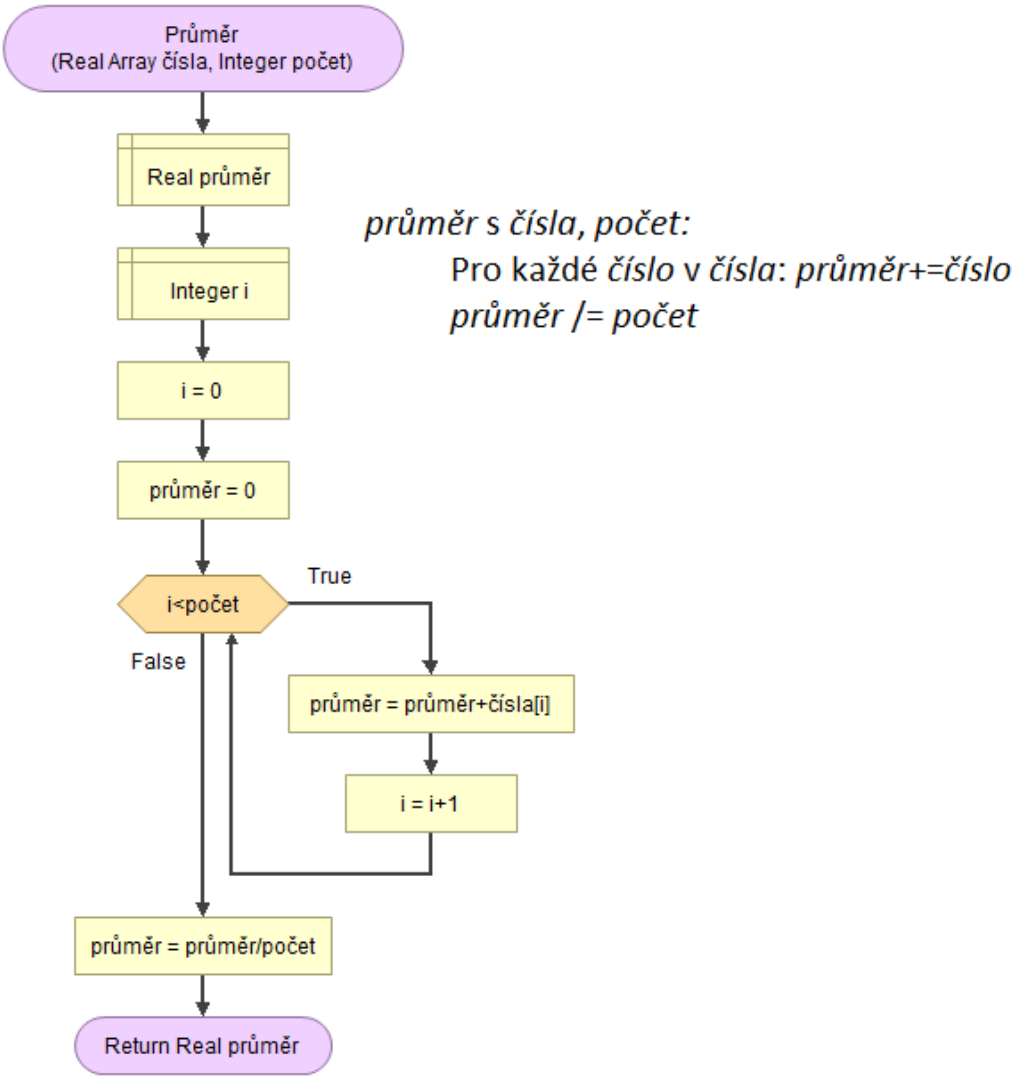
[2]Mají také sklony více růst do šířky, která je ale většinou u vývojového diagramu nevyužita. Je to ovšem problematické v případě velkého množství větvení algoritmu.

2.2 Vývojové diagramy a pseudokód

Pokud se v odborné literatuře vysvětluje nový algoritmus, pak se k tomuto vysvětlení téměř nikdy nepoužívají vývojové diagramy, ale pseudokód. Pseudokód je výrazně flexibilnější a dovoluje jeho tvůrci použít i nové konstrukce, které jsou specifické k danému algoritmu. Vývojovým diagramům (alespoň dle normy ISO 5807-85) navíc chybí symboly pro mnohé běžně používané struktury, jako například pole, seznam, slovník nebo i obyčejný for-cyklus.

Všechny vývojové diagramy lze přesně přepsat do pseudokódu, při přepisu pseudokódu do vývojového diagramu je ovšem nutné přepsat také struktury a konstrukce, se kterými pseudokód pracuje a které typicky ve vývojovém diagramu chybí. Tato flexibilita pseudokódu je ovšem také jeho neduhem, neboť ne všechny programovací jazyky podporují tuto strukturu nebo konstrukci a pseudokód tedy z principu není na rozdíl od vývojového diagramu univerzální.

Tento problém je ilustrován na Obr. 2.5.



Obrázek 2.5: Porovnání vývojového diagramu (vlevo) a pseudokódu (vpravo) algoritmu pro vypočítání průměru z množiny čísel. Vývojový diagram je zakreslen ve Flowgorithmu.

2.3 Norma ISO 5807-85

Norma ISO 5807-85 [3] je mezinárodně platná norma pro dokumentační symboly a konvence pro vývojové diagramy toku dat, programu a systému, síťové diagramy programu a diagramy zdrojů systému. Tato norma je identická s českou normou ČSN ISO 5807 zavedenou roku 1996 a je dodnes aktuální. Je ovšem potřeba brát v potaz, že byla vyvinuta v jiném století a s dnes již relativně zastaralým vhledem na tvorbu programů, proto jsou mnohé sym-

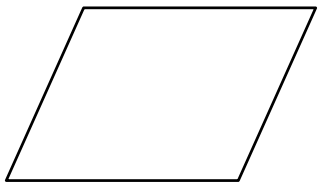
boly neaktuální (například symbol pro děrnou pásku nebo symbol pro ruční operaci (podprogram vykonaný člověkem) se dnes již nejspíš nevyužije) nebo jsou sice stále aktuální, ale z původního popisu je obtížně pochopitelné jejich skutečné použití.

Některé ze symbolů se navíc ukázaly jako ne zcela vhodné a v praxi je tak velmi časté, že se ve vývojových diagramech používá jenom část ze symbolů této normy a mnohé symboly buď získávají mírně odlišnou funkci, než jim byla přidělena, nebo je zcela nahrazena úplně jiným symbolem. Norma ISO 5807-85 je pochopitelně určena pro obecné vývojové diagramy, zatímco tento text rozebírá pouze jeho použití, které je relevantní vzhledem pro vývojové diagramy počítačových algoritmů. Nelze tedy s jistotou konstatovat, že by norma ISO 5807-85 jako celek byla zastaralá, ale pouze to, že její použití pro interpretovatelné vývojové diagramy počítačových algoritmů je pravděpodobně do určité míry suboptimální.

Ze symbolů z normy ISO 5807-85, které jsou i v dnešní době s určitou interpretací stále relevantní pro počítačové algoritmy, se jedná o následující symboly:

2.3.1 Data

- **Data** - Obr. 2.6. Symbol představuje data a tady i libovolnou manipulaci s nimi. Nosič dat není specifikován. Dnes se často používá pro vstup a výstup z klávesnice.



Obrázek 2.6: Symbol normy ISO 5807-85 pro data.

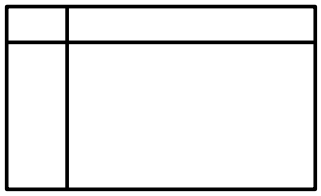
2.3.2 Uložená data

- **Uložená data** - Obr. 2.7. – Symbol představuje někde uložená data ve stavu, který by měl být vhodný pro zpracování v rámci programu. Nosič není specifikován. Tento symbol se dnes prakticky nepoužívá. Je možné ho použít pro souborový vstup a výstup, tuto funkci může ale stejně dobře převzít i symbol pro data.

Obrázek 2.7: Symbol normy ISO 5807-85 pro uložená data.

2.3.3 Interní paměť

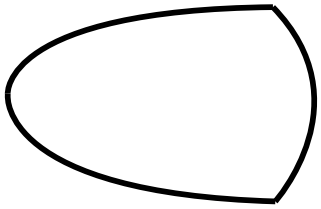
- **Interní paměť** - Obr. 2.8. – Symbol představuje data ve vnitřní paměti. Nejčastěji se používá při deklaraci proměnných nebo obecně alokaci paměti na RAM.



Obrázek 2.8: Symbol normy ISO 5807-85 pro interní paměť.

2.3.4 Zobrazení

- **Zobrazení** - Obr. 2.9. – Symbol představuje data (a odpovídající nosič dat), která jsou čitelná člověkem. Tento symbol se dnes prakticky nepoužívá, ale lze použít pro specifikaci grafického výstupu na obrazovku.



Obrázek 2.9: Symbol normy ISO 5807-85 pro zobrazení.

2.3.5 Zpracování

- **Zpracování** - Obr. 2.10. – Symbol představuje jakoukoliv operaci vedoucí ke změně hodnoty, formy nebo umístění informací, případně ke změně informace o směru toku dat. Jedná se o nejčastěji používaný symbol důležitý hlavně pro přiřazení do proměnné, případně je jej možné použít i pro volání podprogramu, který mění svůj parametr zadaný odkazem. Pro běžné volání podprogramu by se ovšem měl správně použít následující symbol.

Obrázek 2.10: Symbol normy ISO 5807-85 pro zpracování.

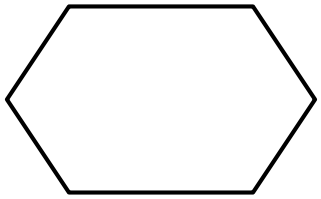
2.3.6 Předdefinované zpracování

- **Předdefinované zpracování** - Obr. 2.11. – Symbol představuje specifikovaný soubor operací definovaných jinde. Tedy libovolné volání podprogramu. Vzhledem k tomu, že v běžném programovacím jazyce může být výstupem (návratovou hodnotou) podprogramu hodnota, kterou je záhodno přiřadit do proměnné, zpracování a předdefinované zpracování můžou v některých případech splývat.

Obrázek 2.11: Symbol normy ISO 5807-85 pro předdefinované zpracování.

2.3.7 Příprava

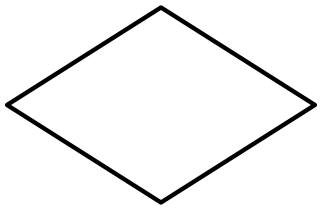
- **Příprava** - Obr. 2.12. – Symbol má představovat úpravu souboru instrukcí pro ovlivnění následující činnosti. To evokuje předzpracování, definice je ovšem natolik obecná, že tento symbol je možné použít na libovolnou úpravu následujících instrukcí. Nejčastěji se používá jako náhrada symbolu pro rozhodování nebo jako jednodušší symbol pro cykly.



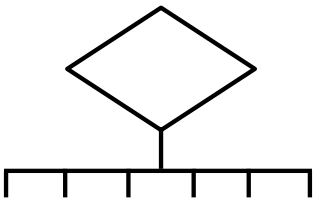
Obrázek 2.12: Symbol normy ISO 5807-85 pro přípravu.

2.3.8 Rozhodování

- **Rozhodování** - Obr. 2.13. – Symbol představuje libovolné rozvětvení programu založené na vstupní podmínce nebo přepínací funkci. Jedná se tak o podmínku nebo případně switch (Obr. 2.14). Ve vývojových diagramech se velmi často nesprávně používá i pro vytvoření cyklu, to ovšem dává smysl pouze pokud je v grafu doprovázeno spojkou.



Obrázek 2.13: Symbol normy ISO 5807-85 pro rozhodování.



Obrázek 2.14: Symbol normy ISO 5807-85 pro rozhodování - případ použití pro switch.

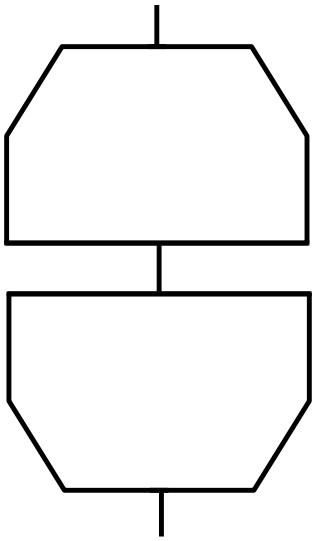
2.3.9 Paralelní režim

- **Paralelní režim** - Obr. 2.15. – Symbol představuje synchronizační bariéru pro více paralelních operací. Symbol se dnes prakticky nepoužívá, měl by ovšem využití v případě vícevláknových aplikací a paralelního programování.

Obrázek 2.15: Symbol normy ISO 5807-85 pro paralelní režim.

2.3.10 Mez cyklu

- **Mez cyklu** - Obr. 2.16. – Tato dvojice symbolů představuje začátek a konec cyklu. Obě části mají identický identifikátor (název cyklu) a hlavička cyklu (obsahující podmínku pro ukončení, inicializaci a jiné) je umístěna v právě jednom z těchto symbolů, což může indikovat charakter cyklu (tedy, jde-li o while, nebo do-while). I přes velmi častou a běžnou povahu cyklů se tento symbol prakticky nepoužívá, protože použití dvou symbolů je zbytečně zdlouhavé. Díky jasnému vymezení začátku a konce cyklu sice není možné se v cyklu ztratit, pro člověka je ovšem složité rychle spojovat oba konce cyklu na základě jen identifikátoru a další symbol navíc zhoršuje orientaci v diagramu. Pochopení cyklu jako něco, co má začátek a konec, je navíc intuitivní pouze lidem s alespoň základním vhledem do informatiky a pro laika je vyloženě zmatečné.



Obrázek 2.16: Symboly normy ISO 5807-85 pro horní a dolní meze cyklu.

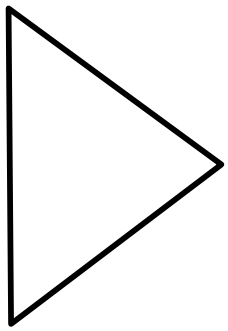
2.3.11 Spojnice

- **Spojnice** - Obr. 2.17. – Symbol představuje hranu mezi uzly. Implicitně směřuje shora dolů, ale je ji možné doplnit o šipku pro přesné určení směru například v případě cyklů.

Obrázek 2.17: Symbol normy ISO 5807-85 pro spojnici.

2.3.12 Přenos řízení

- **Přenos řízení** - Obr. 2.18. – Symbol původně představuje situaci, kdy je správa systému předána jiné části systému, tedy volání programu, událost nebo přesunu ovládání mezi člověkem a strojem. Tento symbol se dnes prakticky nepoužívá, ale je teoreticky možné jej použít v případě vícevláknových programů. Symbol je také možné použít ve zmenšené podobě pro určení směru spojnice – ze spojnice se tak stává šipka. To je nutné v případě, že spojnice nesměřuje shora dolů.



Obrázek 2.18: Symbol normy ISO 5807-85 pro přenos řízení.

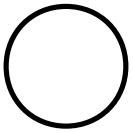
2.3.13 Přerušovaná čára

- **Přerušovaná čára** - Obr. 2.19. – Symbol představuje alternativní vztah dvou a více symbolů nebo komentovanou oblast. Například by měl být použit pro rekurentní vstup programu, k tomu se ovšem prakticky nepoužívá.

Obrázek 2.19: Symbol normy ISO 5807-85 pro přerušovanou čáru.

2.3.14 Spojka

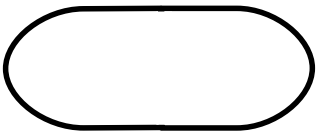
- **Spojka** - Obr. 2.20. – Symbol představuje skok na jinou spojku se stejným identifikátorem. V podstatě se jedná o skok na návěstí, v případě vývojového diagramu ovšem má zvláštní smysl, protože může pomoci zabránit křížení diagramu nebo lepší využití plochy, na které je diagram nakreslen.



Obrázek 2.20: Symbol normy ISO 5807-85 pro spojku.

2.3.15 Mezní značka

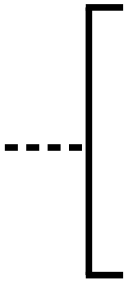
- **Mezní značka** - Obr. 2.21. – Symbol představuje začátek a konec programu nebo podprogramu a respektive také jeho vstup a výstup.



Obrázek 2.21: Symbol normy ISO 5807-85 pro mezní značku.

2.3.16 Anotace

- **Anotace** - Obr. 2.22. – Symbol představuje připojený soubor komentářů k danému uzlu.



Obrázek 2.22: Symbol normy ISO 5807-85 pro anotaci.

2.3.17 Výpustka

- **Výpustka** - Obr. 2.23. – Symbol představuje chybějící část vývojového diagramu, například kvůli nedokončené implementaci nebo pro zmenšení a zjednodušení vývojového diagramu. Častěji se ovšem místo výpustky používá bezejmenný podprogram jako black box.

Obrázek 2.23: Symbol normy ISO 5807-85 pro výpustku.

2.3.18 Co norma neobsahuje

V normě ISO 5807-85 chybí v první řadě jakýkoliv způsob pro znázornění typu dat nebo způsobu uložení - například datového typu, datové struktury nebo třeba jen pole. To jednak diagram činí zmatečným podobným způsobem jako je zmatečné číst pseudokód, jednak to žádným způsobem nevyužívá vizualizace a grafické elementy, které vývojovým diagramům pomáhají v přehlednosti.

Norma ISO 5807-85, jak už bylo vysvětleno výše, používá zbytečně složité konstrukce pro cykly, které se zpravidla nepoužívají, a zcela v ní chybí specifičtější symbol pro často používaný for-cyklus (případně foreach-cyklus).

Při zakreslení delšího algoritmu, který není rozdělen na podprogramy, je navíc diagram velmi dlouhý a typicky je jej třeba rozdělit na několik stránek.

Norma má také další nedostatky z hlediska absence specifických symbolů pro vlastní struktury, případně pro práci s objekty, které jsou dnes součástí téměř všech nejpoužívanější programovacích jazyků.

2.4 Jiné vizualizace algoritmů

Vizualizace algoritmů je těsně spjatá jednak s vizualizací (pracovních) procesů a jednak s vizualizací obecně počítačových algoritmů, která se stala předmětem zkoumání až s prvními digitálními počítači. Povětšinou se jedná o nějakou variantu vývojového diagramu a velká část jich již byla zmíněna výše. Zde je seznam relevantních způsobu vizualizace a důvod, proč nejsou pro výuku programování vhodné:

- Diagram pracovního procesu - Obr. 2.2 - Má ze základu definovány právě 4 symboly, těmi jsou pouze rozhodnutí a operace ve dvou variantách - prováděné člověkem nebo strojem. Později byl pozměněn na následujících 5 symbolů [1]: Operace, Inspekce (rozhodnutí), Přesun, Zpoždění a Uložení. Z použitých symbolů je zjevné, že diagram není vhodný pro použití pro počítačové algoritmy.

- Strukturogram - Obr. 2.4 - Vychází z vývojového diagramu a je jeho kompaktnější, ale méně přehlednou verzí. Zatímco běžný vývojový diagram je intuitivně pochopitelný, strukturogramy jsou pro nováčka zmatečné, protože neurčují přehledně směr procesu.

- DRAKON - Obr. 3.3 - Je téměř identický s vývojovým diagramem. Rozdíly jsou minimální a dále rozebrané v sekci 3.2.1.

- BPMN - Obr. 2.3 - Je rozšířením vývojových diagramů o další symboly specifikující činnosti v rámci business processu. V informatice tyto symboly prakticky nemají smysl, jedná se ovšem o dobrou ukázku toho, jak by bylo možné normu ISO 5807-85 rozšířit, aby ji bylo možné lépe použít pro počítačové algoritmy.

- UML - Diagram aktivit - Diagram velmi podobný vývojovým diagramům, který rozlišuje akce (dále nedělitelné aktivity) a vnořené aktivity (odpovídá podgrafu). Má přitom určený počáteční a konečný stav a umožňuje použití podmínek identických s podmínkami ve vývojovém diagramu. Akce navíc umožňuje ovlivňovat pomocí vstupních a výstupních podmínek, které se připojují k uzlu jako anotace a chovají se v podstatě jako trigger. Diagram aktivit celkově není pro počítačové algoritmy vhodný, vnořené aktivity ovšem slouží jako vhodná ilustrace pro fungování kolabovatelných sekcí grafu a vstupní a výstupní podmínky jsou také zajímavou možností, jejíž budoucí implementace by stála za zvážení.

- UML - Stavový diagram - Diagram vyhodnotitelný konečným automatem, skládá se z přechodů a stavů. V běžném použití se považuje každá hodnota proměnné, které může nabývat, stačilo by ovšem použít jako stav pouze pozici v kódu. Stavový diagram ovšem celkově nemá žádné vhodné vlastnosti pro reprezentaci počítačového algoritmu.

- UML - Sekvenční diagram - Diagram zachycující postup procesu v rámci času a jeho komunikaci/souběh s ostatními prvky (například vlákny). Šlo by jej využít pro organizaci vláken a v určité přepracované podobě by ho možná šlo použít i pro vzájemné volání metod mezi instancemi tříd, reprezentace samotného počítačového algoritmu v něm by byla ovšem zdlouhavá a složitá, kromě jiného také proto, že v sekvenčním diagramu se velmi nepřehledně zobrazují podmínky.

- Pseudokód - Obr. 2.5 - Má zásadní nedostatky v přehlednosti a je pro běžného začátečníka hůře čitelný než běžný programovací jazyk. Pseudokód nemá pro výuku základů programování žádnou výhodu nad kódem běžného programovacího jazyka[3].

[3]Ačkoliv je vhodné studenty naučit číst pseudokód, neboť jde o v současné době nejčastější způsob zápisu algoritmů.

3 Výukové nástroje pro programování

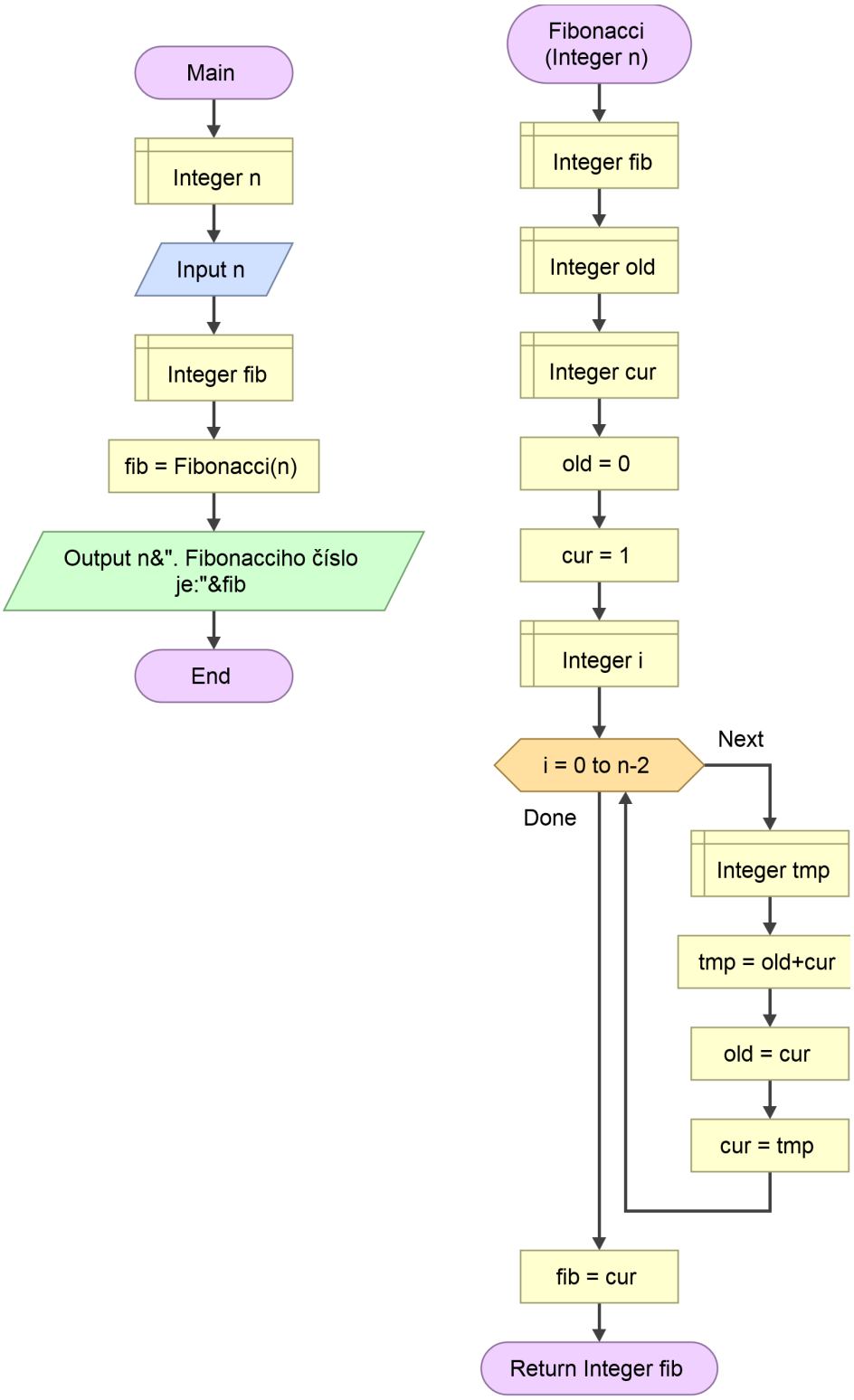
Při výuce základů programování se běžně užívají různé pomocné nástroje, které studentům pomáhají si přirozenou cestou osvojit analytické myšlení a schopnost algoritmizace, často i cestou hry nebo s programováním zdánlivě nesouvisejícím cvičením. Nejznámější z nich, alespoň v České republice, je nejspíš Robot Karel, jehož hlavní výhoda tkví v jednoduchosti pochopení řešeného problému a zapojení studenta. Procvičuje ovšem jen samotné základy a nemůže být použit obecně pro implementaci obecných algoritmů.

Výuka algoritmizace, alespoň co se týče obecných algoritmů, tak probíhá buď rovnou psaním kódu, nebo tvorbou vývojových diagramů. Vytváření vývojových diagramů má jasnou nevýhodu tkvící v tom, že jejich kreslení trvá výrazně déle a nelze je typicky přeložit jako program. První problém řeší už mnohé nástroje pro tvorbu grafů, pro výuku programování jsou ovšem zajímavější interaktivní nástroje, které umožňují vývojové diagramy nejen vytvářet, ale dokonce interpretovat jako program.

Zbytek této kapitoly se zabývá právě existujícími nástroji, které dovolují vytvářet vývojové diagramy a následně je interpretovat.

3.1 Flowgorithm

Flowgorithm [20] je vizuální programovací jazyk, který je na Západočeské univerzitě aktivně používán jednak pro výuku v rámci BootCampu Fakulty aplikovaných věd a jednak v rámci výuku na Fakultě strojní v rámci předmětu Technická informatika ve strojírenství. Z pohledu obou fakult se jedná v současné době o nejvhodnější nástroj pro výuku základů programování prostřednictvím vývojových diagramů, a proto slouží v mnoha ohledech jako vzor pro novou aplikaci. Ukázka na Obr. 3.1.



Obrázek 3.1: Příklad vývojového diagramu vytvořeného ve vývojovém prostředí Flowgorithm. Výpočet Fibonacciho posloupnosti.

Svou syntaxí odpovídá Flowgorithm vývojovému diagramu tak, jak jej určuje mezinárodní norma ISO 5807-85 s pouze drobnými rozdíly, které jsou z velké části zdůvodnitelné uživatelským komfortem a které jsou v rámci normy ISO 5807-85 stále připustitelné – konkrétně se pro cykly místo jednotlivých mezí cyklu používá jeden symbol pro zpracování.

Každý vytvářený diagram má přesně stanovený začátek a konec a hrany mezi nimi. Uživatel (i. e. programátor) může přidávat nové uzly (například přiřazení, podmínky, cykly a jiné) kliknutím na libovolné hrany mezi uzly. Tímto způsobem lze vytvořit libovolně komplexní vývojový diagram, a tedy také algoritmus a program, který je možné v rámci samotného Flowgorithmu spouštět a také krokovat. Flowgorithm má navíc některé vlastnosti, které nejsou typické pro vývojové diagramy, jak je možnost vytvořit více různých vývojových diagramů a volat je (s případným parametrem anebo návratovou hodnotou) jako podprogram, či přidávat breakpointy a komentáře jako uzly grafu. Všechny uzly je navíc možné kopírovat pomocí běžného výběru levým tlačítkem myši a Ctrl+C a následně vložit na místo jiné hrany.

Flowgorithm zaujímá hlavně svým uživatelským komfortem, intuitivností pro začátečníky a velkou mírou restrikcí, která zabraňuje začátečníkovi dělat chyby, aniž by pochopil, v čem chyba spočívá. Všechny uzly jsou barevně odlišeny a rovněž jsou zvláštní barvou označeny nevyplněné a chybné uzly, grafické nastavení všech uzlů lze navíc specifikovat použitím layoutů a barevných schémat. Flowgorithm je přeložený do třiceti různých přirozených jazyků a umožňuje dokonce i konvertovat vytvořený vývojový diagram do jednadvaceti programovacích jazyků a tří dalších pseudokódů, což výrazně pomáhá začínajícím programátorům pochopit, že kód je pouze praktickou implementací konkrétního algoritmu a že všechny procedurální jazyky stojí na stejných základech a jsou si velmi podobné.

Z hlediska komfortu není Flowgorithm perfektní – například vyžaduje před použitím proměnné vždy její deklaraci, neumožňuje ji ovšem inicializovat v rámci deklarace a ani neumožňuje proměnnou deklarovat v rámci hlavičky cyklu, což je ve většině programovacích jazyků u for-cyklu běžná praxe. Podprogramy navíc je možné volat jenom v rámci stejného projektu, není možné vytvořit projekt jako knihovnu a obecně není nijak možné vkládat podprogramy z jiného souboru *.fprg* (formát projektu Flowgorithmu, Obr. 3.2), aniž by tento soubor nebyl přímo upravován jako základ pro nový projekt.

Flowgorithm rovněž nenutí uživatele svůj projekt ukládat, což se možná zdá jako zanedbatelný detail, ale u studentů vede velmi často k tomu, že vytvořený vývojový diagram po splnění úkolu zahazují a další vývojový diagram vytvářejí zcela od začátku. To je nepříjemné hlavně u začátečníků, kteří pak často neví, jak vytvořit tu část diagramu, kterou už předtím vytvořili v jiném projektu, ale protože si původní projekt neuložili, nemůžou se na něj už podívat.

```
<function name="Main" type="None" variable="">
    <parameters/>
    <body>
        <declare name="a" type="Integer" array="False" size=""/>
        <if expression="a==0">
            <then>
                <for variable="a" start="0" end="42" direction="inc" step="1">
                    <while expression="a&lt;21">
                        <output expression="a" newline="True"/>
                        <do expression="a&lt;10">
                            <output expression="a" newline="True"/>
                        </do>
                        <output expression="a" newline="True"/>
                    </while>
                    <output expression="a" newline="True"/>
                </for>
            </then>
            <else>
                <if expression="a">
                    <then>
                        <if expression="a-1==0">
                            <then>
                                <if expression="a==a">
                                    <then>
                                        <output expression="a" newline="True"/>
                                    </then>
                                    <else/>
                                </if>
                            </then>
                            <else/>
                        </if>
                    </then>
                    <else/>
                </if>
            </else>
        </if>
    </body>
</function>
```

Obrázek 3.2: Příklad interní struktury .fprg souboru (uloženého Flowgorithm vývojového diagramu).

Flowgorithm má rovněž vážné nedostatky z hlediska výuky, protože například nepovoluje vytvoření vícerozměrných polí, neumožňuje načítat a ukládat data do souboru a v rámci hlavičky for-cyklu umožňuje řídící proměnnou pouze inkrementovat, nebo dekrementovat o určitý krok, nikoliv ji násobit, dělit nebo jakkoliv jinak upravovat. Rovněž neumožňuje žádný grafický výstup a nemá žádné interní metody pro například časování, práci s řetězci (vyjma jejich konkatenace) nebo složitější matematické operace.

Flowgorithm se vyvíjí od roku 2014 a jedná se dlouhodobě o nejlepší nástroj pro tvorbu a interpretaci vývojových diagramů, alespoň podle vedení BootCampu a vyučujících Technické informatiky ve strojírenství. Pro potřeby výuky by se ovšem hodily některé další funkce, které Flowgorithm v současné době nemá (například vícerozměrná pole) a Flowgoritm není možné upravovat - jeho zdrojový kód není volně přístupný a uživatelská licence[1] explicitně zapovídá jakékoliv pokusy o reverzní inženýring nebo dekompilaci. S autorem je navíc obtížné komunikovat, přestože odpovídá na bug reporty. Flowgorithm tedy bohužel nelze pro potřeby výuky na Západočeské Univerzitě rozšířit, pro implementaci nových funkcí je nutné vytvořit novou aplikaci.

Klady	Zápory
Barevné uzly	Chybí statická kontrola
Jasně odlišné uzly	
Vizuálně přívětivé	
Krokování	
Breakpointy	
Technický stav	
Překlad do programovacích jazyků	

3.2 Výčet známých nástrojů

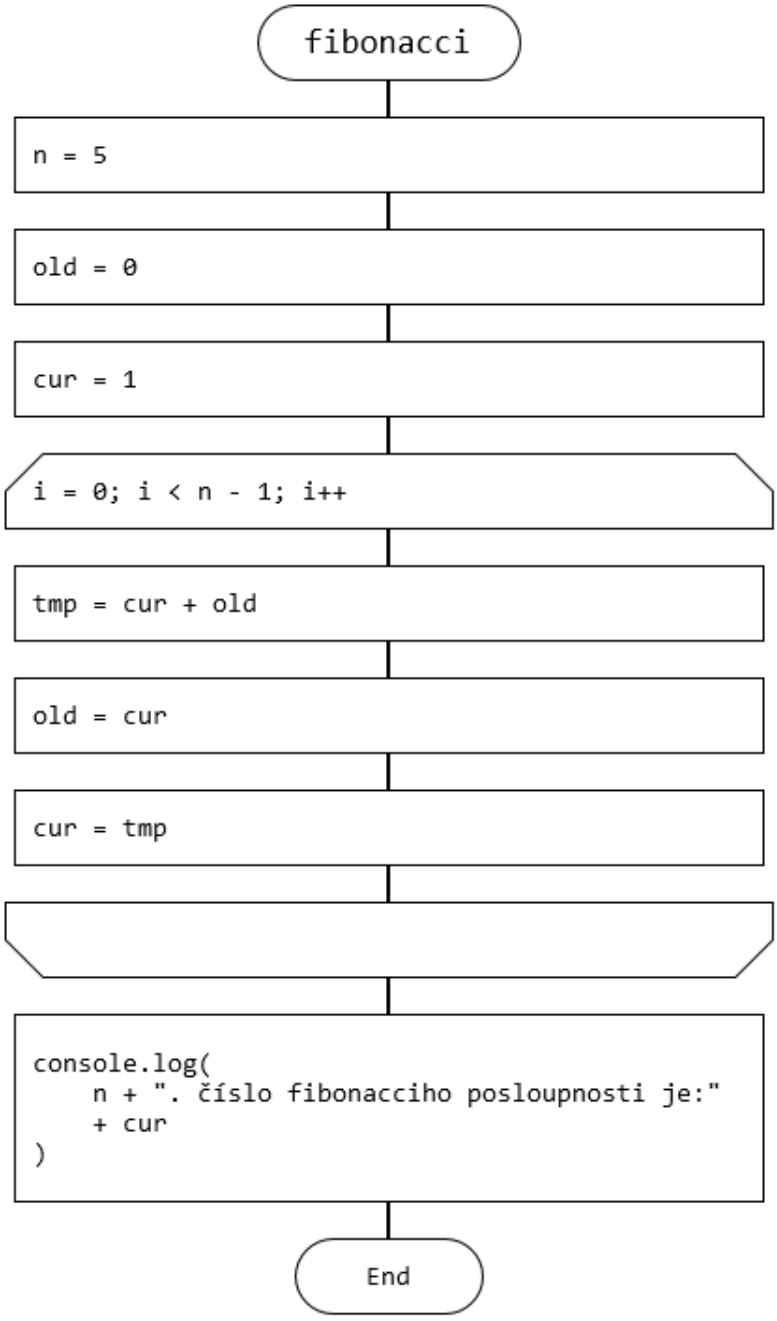
Flowgorithm není jediným existujícím nástrojem pro tvorbu a interpretaci vývojových diagramů. V této sekci je výčet ostatních známých nástrojů, které splňují stejnou nebo podobnou funkci.

[1] http://www.flowgorithm.org/about/Flowgorithm%20-%20EULA.pdf

3.2.1 DRAKON

DRAKON[31] je hybrid mezi vývojovým nástrojem a programovacím jazykem (Obr. 3.3). Využívá jiný programovací jazyk (u webové verze JavaScript), ovšem jeho jednotlivé příkazy skládá do vývojového diagramu, což sice umožňuje omezeně vizuálně programovat, jenže samotné příkazy přidávané do jednotlivých uzlů musí splňovat syntaxi programovacího jazyka a prostředí samotné uživateli nijak nenapovídá, co by měl do daného uzlu napsat. DRAKON tak sice mírně zvyšuje přehlednost u jednoduchých programů, ovšem, zvláště pro nováčky, to je za cenu zvýšení obtížnosti psaní kódu, neboť je takto třeba psát nejen kód, ale zároveň i vybrat správný uzel. DRAKON je zvláště nepřehledný také díky tomu, že má uzly na rozdíl například od Flowgorithmu ze základu odlišené pouze tvarem, a nikoliv barvou[2], a velmi často se tak stane, že člověk zamění například konec cyklu za uzel pro "akci". Zásadním problémem je pak naprostá absence jakékoliv statické kontroly kódu a obtížné používání vývojového prostředí. Velká část materiálů k němu byla navíc až do nedávna dostupná pouze v ruštině[32].

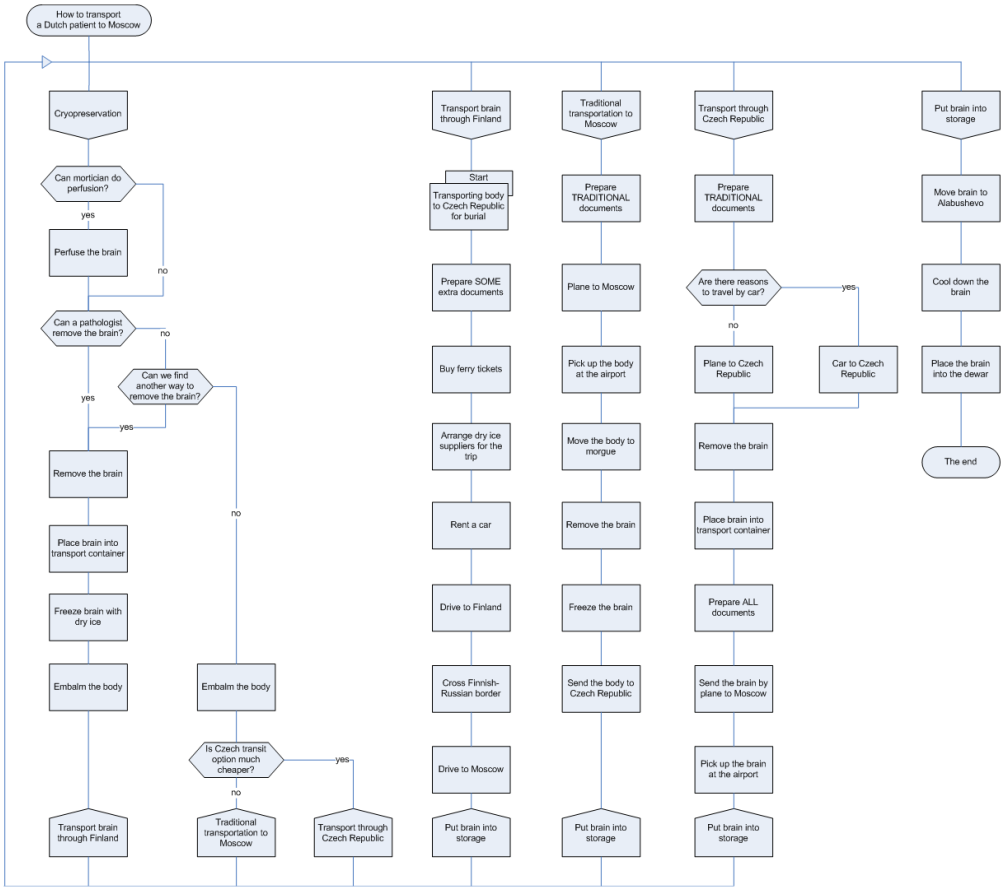
[2]DRAKON IDE umožňuje změnit grafické rozvržení některé z nich odlišují barvu některých uzlů, je ovšem potřeba pro to změnit nastavení.



Obrázek 3.3: Příklad vývojového diagramu v prostředí DRAKON IDE. Fibonacciho posloupnost.

DRAKON vychází z kombinace mezinárodní normy pro vývojové diagramy ISO 5807-85 a ruské normy «Gost 19.701-90». To vede k několika zásadním rozdílům, z nich nejdůležitější je ten, že DRAKON umožňuje rozvětvování diagramu přes „Silhouette branch“, což je zvláštní způsob větvení,

který není závislý na podmínce, má předem danou možnost, která nastane v prvním průchodu a všechny následné průchody jsou určeny výběrem dané větve. Tento „Silhouette branch“ tak může být chápán jako hybrid mezi skokem na návěstí a voláním podprogramu a značně zlepšuje kompaktnost diagramu využitím místa na šířku, které je u běžných diagramů nevyužito. To je škoda zvláště na širokoúhlém monitoru, a je zároveň problémem, na který si opakovaně stěžovali vyučující z Fakulty strojní při používání Flowgorithmu (zvláště v případě, kdy chtěli diagram exportovat do obrázku). „Silhouette branch“ je znázorněn na Obr. 3.4.



Obrázek 3.4: Příklad vývojového diagramu typu DRAKON, představuje algoritmus pro převoz mozku do Ruska pro kryoprezervaci[35]. Ukázka použití „Silhouette branch“.

Klady	Zápory
Dobrá dokumentace[3]	Chybí statická kontrola
Jasně odlišné uzly[4]	Jednobarevné[5]
Umožňuje růst do šířky	Chybí krokování
Má objekty díky použití JS	Obtížná orientace v GUI

DRAKON, jako vizuální programovací jazyk, popisuje velké množství nadstandardních symbolů, které sice nejsou implementovány ani v něm, ani v žádném jiném podobném nástroji, ale jsou používány v běžných programovacích jazycích a mělo by tedy smysl je v rámci nějakého komplexního nástroje implementovat. Autor moderní dokumentace DRAKONu ovšem přímo varuje před programovacím přístupem při vytváření vývojových diagramů na základě DRAKONu a upozorňuje, že k tomu nebyl zamýšlen. Tyto symboly je možné vidět na Obr. 3.5 a 3.6. Konkrétně k jednotlivým symbolům v porovnání s ISO 5807-85 níže:

1. Title (začátek podprogramu) - Identický s ISO 5807-85 mezní značkou shora, Sekce 2.3.15.

2. End (konec podprogramu) - Identický s ISO 5807-85 mezní značkou zdola, Sekce 2.3.15.

3. Action (příkaz) - Identický s ISO 5807-85 symbolem pro zpracování, Sekce 2.3.5.

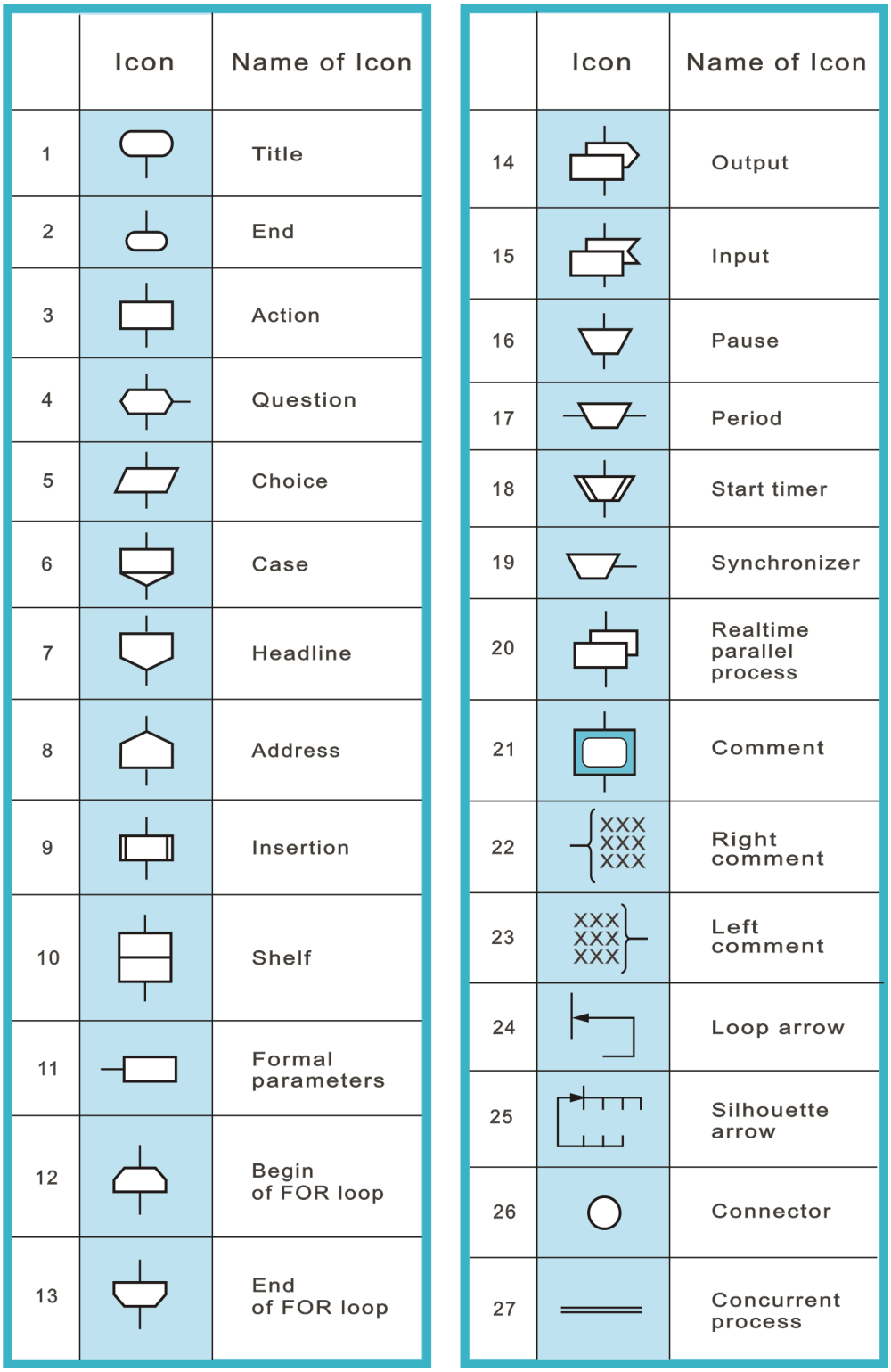
4. Question (podmínka/smyčka) - Identický s ISO 5807-85 symbolem pro přípravu, Sekce 2.3.7. Zde se ovšem používá místo kosočtvercového symbolu pro podmínku. Pro delší text uvnitř je vhodnější symbol obdélníkového tvaru, dává tedy smysl nahradit kosočtverec šestistěnem, které se graficky jeví jako kosočtverec roztáhlý do šířky.

5. Choice (přepínač) - Identický s ISO 5807-85 symbolem pro data, Sekce 2.3.1. Zde se ovšem používá jako speciální symbol pro začátek přepínače (switch). Norma ISO 5807-85 pro tento tento symbol používá stejně jako pro podmínku kosočtverec. Není jasné, proč se pro „Choice“ a „Question“ používají rozdílné znaky - přepínač je lehce odlišitelný od

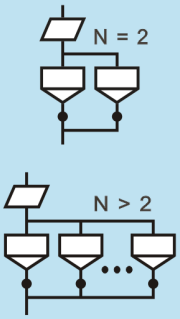
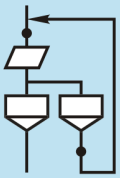
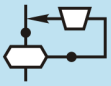
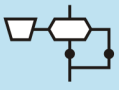
[3]Je velmi nová a místy nekonzistentní se staršími dokumenty k DRAKON.

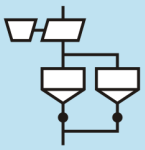
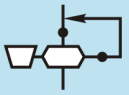
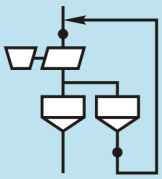
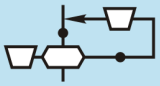
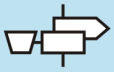
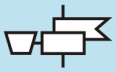
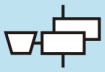
[4]Platí jen pro dokumentaci, DRAKON IDE obsahuje jen redukovanou funkcionalitu a například pro vstup a výstup je stále možné použít symbol pro „Action“, který původně funkcionalitu vstupu a výstupu zastával.

[5]Symbol „Comment“ je zobrazený v barvě, k orientaci v diagramu tedy barva výrazně nepomáhá. DRAKON IDE lze částečně přepnout, aby některé symboly barevně obarvil.



Obrázek 3.5: První část symbolů používaných v rámci jazyka DRAKON[36].

	Macroicon	Name of Macroicon
1		Title with parameters
2		Fork
3		Switch (number of cases N >= 2)
4		SIMPLE loop
5		SWITCH loop
6		FOR loop
7		WAIT loop
8		Action by timer
9		Shelf by timer
10		Fork by timer

	Macroicon	Name of Macroicon
11		Switch by timer
12		SIMPLE loop by timer
13		SWITCH loop by timer
14		FOR loop by timer
15		WAIT loop by timer
16		Insertion by timer
17		Output by timer
18		Input by timer
19		Start timer by timer
20		Parallel process by timer
21		TREE loop

Obrázek 3.6: Druhá část symbolů používaných v rámci jazyka DRAKON[37].

podmínky právě díky množině případů, tato volba se tedy jeví jako zbytečně matoucí, protože tento znak není žádným způsobem podobný znaku pro podmínku a DRAKON disponuje velkým množstvím různých znaků.

6. Case (případ) - Norma ISO 5807-85 žádný podobný symbol v rámci vývojových diagramů neobsahuje, hodnota případu se zapisuje úsporně přímo nad spojnici vedoucí k dané větvi případu. To dává smysl, pokud je hodnota případu krátká, například číselná. Pro hodnotu případu v podobě textového řetězce má ovšem DRAKON výhodu v přehlednosti.

7. Headline (začátek „Silhouette branch“) - Norma ISO 5807-85 používá podobný obrácený symbol pro začátek smyčky, symbol tedy implikuje vazbu na mez souboru instrukcí. Symbol je též podobný symbolu „Case", což dává smysl vzhledem k tomu, že má podobnou funkci. Tvar symbolu navíc obsahuje trojúhelník, který vizuálně naznačuje šipku a tedy směr, mít začátek větve širší (tedy mít symbol obráceně) dává tedy vzhledem ke tvaru symbolu smysl. Symbol má funkci návěstí pro „Silhouette branch“. V rámci normy ISO 5807-85 má podobnou funkci spojka (Sekce 2.3.14).

8. Address (konec „Silhouette branch“) - Platí totéž, co pro „Headline“, znak je jen obráceně. Hodnota odpovídá hodnotě headline (návěstí), na které se má následně pokračovat.

9. Insertion (volání podprogramu) - Identický s ISO 5807-85 symbolem pro předdefinované zpracování, Sekce 2.3.6.

10. Shelf (složená instrukce) - Norma ISO 5807-85 žádný podobný symbol neobsahuje. Symbol má dvě části, kde první určuje místo, objekt, aktéra nebo vykonávanou službu a druhý určuje hodnotu. Například: "maximum" a "Math.Max(10, maximum)", "printer" a "Print(text)", "Client>Server" a "Login", "SetValue" a "10". Tento symbol se snaží rázně oddělit dvě logicky různé části instrukce, z toho důvodu je ovšem symbol velký, méně přehledný a zcela mu chybí vazba mezi těmito částmi, kterou by jinak bylo možné v rámci jednoho celku pochopit. Jak je ukázáno na Obr. 3.3, interaktivní webová verze DRAKONu tento symbol raději nepoužívá, stejně jako v případě vstupu a výstupu, a používá pro všechno symbol pro „Action“.

11. Formal parameters - Identický s ISO 5807-85 symbolem pro zpracování, jen aplikovaný specificky pro deklaraci parametrů podprogramu jako symbol vložený napravo od symbolu „Title“. Symbol pomáhá v uvědomění, že podprogram obsahuje parametr, nepomáhá ovšem v přehlednosti mezi jednotlivými parametry, protože se v rámci DRAKONu všechny parametry daného podprogramu zapisují do jediného symbolu bez specifického oddělovače (v rámci [32] je použit konec řádky). Nezdá se tak, že by symbol oddělením parametrů od názvu podprogramu výrazně napomáhal v přehlednosti.

12. Begin of FOR loop - Identický s ISO 5807-85 symbolem pro horní mez cyklu, Sekce 2.3.10. Symbol je jen aplikován specificky pro for cyklus.

13. End of FOR loop - Identický s ISO 5807-85 symbolem pro dolní mez cyklu, Sekce 2.3.10. Symbol je jen aplikován specificky pro for cyklus.

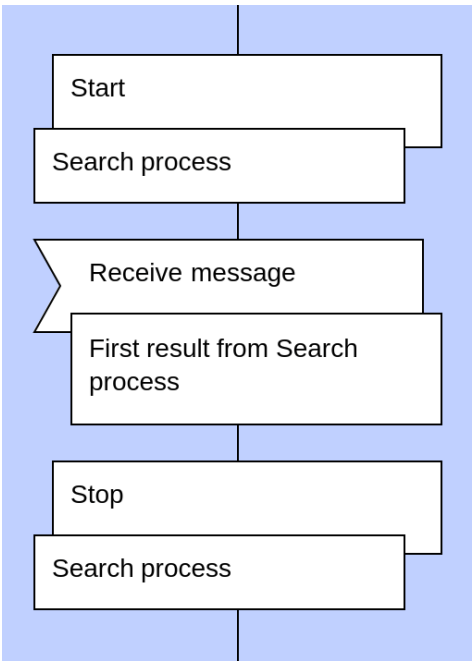
14. Output - Norma ISO 5807-85 žádný podobný symbol neobsahuje, horní část je ovšem identická s podobou výstupu ve Flowgorithmu a používá se se stejným účelem často i jinde. Stejně jako v případě „Shelf“, i tady je instrukce složitě rozdělena do dvou částí, kde druhá obsahuje hodnotu. V rámci [32] je připuštěna i zjednodušená varianta, která má pouze horní část. Dle [32] je také možné poslat výstup procesu běžícímu na pozadí (viz „Realtime parallel process“). Interaktivní webová verze DRAKONu používá pro tyto účely pouze symbol pro „Action“.

15. Input - Norma ISO 5807-85 žádný podobný symbol neobsahuje, horní část se ovšem někdy používá pro symbol vstupu i v jiných programech. Stejně jako v případě „Shelf“, i tady je instrukce složitě rozdělena do dvou částí, kde druhá obsahuje hodnotu. V rámci [32] je připuštěna i zjednodušená varianta, která má pouze horní část. Dle [32] je také možné vyžádat vstup od procesu běžícího na pozadí (viz „Realtime parallel process“). Interaktivní webová verze DRAKONu používá pro tyto účely pouze symbol pro „Action“.

16. Pause - Norma ISO 5807-85 takový symbol neobsahuje a nevyskytuje se ani v programech podobného typu. Účel instrukce je počkání - blokace interpretace - po dobu danou hodnotou tohoto uzlu. Symbol tak odpovídá časové Wait() instrukci, která ovšem není dobrým prostředkem pro synchronizaci paralelních procesů. Vhodnější by bylo použití

synchronizačních primitiv (semafor a mutex nebo i jen monitor), která ovšem DRAKON neobsahuje. Symbol je identický se symbolem pro „Period"a „Synchronizer“, což vzhledem k tomu, že všechny tři řeší čekání nebo časový limit, dává smysl.

17. Period - Symbol je identický s „Pause“, jeho použití je ovšem nejasná, protože dokumentace[32] pro „Period"používá úplně jiný symbol, viz Obr. 3.7. V dokumentaci se „Period“ vyskytuje jako samostatný úsek, označený na začátku a na konci, který musí být dokončen v předepsaném intervalu. Užití původního symbolu se nedaří dohledat, nejspíš bylo jeho původní užití nepraktické, a proto bylo změněno.



Obrázek 3.7: Obrázek použití symbolu „Control period“ v jazyce DRAKON dle moderní dokumentace[32].

18. Start timer - Symbol se podobá symbolu pro „Pause“, který má ale podobně jako „Insertion“ zdvojený pravý a levý okraj symbolu. Souvislost spuštění časovače s voláním podprogramu není jasná, symbol pro „Start timer“ ovšem navozuje dojem stisknutelného tlačítka, což společně s tvarem odpovídajícím „Pause“ a „Synchronizer“ navozuje nějakou akci s časem.

19. Synchronizer - Symbol se je identický symbolu pro „Pause“ a v dokumentaci je možné ho najít pod názvem „Duration“[32]. Symbol je možné buď použít v kombinaci se „Start timer“ - v tom případě vyjadřuje, kdy od spuštění časovače se má příkaz vykonat - nebo samostatně - v tom případě vyjadřuje, jak dlouho má příkaz trvat. Oboje použití mají smysl z hlediska lidské činnosti, z hlediska programovacího jazyka není ovšem ideální ani jedno.

20. Realtime parallel process - Symbol odpovídá dvojici symbolů pro „Action“ uspořádané stejným způsobem jako „Input“ a „Output“, díky čemuž je jasně patrná souvislost mezi těmito symboly. „Input“ i „Output“ totiž dle [32] může být použit pro komunikaci s jiným procesem běžícím na pozadí a tak zastávat funkci blokujícího příjmu (receive), respektive neblokujícího poslání. Právě správu takového procesu řeší symbol „Realtime parallel process“, horní část odpovídá operaci nad daným procesem (jmenovitě: spustit, pozastavit, pokračovat a ukončit) a spodní jej identifikuje názvem, který odpovídá názvu podprogramu, který má tento na pozadí spouštěný proces vykonávat. ISO 5807-85 samo o sobě umožňuje paralelní procesy, ovšem pouze na stejné úrovni vytvářené na principu fork s pomocí podmínky a ukončované symbolem pro paralelní režim (Sekce 2.3.9), který představuje synchronizační bariéru. DRAKON tuto variantu umožňuje také, varianta s „Realtime parallel process“ je ovšem skladnější a vhodnější pro běžné užití programátora, neboť více odpovídá způsobu, jakým k paralelizaci běžný programátor přistupuje, tedy jako k volání samostatné služby. Pro použití k optimalizační paralelizaci mu ovšem chybí synchronizační primitiva nebo alespoň kompatibilita s již existujícím symbolem pro paralelní režim, který ale zcela chybí. Řídit běh procesu čistě z hlavního procesu také není ideální.

21. Comment - Norma ISO 5807-85 žádný podobný symbol neobsahuje, pro komentování používá symbol pro anotaci, (Sekce 2.3.16), případně přerušovanou čáru (Sekce 2.3.13). Tento znak je zvláštní zvláště tím, že jako jediný není černobílý, což ztěžuje jeho tisk. Přidání dalšího zcela odlišného symbolu, když DRAKON obsahuje i jiný symbol pro komentář, jeví se výrazně nadbytečné.

22. Right comment - Symbol je téměř identický symbolu normy ISO 5807-85 pro anotaci (Sekce 2.3.16), pouze tvar spíše než hranaté závorce napojené přerušovanou čarou odpovídá složené závorce napojené nepřerušovanou čarou. Funkce je shodná.

23. Left Comment - Symbol je identický s „Right comment“ napojeným na uzel z druhé strany.

24. Loop arrow - Jedná se o spojnici (Sekce 2.3.11) se šipkou pro určení směru, díky čemuž je možné vracet se ve vývojovém diagramu a vytvářet cykly v jakémkoliv místě, jak je vidět na Obr. 3.6. DRAKON tak kromě běžného cyklu na základě podmínky (na rozdíl od ISO 5807-85, která povoluje cyklus jen v rámci předem stanovených mezí) povoluje i podstatně nepřehlednější cyklus na základě přepínače, a dokonce i cyklus na základě fork nebo cyklus uvnitř zdánlivě uzavřeného for-cyklu, který vede vně. To jednak zhoršuje přehlednost, jednak povoluje potencionálně velmi nebezpečné konstrukce, u kterých není lehce odhadnutelné chování (stejně jako v případě Goto instrukce). Použití „Loop arrow“ jako samostatného symbolu patrně není vhodné.

25. Silhouette arrow - Jedná se o spojnici (Sekce 2.3.11) se šipkou pro určení směru, určenou čistě k propojení všech konců a začátku všech „Silhouette branch“ (Obr. 3.4). Cílem symbolu je jasně ukázat směr pohybu zpět nahoru, ovšem tím také vzhledem k propojení všech „Silhouette branch“ vyvolává dojem, že může přejít do libovolného „Silhouette branch“ (což není pravda, následující „Silhouette branch“ je vybrán hodnotou v symbolu „Address“) a ztrácí se tak přehlednost. Vývojový diagram by byl nejspíš přehledně bez tohoto symbolu, DRAKON totiž, stejně jako ISO 5807-85, umožňuje volání podprogramu na základě jeho názvu a symboly „Headline“ a „Address“ je možné chápat jako variantu názvu podprogramu („Silhouette branch“) a jeho volání.

26. Connector - Identický s ISO 5807-85 symbolem pro spojku (Sekce 2.3.14).

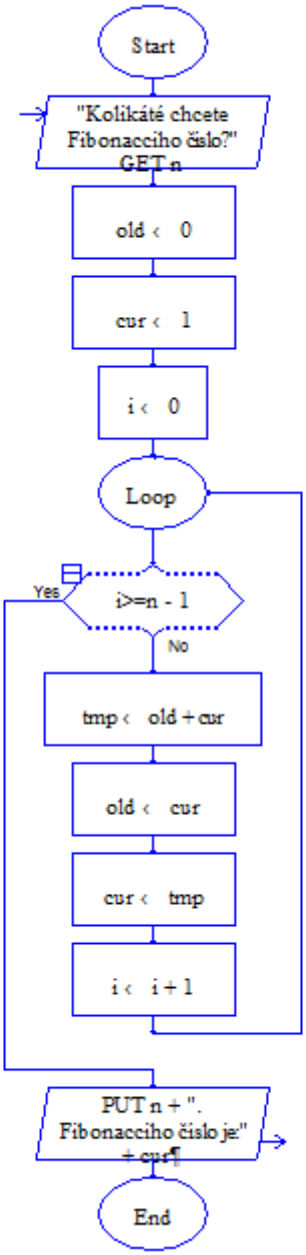
27. Concurrent process - Identický s ISO 5807-85 symbolem pro paralelní režim (Sekce 2.3.9).

3.2.2 Raptor

Raptor [16] je minimalistický vizuální programovací jazyk založený na vývojových diagramech, který se snaží zajistit funkčnost základních operátorů a některých knihoven jako v C a Pascalu. Dle studie [17] má pozitivní vliv na schopnost studentů pochopit látku a naučit se programovat v porovnání s programováním v Ada nebo Matlabu. Raptor obsahuje pouze uzly pro při-

řazení, volání metody, vstup, výstup, podmínku a smyčku, ovšem v rámci přiřazení a volání je možné použít velké množství nadstandartních metod implementovaných přímo v Raptoru, které umožňují i například grafický výstup.

Raptor se evidentně nesnaží splňovat žádnou normu, symbolům z normy ISO 5807-85 odpovídá pouze přiřazení. Symbol pro podmínku (a identicky cyklus) je rozšířený, aby se do něj vešel delší text (viz Obr. 3.8) a navíc umožňuje kolabovat a expandovat větve podmínky (u cyklu vnitřek cyklu), což výrazně napomáhá přehlednosti. Volání podprogramu používá místo symbolu pro předdefinované zpracování symbol pro zpracování s tlustou šipkou vpravo. Symboly pro vstup a výstup jsou upraveny tenkými šipkami ukazující dovnitř a ven. Cyklus má navíc ještě zvláštní symbol pro začátek cyklu, který vizuálně odpovídá spojce.



Obrázek 3.8: Příklad vývojového diagramu v nástroji Raptor. Fibonacciho posloupnost.

Nástroj bohužel není v současné podobě příliš odladěný a výrazně se na něm podepisuje jeho stáří (nejstarší dohledatelná verze 3.5 je z roku 2006). Raptor je sice stále podporován a stále se pro něj objevují nové verze (poslední verze je z roku 2019), přinášejí ale téměř pouze podporu nových

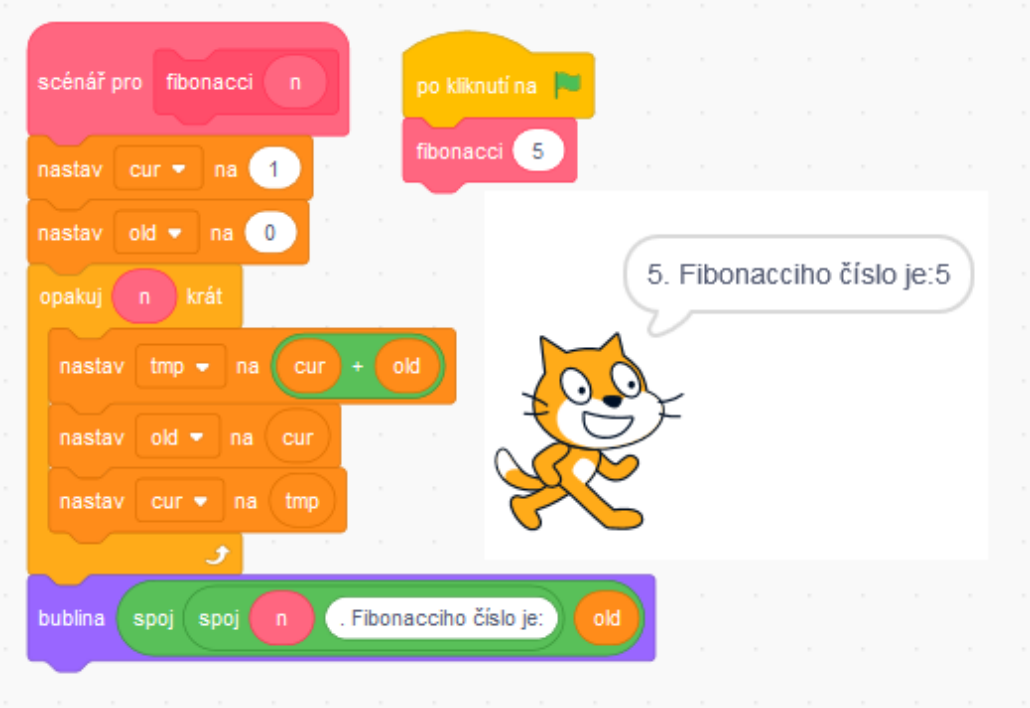
technologií a bug fixy, které ne všechny jsou řešeny zcela korektně. Raptor například od svého spuštění zcela blokuje použití clipboardu mimo sám sebe, což výrazně znepříjemňuje například screenshoting vytvořeného vývojového diagramu v něm.

Klady	Zápory
Nativní metody	Špatný technický stav
Jasně odlišné uzly	Jednobarevné uzly
2D grafika	Nepřívětivé GUI
Kolabovatelné uzly	Chybí statická kontrola

3.2.3 Scratch

Scratch je vizuální programovací jazyk zaměřený hlavně na výuku algoritmizace a základů programování pro děti. Na rozdíl od ostatních zde zmíněných programovacích jazyků a vývojových prostředí vůbec nevyužívá běžné vývojové diagramy a místo nich používá bloky připomínající puzzle. Tyto bloky pasují do sebe předem daným způsobem (viz Obr. 3.9) a jejich elementy lze dle tvaru jasně identifikovat: podmínky jako šestiúhelníky, výrazy a proměnné jako elipsy, výběr z nabídky jako obdélníky a ostatní jako základní bloky, do kterých je možné vkládat dle typu ostatní elementy včetně dalších bloků. Uživateli je tak přímo naznačeno, co by měl použít, a zároveň je pevně omezen, a nemůže tak udělat chybu. Stejné typy bloků jsou navíc označeny stejnou barvou.

V rámci jazyka Scratch je navíc místo jen obyčejné konzole výstupním zařízením vždy obrazovka s minimálně jednou 2D postavou a pozadím, která je prostředníkem prováděných výstupních příkazů. Místo nápisu do konzole tak postava nebo postavy přímo mluví. Postavy je velmi jednoduché animovat a pro uživatele se tak programování lehce stává hrou, respektive vytvářením hry, což je nejspíš jednou z příčin velké popularity Scratche [6] a to nejen u dětí. Jazyk má navíc implementované i události a umožňuje více aktivních vláken. Uživatel díky tomu dokáže vytvořit program, který by jinak se svými schopnostmi vytvořit nedokázal, ovšem cenou za to je kvalita kódu.



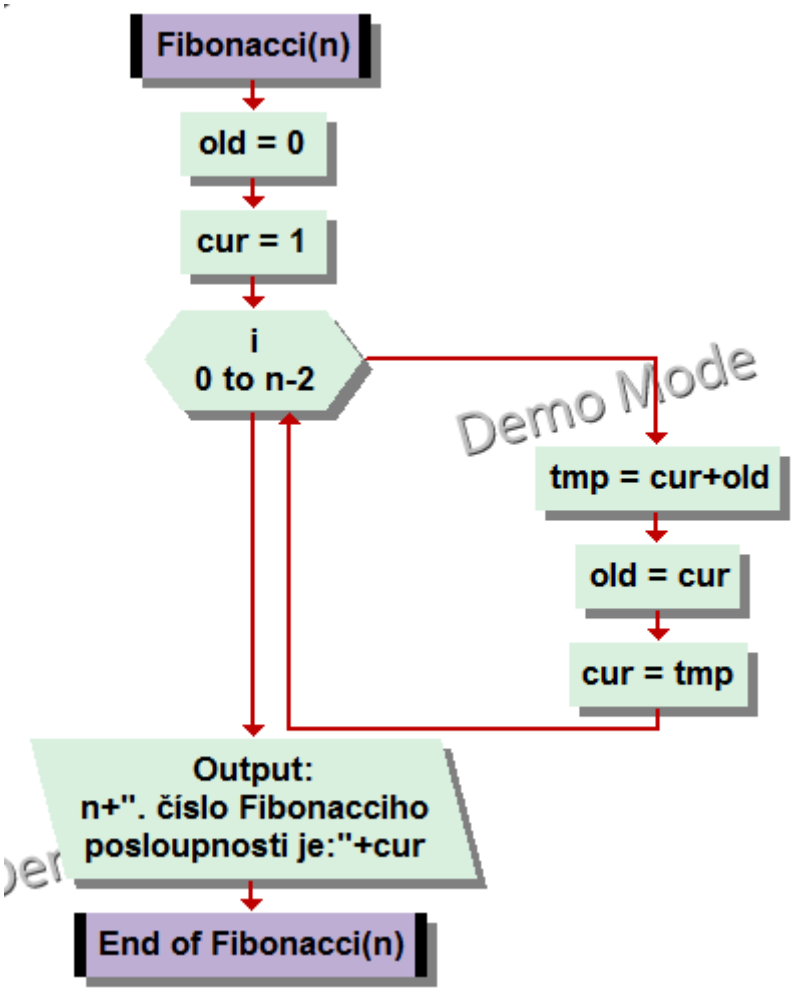
Obrázek 3.9: Příklad programu v jazyce/nástroji Scratch. Fibonacciho posloupnost.

Klady	Zápory
Velmi populární[6]	Pomalé psaní programu
Jasně odlišné uzly	Svazující gramatika
Barevné uzly	Působí infantilně
Přívětivé GUI	
Statická kontrola	
2D grafika	
Více vláken	
Krokování	

3.2.4 Visual Logic

Visual Logic [7] je vizuální programovací jazyk velmi podobný Flowgorithmu funkci i grafickou syntaxí (Obr. 3.10). Hlavní rozdíl je v tom, že Visual Logic je placený (89 dolarů), vypadá vizuálně méně přívětivě a umožňuje navíc kreslení v jednoduché 2D grafice ve stylu takzvané želví grafiky [27].

[6]Díky tomu má také velké množství volně stáhnutelných programů v něm napsaných.

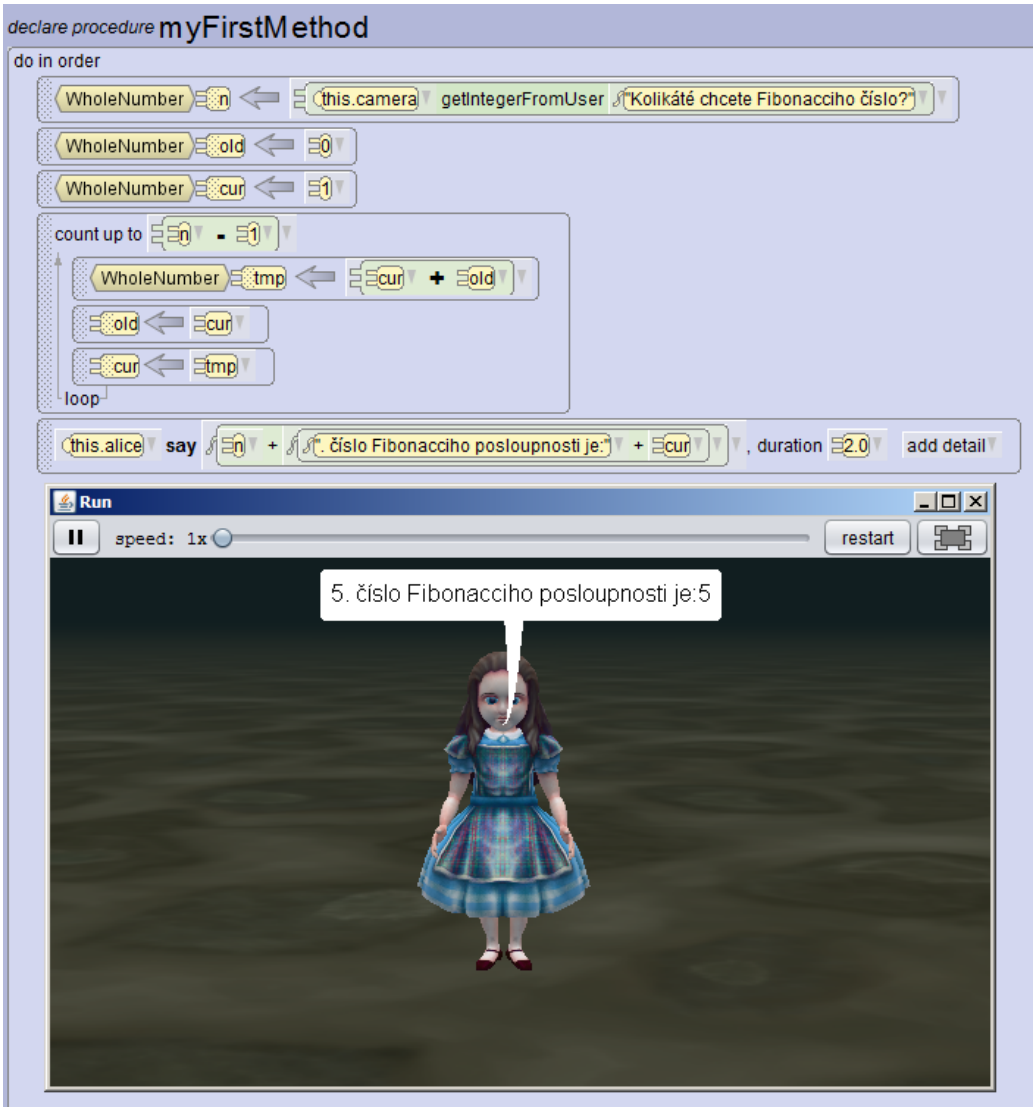


Obrázek 3.10: Příklad programu v jazyce/nástroji Visual Logic. Fibonacciho posloupnost.

Klady	Zápory
Želví grafika	Placené
Jasně odlišné uzly	Vizuálně zastaralé
Barevné uzly	Chybí statická kontrola
Přehledné GUI	

3.2.5 Alice

Alice [40] je komplexní objektový vizuální programovací jazyk s 3D grafickým výstupem velmi podobný Scratchi jak funkčností tak syntaxí. Stejně jako Scratch používá Alice blokovou strukturu kódu, na rozdíl od Scratche je ovšem podstatně méně intuitivní a přehledná, nerozlišuje tak výrazně, kde začínají a končí jednotlivé elementy, a elementy (kromě bloků) není navíc možné přidávat přes *Drag and Drop*, ale musí se vždy složitě vybrat z rozbalovací lišty (Obr. 3.11).



Obrázek 3.11: Příklad programu v jazyce/nástroji Alice. Fibonacciho posloupnost.

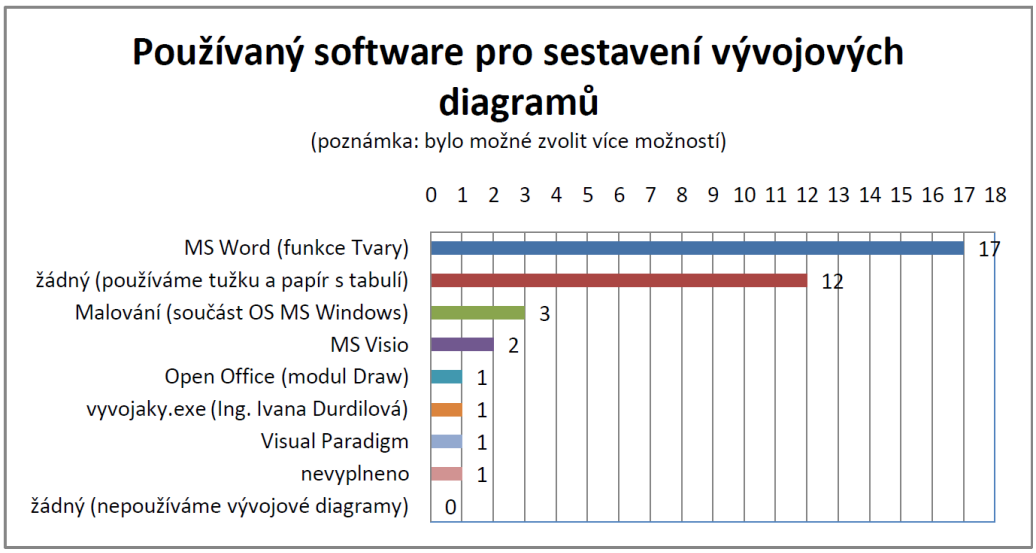
Alice navíc přesně dodržuje principy objektově orientovaného programování a všechny procedury a funkce jsou vlastností nějakého konkrétního objektu, který je vždy instančním atributem samotné scény (přístupná dle javovské syntaxe přes “this“), to nutně pro začátečníky zesložiťuje pochopení fungování algoritmizace, překvapivě ovšem i přesto má Alice velmi pozitivní výsledky z hlediska použití ve výuce, a to zvláště u dívek [26] a mladých žen. Alice má pozitivní výsledky ovšem u obou pohlaví [33] [21].

Alice přitahuje zvláště možností jednoduše vytvářet interaktivní příběhy v třírozměrné grafice a jednoduché počítačové hry, a to s využitím jednoduchých předpřipravených modelů, jejich vizuální stránka se zdá být atraktivní zejména pro ženské publikum, kterému bývá velmi často doporučována[8].

Klady	Zápory
3D grafika	Pomalé psaní programu
Statická kontrola	Svazující gramatika
Objekty	Velmi složité GUI
	Citelně chybí Drag'n'Drop
	Odlišné, ale matoucí uzly

3.3 Použití ve výuce v ČR

Dle průzkumu [13] se ve výuce algoritmizace na středních školách používal v roce 2012 pro vizualizaci algoritmů a vývojových diagramů prakticky pouze Microsoft Word a případně tužka a papír (Obr. 3.12). I přesto už v té době existovalo několik aplikací pro tvorbu a vizualizaci právě vývojových diagramů české výroby. Jedná se o následující:



Obrázek 3.12: Software používaný na středních školách dle [13] pro tvorbu vývojových diagramů v roce 2012.

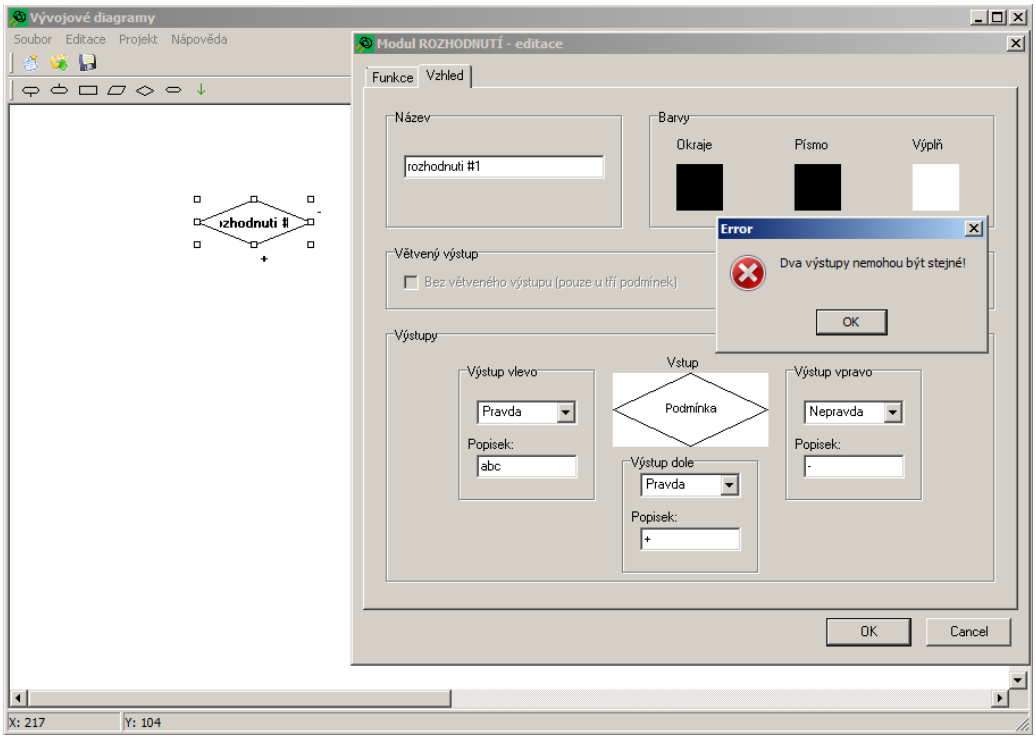
3.3.1 vyvojaky.exe (Durdilová)

Aplikace *vyvojaky.exe* by měla být vytvořená v rámci [24] a dle [13] byla používána ještě v roce 2012 alespoň na jedné české střední škole pro výuku algoritmizace. Dnes již tato aplikace není dohledatelná a stránka, na kterou se odkazuje, neexistuje.

3.3.2 VyvDiag.exe (Snoza)

Aplikace *VyvDiag.exe* byla vytvořena v rámci bakalářské práce [45] a na rozdíl od [24] stále lze dohledat. Aplikace ovšem nebyla žádným způsobem od roku 2009 aktualizována a už při vytvoření nebyla zjevně zamýšlena s UX jako prioritou, priorita byla očividně dána co největšímu množství možností a nastavení pro uživatele, který se v nich ovšem, zvláště pokud jde o začátečníka, rychle ztratí. Například na Obr. 3.13 můžeme vidět, že program umožňuje zcela zbytečně větvení podmínky do tří směrů, pro každý chce ovšem logicky jinou možnost (pravda/nepravda) a proto je skutečně možné vybrat vždy jen 2 směry.

Aplikace má i další mouchy, například program momentálně nedokáže otevřít nápovědu a při každém ukončení se ukončí s chybou. Zcela zásadní je ovšem jeho nepřehlednost pro začátečníka, kvůli které je aplikace pro výuku základů programování ne zcela vhodná.



Obrázek 3.13: Ukázka programu VyvDiag.exe (Snoza). Nastavení symbolu rozhodování (podmínky) umožňuje větvení do tří směrů, možné je ovšem vybrat jen dva.

Klady	Zápory
	Velmi špatný technický stav
	Nepřehledné
	Jednobarevné
	Zastaralé

3.3.3 PS Diagram (Bartyzal)

Aplikace PS Diagram byla vytvořena v rámci bakalářské práce [13] a na rozdíl od svých předchůdců jde o aplikaci stále podporovanou a aktualizovanou. Aplikaci je napsaná v programovacím jazyce Java, pro zajištění kompatibility si udržuje vlastní verzi JRE a obsahuje i automatické aktualizace, bez kterých nepovolí uživateli aplikaci spustit.

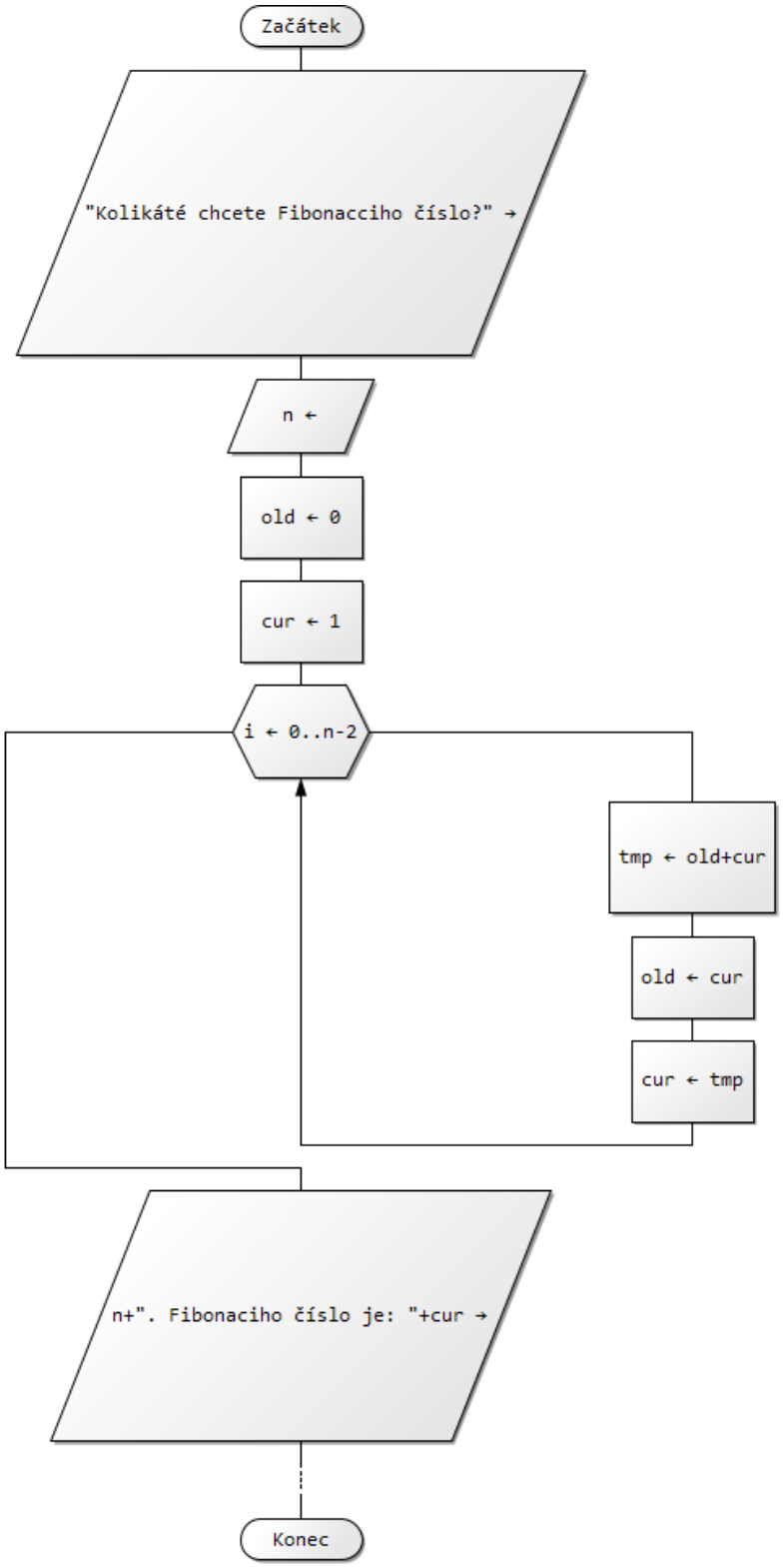
Aplikace má intuitivní ovládání, je přehledná, nezahlcuje uživatele zbytečnými možnostmi a působí graficky přívětivě. Použité symboly odpovídají normě ISO 5807-85 (viz Sekce 2.3) a to bez žádných úprav. Symboly nejsou barevně odlišeny, což mírně zhoršuje orientaci a aplikace i přes dlouhý vývoj vykazuje mírné nedostatky, které se projeví při běžném používání (například

na Obr. 3.14 je patrné, že se velikost symbolů přizpůsobuje svému obsahu. To ovšem nebere v úvahu, že například výstup může obsahovat velké množství textu a symbol tak může nadbytečně zabírat velké množství plochy, tento text navíc není zalamován, plocha symbolu tak roste kvadraticky s množstvím textu.)

Aplikace i přesto umožňuje vytvářet a interpretovat vývojové diagramy s profesionální mírou kvality a lze ji zařadit spolu s Flowgorithmem k mezinárodně nejlepším nástrojům pro tvorbu a interpretaci vývojových diagramů. Během analýzy zadání této diplomové práce byla promýšlena i varianta rozšíření této aplikace, bylo od ní ovšem ustoupeno, neboť v té době byl bytecode aplikace obfuskován a tedy v té době autor zjevně nechtěl, aby jeho aplikaci kdokoliv upravoval[7]. Vyučující Technické informatiky ve strojírenství byli navíc proti vývoji aplikace v jazyce Java a prosazovali vývoj v jazyce C#.

Klady	Zápory
Animované krokování	Mírné technické neduhy
Stále aktualizované	Jednobarevné uzly
Umožňuje Drag'n'Drop	Obtížné rozlišení uzlů
Přehledné GUI	Plýtvání prostorem
	Chybí statická kontrola

[7]Autor v létě rok 2021 změnil názor a otevřel PS Diagram jako open source.



Obrázek 3.14: Ukázka programu PS Diagram. Fibonacciho posloupnost.

3.4 Některé další nástroje pro výuku programování

- Logo [47] - Starý programovací jazyk založený na Lispu používající takzvanou "želví grafiku". Sám o sobě dnes není (používaným) nástrojem pro výuku programování, ale je předchůdcem nebo inspirací velkého množství nástrojů v této kapitole (NetLogo, Scratch, Etoys, Squeak).

- NetLogo [48] - Multiagentní programovatelné prostředí používané pro různé druhy simulací. Poslední verze je z roku 2020, aplikace je k dispozici pod GPL licencí.

- Etoys [25] - Nástroj pro výuku programování založený na objektovém jazyce Smalltalk (varianta Squeak) zaměřený hlavně pro děti. Poslední verze je z roku 2012, aplikace je k dispozici pod kombinací licencí MIT a Apache 2.0.

- Squeak - [10] - Objektový programovací jazyk založený na Smalltalku. První použití je v rámci Etoys. Pro Squeak ale existuje i vývojové prostředí Morphic, které, na rozdíl od Etoys prostředí (Tweak), se snaží o minimální důraz na grafickou stránku nástroje. Poslední verze je z roku 2020 a je stejně jako Etoys k dispozici pod kombinací licencí MIT a Apache 2.0.

- Robot Karel [39] - Programovací jazyk pro výuku algoritmizace spočívající v ovládání pohybu robota na čtverečkovém poli. Stále se objevují nové moderní varianty vývojového prostředí pro něj, poslední je nejspíš Karel-3D z roku 2018. Jazyk je oblíbený zvláště ve střední Evropě.

- Stencyl [19] - Vývojové prostředí pro programování jednoduchých 2D videoher, které jsou konvertovatelné do Flashe a tedy hratelné v prohlížeči (kromě jiného). Poslední verze je z roku 2020. Základní verze je zdarma - samotný engine je dostupný s MIT licencí, pokročilé verze jsou ovšem placené.

- LARP [28] - Jazyk a nástroj pro vytváření a interpretaci vývojových diagramů i pseudokódu - nástroj mezi nimi umí libovolně překládat a umožňuje přímo editovat obojí. Poslední verze je z roku 2008 a aplikace umožňuje vybrat si mezi freeware a shareware licencí.

4 Specifikace interpretu vývojových diagramů

Nová aplikace má složit primárně pro výuku základů programování hlavně v rámci Západočeské univerzity, a to v první řadě v rámci předmětu Technická informatika ve strojírenství, ve druhé řadě v rámci BootCampu a předmětů Fakulty aplikovaných věd a případně také ve výuce kdekoliv jinde.

Technická informatika ve strojírenství je předmět, který složí na Fakultě strojní jako úvod do problematiky programování a algoritmizace a pro většinu studentů Fakulty strojní se jedná o předmět, který jim o programování řekne za celé jejich studium na Fakultě strojní nejvíce. Předmět se zabývá v první řadě hlavně algoritmizací a to na základě vývojových diagramů vytvářených a interpretovaných programem Flowgorithm (viz sekce 3.1), ve druhé řadě pak následuje výukou základů programování v jazyce C#. Vyučovány jsou podmínky, cykly, datové typy, metody a pole - včetně dvourozměrného.

BootCamp FAV je pětidenní intenzivní kurz základů programování pro nové studenty Fakulty aplikovaných věd, zvláště pak ty, kteří nikdy předtím neprogramovali. Algoritmizace je cvičena aktivním řešením problémů (například typu Robot Karel) zpočátku jednak kreslením na papír, jednak programováním ve Flowgorithm. V posledních dnech se pak studenti od Flowgorithmu odprošťují a přechází na programování v běžném programovacím jazyku. Velmi podobně jako v Technické informatice ve strojírenství se procvičují podmínky, cykly, datové typy, metody a pole, ovšem v tomto případně trvá výuka pouze pět dní a záleží výrazně na schopnostech a znalostech studentů, kolik si toho z BootCampu odnesou.

Základy algoritmizace jako takové se momentálně v rámci výuky žádného předmětu na Fakultě aplikovaných věd nevyučují, a ani vývojové diagramy se nepoužívají. Přepokládá se, že studenti si již výukou základů algoritmizace a ideálně také základů programování prošli anebo jsou schopni se to rychle doučit. Zatímco základy programování jsou ale v rámci výuky předmětu Počítače a Programování 1 (PPA1) zopakovány a studenti mají k dispozici online cvičebnici[1] v jazyce Java, algoritmizace jako taková není vyučována a očekává se, že se ji studenti naučí praxí během programování. Je otázkou, jestli by součástí studijních materiálů neměly být také materiály procvičující nejprve algoritmizaci.

[1] https://programovani.kiv.zcu.cz/

Hlavní výhoda výuky algoritmizace a použití vývojových diagramů pro popis algoritmu spočívá v nezávislosti na programovacím jazyce. Studenti tak nemusí řešit zvláštnosti konkrétních programovacích jazyků a nenavyknou si na ně spoléhat, což výrazně zjednodušuje učení nových programovacích jazyků[2]. Lepším řešením tohoto problému je, pokud je studentům ukázán kód v různých programovacích jazycích vygenerovaný dle vývojového diagramu, což například Flowgorithm umožňuje.

4.1 Minimální požadavky

Minimální požadavky odpovídají požadavkům, jejich absence je neslučitelná s použitelností daného nástroje pro tvorbu a interpretaci vývojových diagramů pro použití ve současné výuce. Jejich deklarace je podstatná, neboť velká část nástrojů ukázaná v Sekci 3 alespoň jednu z nich nesplňuje (včetně nástrojů, které se dle [13] v českém školství pro výuku algoritmizace v minulosti používaly). Jedná se o tyto požadavky:

- Program umožňuje vytváření vývojových diagramů, které obsahují deklarace, přiřazení, podmínky, cyklus, vstup z klávesnice a konzolový výstup (vzor Flowgorithm).[3]
- Program umožňuje vytvářené vývojové diagramy ukládat na disk a načítat je.
- Program umožňuje takto vytvořený diagram interpretovat.[4]
- Program umožňuje v rámci vývojového diagramu deklarovat a používat proměnné datových typů integer, real, string, boolean a jejich pole.[5]

[2]Například je velmi časté, že se student naučí v nějakém vyšším programovacím jazyce používat kolekce, aniž by chápal, jak fungují. Pro takového studenta je výrazně obtížnější pracovat v programovacím jazyce, který kolekce ze základu nemá. V porovnání pro studenta, který dokáže pro daný algoritmus vytvořit vývojový diagram, by mělo být triviální tento vývojový diagram přepsat do libovolného (procedurálního) programovacího jazyka.

[3]Většina nástrojů v Sekci 3 například nemá specifický symbol pro deklaraci a přiřazení a používá pro to uzel pro obecný příkaz, jehož interpretace je závislá na gramatice jazyka výrazu. To sice výrazně zjednodušuje práci vývojáře daného nástroje, ale zároveň dělá nástroj výrazně nepřehlednější pro uživatele, který musí být o to zdatnější.

[4]Dle [13] se ještě v roce 2012 ve většině českých středních škol vyučovala algoritmizace kreslením vývojových diagramů na papír nebo v rámci editoru neodporující jeho interpretaci.

[5]Velká část nástrojů zmíněná v Sekci 3 explicitně nedeklaruje datové typy a jejich použití tak buď není zcela jasné, nebo je stejně jako v Javascriptu nepřehledné.

- Datové typy bude možné konvergovat použitím explicitní konverze. [6]

Nástroj by měl též splňovat následující mimofunkční vlastnosti:

- Program bude spustitelný na operačním systému Windows 7[7] a novějším.
- Program by měl minimalizovat možnost, že uživatel cokoliv pokazí/udělá špatně.
- Program by měl minimalizovat možnost, že uživatel nepochopí/špatně pochopí jakoukoliv funkcionalitu aplikace.
- Grafická stránka programu by měla být vizuálně přívětivá.
- Program by mělo být možné dále upravovat.

Většina nástrojů pro tvorbu a interpretaci vývojových diagramů selhává zvláště z hlediska mimofunkčních požadavků, neboť si je ne každý vývojář dokáže uvědomit. V rámci kapitoly 3 je nejlépe naplňují Flowgorithm a PS Diagram, a tak složí jako vhodný vzor pro novou aplikaci. V implementované aplikaci byly všechny tyto funkce plně naplněny.

4.2 Vhodné vlastnosti

Na základě průzkumu vlastností, které mají existující nástroje pro tvorbu a interpretaci vývojových diagramů, a osobních zkušeností s tím, jak studenti programování pracují s Flowgorithmem a obecně i jinými programovacími jazyky, byl vytvořen následující seznam vlastností, které jsou u nástroje pro tvorbu a interpretaci vývojových diagramů prospěšné a buď jsou implementované v nějakém existujícím nástroji anebo se nejeví přehnaně komplexní je implementovat. Jedná se o následující:

- Program umožňuje vytvářet vlastní podprogramy v podobě vývojových diagramů a ty pak volat (třeba i rekurzivně). Parametry podprogramů budou u primitiv předávány hodnotou, u polí referencí.

[6] Konverze mezi datovými typy je ve většině nástrojů v ze Sekce 3 zmatečná a je nekonzistentní v rámci různých datových typů. U některých není zřejmé, jestli je konverze ze základu vůbec podporovaná.

[7] Případně spustitelný po instalaci aplikace nebo podpůrného prostředí jako je .NET

- Program dokáže načíst uložené vývojové diagramy z Flowgorithmu[8]. - Naopak ukládat vývojové diagramy ve formátu Flowgorithmu nemusí být z principu vždy možné, pokud vývojový diagram obsahuje nadstandardní funkce, které Flowgorithm nepodporuje.

- Program umožňuje přeložit v něm vytvořený vývojový diagram do ekvivalentního funkčního kódu (například do jazyka C# nebo Java).[9]

- Diagram půjde inteligentně exportovat do obrázku (ideálně vektorového) a diagram by měl být při tom upraven tak, aby šel přehledně vytisknout na papír.[10]

- Parametry podprogramů bude možné upravit tak, aby bylo možné zvolit, mají-li se předávat hodnotou nebo odkazem.[11]

- Program požaduje uložení vývojového diagramu před jeho interpretací[12].

- Program umožňuje volat podprogramy z jiných tímto programem vytvořených vývojových diagramů, ke kterým má přístup (i v jiných souborech).[13]

- V programu jsou jasně a zřetelně odlišeny symboly pro vstup z klávesnice a výstup do konzole tak, aby se ji student nemohl splést, nebo aby si svoji chybu rychle uvědomil[14].

- Proměnné je možné inicializovat při deklaraci[15].

[8]Požadováno vyučujícími Technické informatiky ve strojírenství. Mají totiž velké množství vývojových diagramů uložených ve formátu Flowgorithmu.

[9]Z nalezených nástrojů má tuto funkcionalitu pouze Flowgorithm.

[10]Toto žádný z nalezených nástrojů nedokáže. „Silhouette branch“ z DRAKON ovšem může být použit k částečnému řešení tohoto problému.

[11]Toto žádný z nalezených nástrojů nedokáže.

[12]Velké množství studentů má tendenci kód neukládat a začínat vždy od začátku, i když by mohli u nového úkolu část starého kódu znovu použít.

[13]Flowgorithm například nedokáže importovat metody z jiných souborů. DRAKON IDE je jediný z testovacích nástrojů, u kterých se tuto možnost podařilo vyzkoušet.

[14]Toto je pro studenty problém dokonce i v případě, že jsou vstup a výstup výrazně odlišený barvou jako je tomu ve Flowgorithmu.

[15]Ač ve vývojových diagramech je zvykem tyto dvě operace odlišovat, v praxi je většina programátorů inicializuje během deklarace. V podobných nástrojích často dochází k deklaraci přiřazením (vzor Javascript), což je zmatečné. Deklarace a přiřazení musí být rozhodně odlišeno. Stačí ovšem, nebude-li možné re-deklarovat proměnné a deklarovat přiřazením - přiřazovat během deklarace ničemu nevadí.

- Program provádí alespoň základní statickou kontrolu syntaxe v rámci výrazů a podmínek a upozorní uživatele, kde je chyba, než uživatel program spustí.[16]
- Program graficky označuje uzly grafu, pro které platí[17]:
  - je v nich chyba
  - jsou prázdné
  - jsou zakomentované
  - je na ně dán breakpoint
  - jsou právě krokovány
- Některé části vývojového diagramu (funkce, cykly, podmínky, nebo libovolné označené regiony) je možné kolabovat a expandovat, aby se zajistila přehlednost grafu.[18]
- Program umožňuje vytváření komentářů vedle uzlů nebo hran grafu.[19]
- Program by mělo být jednoduché přeložit do různých jazyků.[20]
- Program umožňuje souborový vstup a výstup v rámci interpretace.[11]
- Program podporuje dvou nebo více dimenzionální pole.[21]

Všechny tyto vlastnosti, s výjimkou souborového vstupu a výstupu a exportu do obrázku, byly implementovány. O tyto zbývající vlastnosti nebyl během posledních fází vývoje ze strany vyučujících z fakulty strojní velký zájem, a proto na ně nebyly vyhrazeny člověkohodiny (a v případě souborového vstupu a výstupu ani neproběhla analýza).

[16]Kontrolu syntaxe částečně řeší Scratch a Alice tím, že uživateli vůbec nedovolí syntaxi porušit. To ovšem uživatele výrazně svazuje a zpomaluje rychlost psaní programu.

[17]Tuto funkcionalitu zvládají z větší části Flowgorithm a PS Diagram

[18]Tato funkcionalita byla nalezena pouze u nástroje Raptor.

[19]Tato triviální možnost u velké části testovaných nástrojů chybí.

[20]Flowgorithm podporuje velké množství přirozených jazyků, ostatní jsou pouze v jediném jazyce.

[21]Nástroje, které pro interpretaci výrazu používají Javascript, jako DRAKON nebo PS Diagram, tuto možnost umožňují, ostatním chybí.

4.3 Některé další možnosti a rozšíření

Tyto požadavky jsou velmi různorodé a sebrány spíš jako názory různých odborníků k tomu, co by se také dalo s takovým programem dělat, pokud už by byl vytvořen, a to v podobě, která je podobná Flowgorithmu. Jsou zde jen pro dokreslení možností, kterými se vývoj tohoto programu také může v budoucnu ubírat, bez diskuze k tomu, jestli by tyto funkcionality byly výuce programování nějak nápomocné. Většina jich je výrazně časově náročná, a i bez nich tato práce několikanásobně překračuje časovou dotaci předmětu Diplomová práce, nejen proto se s jejich implementací v rámci této diplomové práce rozhodně **nepočítalo**:

- Program by mohl umožnit vytváření a rušení vláken, jejich synchronizaci a prvky pro řešení kritické sekce.[22]
- Program by mohl umožnit vykreslování 2D grafiky, případně i 3D grafiky.[23]
- Program by mohl umožnit volání nativního C#, případně C++ kódu.[24]
- Program by mohl umožnit vytvářet nové datové typy jako struktury.
- Program by mohl podporovat objektově orientované programování.[25]
- Program by mohl implementovat datový typ ratio (tedy číselný zlomek)[26].

[22]DRAKON jako jediný umožňuje limitovanou práci s vlákny, ovšem jeho řešení není zcela ideální. Pokud by měl nástroj sloužit pro výuku paralelizace, musela by být v něm práce s vlákny komplikovanější.

[23]Visual Logic, Scratch a Raptor umožňují vykreslování ve 2D grafice, Alice umožňuje dokonce vykreslování ve 3D grafice. Pro výuku programování toto není nutné, ale jde o funkčnost, která by se mohla hodit, pokud by byl nástroj použit pro úvod do počítačové grafiky.

[24]Volání nativních C# method byla během implementace vyzkoušena a aplikace tímto způsobem používá některé nativní C# metody. Není ale možné pro uživatele volat nové metody. Výrazně by to zesložitovalo testování a pro běžné uživatele stačí množina nejběžnějších nativních metod.

[25]Toto de facto splňuje DRAKON IDE díky použití Javascriptu (vývojový diagram funguje u něj jen jako obálka pro Javascript kód) a Alice. Ani v jednom případě to nepomáhá k výuce programování, a naopak to uživatele zbytečně mate.

[26]Vzhledem k tomu, že se tento datový typ mimo C++ v běžných programovacích jazycích nevyskytuje (a i jako datové struktury je množství jeho použití zanedbatelné v porovnání s čísly s pohyblivou řádovou čárkou), patrně není vhodné, aby s ním studenti začínali, protože k němu v jiných jazycích nebudou mít přístup a potažmo si nezvyknou na problém numerické přesnosti reálných proměnných.

- Program by mohl umožnit automatický překlad sama sebe do libovolného přirozeného jazyka.[27]
- Program by mohl umožnit překlad vývojového diagramu do dalších programovacích jazyků.[28]
- Program by mohl umožnit uzly přesouvat mezi vyhraněnými místy metodou Drag'n'Drop.[29]
- Program by mohl umožnit uživateli jednotlivé uzly grafu libovolně přeskládat a případně se vrátit do základního stavu.[30]
- Program by mohl umožnit uživateli libovolně označit libovolné uzly grafu.[31]
- Program by mohl umožnit uživateli změnit grafický a barevný styl uzlů a hran grafu.[32]

[27]Flowgorithm umožňuje aplikaci používat ve třiceti různých přirozených jazycích.

[28]Flowgorithm umožňuje překlad do dvaceti čtyř programovacích jazyků.

[29]PS Diagram umožňuje dokonce pouze Drag'n'Drop a to i pro přidání nového uzlu (což je časově zdlouhavější). Použití Drag'n'Drop pro přesun existujícího uzlu dává smysl a na absenci této možnosti si studenti někdy stěžují v rámci používání Flowgorithmu, kde musí v takové situaci uzel zkopírovat, vložit na nové místo a smazat jej na starém místě.

[30]Na rozdíl od pouhých editorů vývojových diagramů, žádný z testovaných nástrojů pro interpretaci vývojových diagramů neumožňuje uzly volně posouvat mimo vyhraněné pozice (uzly nebo bloky). Na rozdíl od přesunu existujícího uzlu na jiné místo, tato funkcionalita má velmi redukovanou prospěšnost.

[31]Tato možnost není ani v běžných programovacích jazycích, může ovšem pomoci v orientaci.

[32]Flowgorithm a DRAKON IDE oba umožňují změnit barevný styl uzlů. Flowgorithm navíc umožňuje změnit i tvar uzlů. Většina uživatelů ovšem používá základní styl a změní ho spíše jen omylem. Více různých stylů navíc výrazně znesnadňuje komunikaci uživatele s jinými uživateli, protože stejný diagram u nich vypadá jinak.

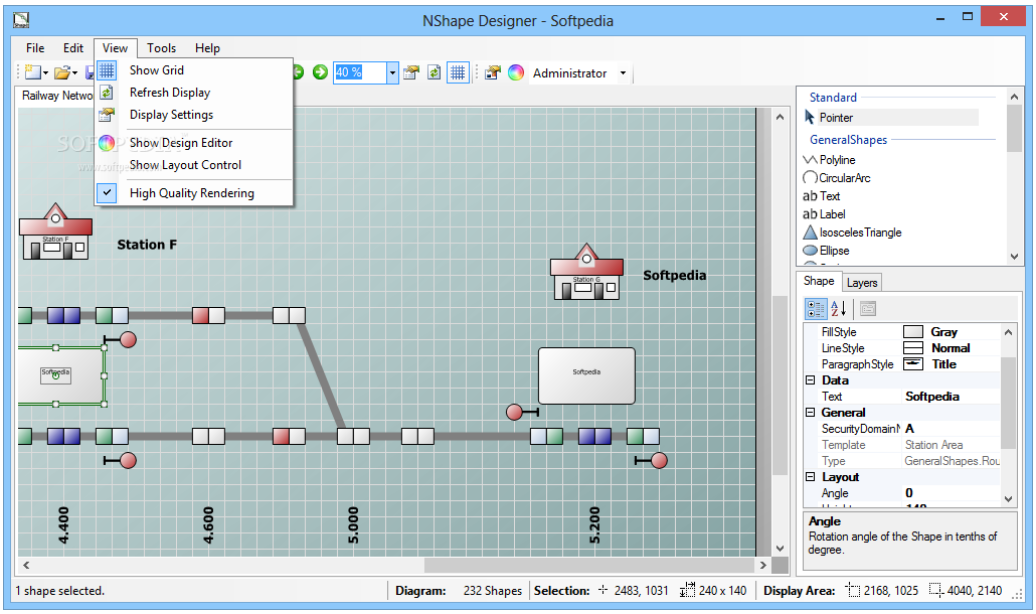
5 Nástroje pro vizualizaci vývojových diagramů a grafů v C#

Na základě přání vyučujících Technické informatiky ve strojírenství bylo rozhodnuto, že vývoj nové aplikace bude probíhat v programovacím jazyce C# a ideálně ve WPF, u kterého se jakožto novější technologie očekávají lepší vlastnosti. Vzhledem k tomu, že vývoj vizualizátoru vývojových diagramů (tedy nástroje pro vykreslování vývojových diagramů a jeho rozložení) by byl časově náročný, že je takový vizualizátor jen jednou z částí v rámci této práce zamýšleného nástroje a že se vizualizace vývojových diagramů jeví jako relativně často řešená operace, která by tím pádem měla mít někým již zhotovené řešení, bylo hned ze začátku vývoje rozhodnuto, že nebude vyvíjen nový vizualizátor, ale bude použit nějaký již existující jako externí knihovna.

Toto rozhodnutí se později neukázalo jako optimální, neboť externí knihovny vizualizující vývojové diagramy, které by bylo možné použít a jsou specificky pro C# nejsou zdaleka tak početné a spolehlivé, jak se očekávalo. Konkrétně byly nalezeny následující nástroje:

5.1 NShape

NShape [4] je grafická C# knihovna pro kreslení různých typů diagramů (Obr. 5.1) - vývojových, sekvenčních, schémat zapojení apod. Hlavní účel knihovny je umožnit uživateli prohlížet, komentovat, upravovat a vytvářet tyto diagramy a ne obecně jen diagramy vykreslovat, jedná se navíc o poměrně složitou aplikaci určenou primárně pro použití ve velkých společnostech. Poskytuje možnosti vytvářet a používat šablony pro generování uzlů, uzly slepovat, agregovat, undo/redo, zooming a různé vrstvy grafu a seskupení, přičemž dokáže efektivně zobrazit tisíce uzlů. Navíc nabízí podporu pro uložení do SQL a XML pro lepší uložení do repozitáře.



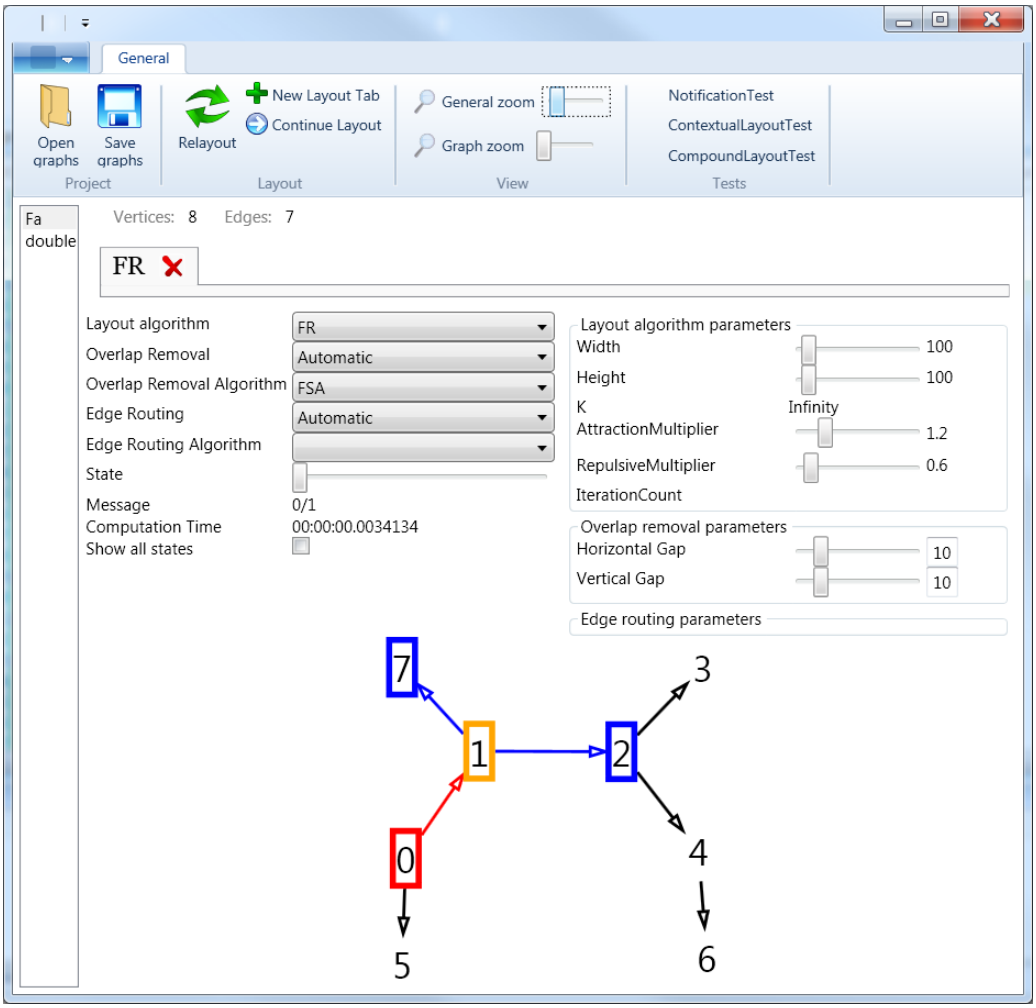
Obrázek 5.1: Ukázka diagramu vytvořeného v knihovně NShape v rámci frameworku NShape Designer.

Poslední nalezená oficiální verze NShape je z roku 2013, na Githubu je ovšem možné najít poslední stable build z roku 2017. NShape je poskytován v rámci open source (GNU GPL v3) licence a je tak možné si dle potřeby upravit kód. Knihovna je postavena nad .NET Windows Forms a nezdá se, že by ze základu povolila vytváření vlastních tvarů uzlu.

NShape se nejeví jako vhodná knihovna pro potřeby nástroje pro tvorbu a interpretaci vývojových diagramů. Nabízí příliš mnoho komplexních funkcí, které ale nejsou relevantní vzhledem k vyvíjenému nástroji, nelze v něm jednoduše vytvořit chybějící tvary uzlů a je postavená jen nad .NET Windows Forms. Navíc není zcela jasné, jestli by bylo jednoduše možné použít jenom relevantní část funkcí této knihovny.

5.2 Graph#

Graph# [34] je grafická C# knihovna pro různá rozložení uzlů grafu a vykreslení grafu. Specializuje se zjevně na velké grafy a hledání závislostí v nich. Uzly nemají žádné tvary (Obr. 5.2) a nezdá se, že by se počítalo s jejich implementací. Je nejisté, co knihovna vlastně přesně umí, protože jí zcela chybí jakákoliv dokumentace kromě výčtu podporovaných rozložení grafu.



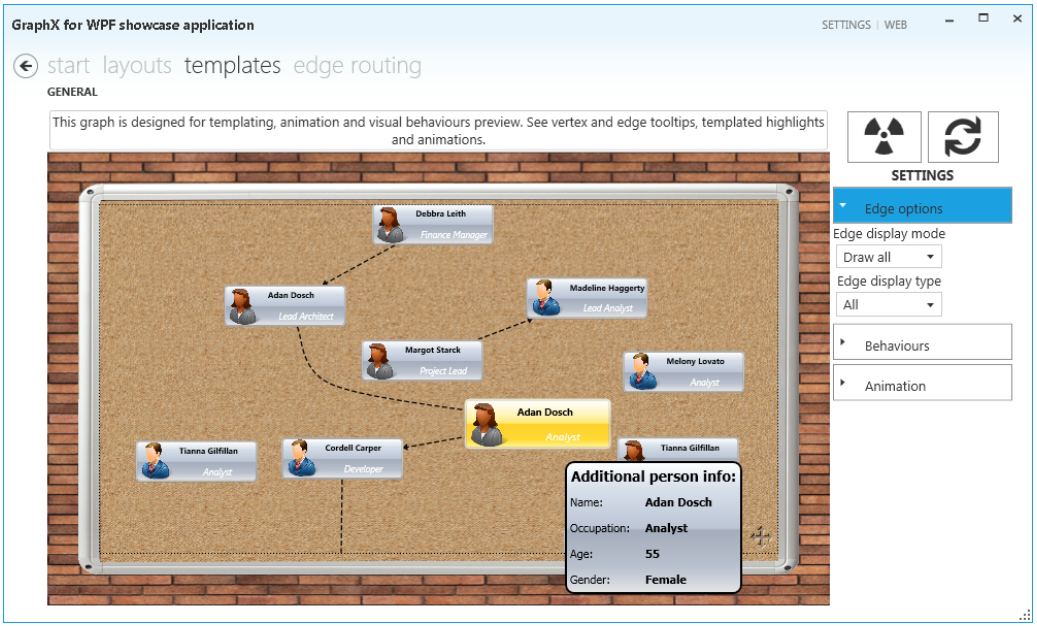
Obrázek 5.2: Ukázka diagramu vytvořeného v knihovně Graph#.

Knihovna je pro WPF a pro svou funkčnost požaduje komponenty *Code Contracts for .NET* a *quickgraph*. Licence aplikace není přesně určena, ovšem zdrojové kódy jsou k dispozici na GitHubu, lze tedy předpokládat, že jde o nějakou open source licenci. Tvůrce jde dohledat jen podle uživatelského jména na GitHubu, knihovna ovšem byla vyvinuta už v roce 2011 a od té doby vývoj není aktualizován. Není proto jisté, jestli je stále funkční.

Tato knihovna není vhodná pro použití hlavně kvůli dlouhé době bez aktualizací a absence možnosti nastavit tvar uzlů grafu.

5.3 GraphX for .NET

GraphX for .NET [43] je grafická C# knihovna založená na knihovně Graph#. Na rozdíl od Graph# ale povoluje velmi flexibilní a komplexní úpravy uzlů grafu, jak je vidět například ona Obr. 5.3. Základní verze GraphX je založená na *Apache Spark API* a obsahuje velmi bohatou a přesnou dokumentaci, bohužel ale dokumentace neodpovídá nadstavbě pro C# - GraphX for .NET - její syntaxe je jiná a výrazně nepřehledná.



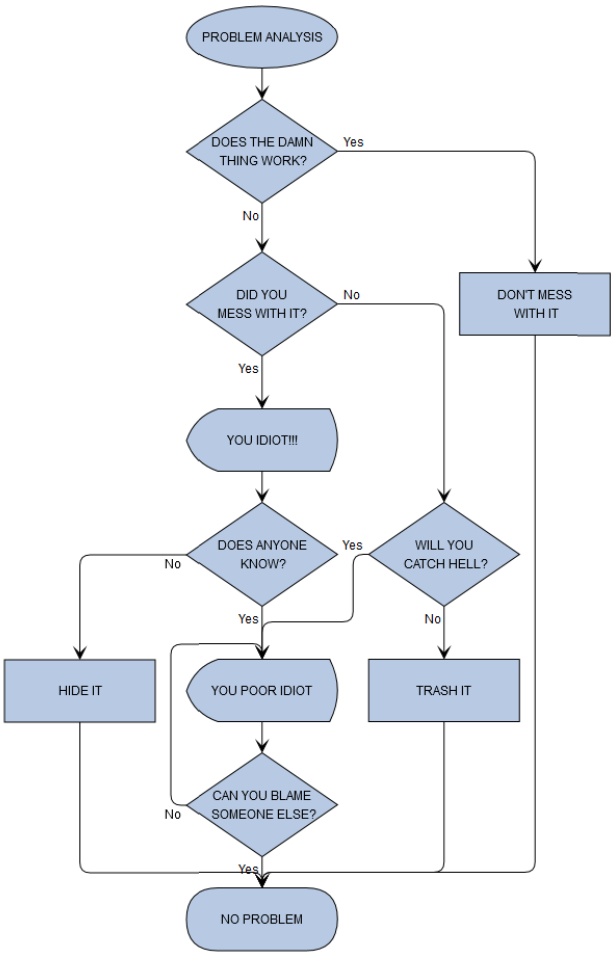
Obrázek 5.3: Ukázka diagramu vytvořeného v knihovně GraphX.

GraphX for .NET by měla podporovat WPF, Windows Forms, METRO, a dokonce i UWA. Základní verze je k dispozici v open source licenci *Apache-2.0 License*, má ovšem také placenou PRO verzi, která byla ještě v roce 2020 k dispozici za nejméně 3000 dolarů na serveru *http://panthernet.ru/* - ten již neexistuje. Na stejném webu také bylo k dispozici fórum, které ovšem bylo jen v ruštině, a odkaz na dokumentaci, který vedl pouze na reklamy. Na GitHubu je možné dohledat verzi updatovanou naposledy v dubnu 2020 s velmi komplexními a nepřehlednými příklady, obtížně se ovšem podle nich zjišťuje, jak knihovna vlastně funguje.

GraphX for .NET by nejspíš šlo použít jako grafovou knihovnu nástroje pro tvorbu a interpretaci vývojových diagramů, zmatečný přístup autora této knihovny k dokumentaci a její stav ovšem nepůsobí nejpřívětivějším dojmem.

5.4 yFiles

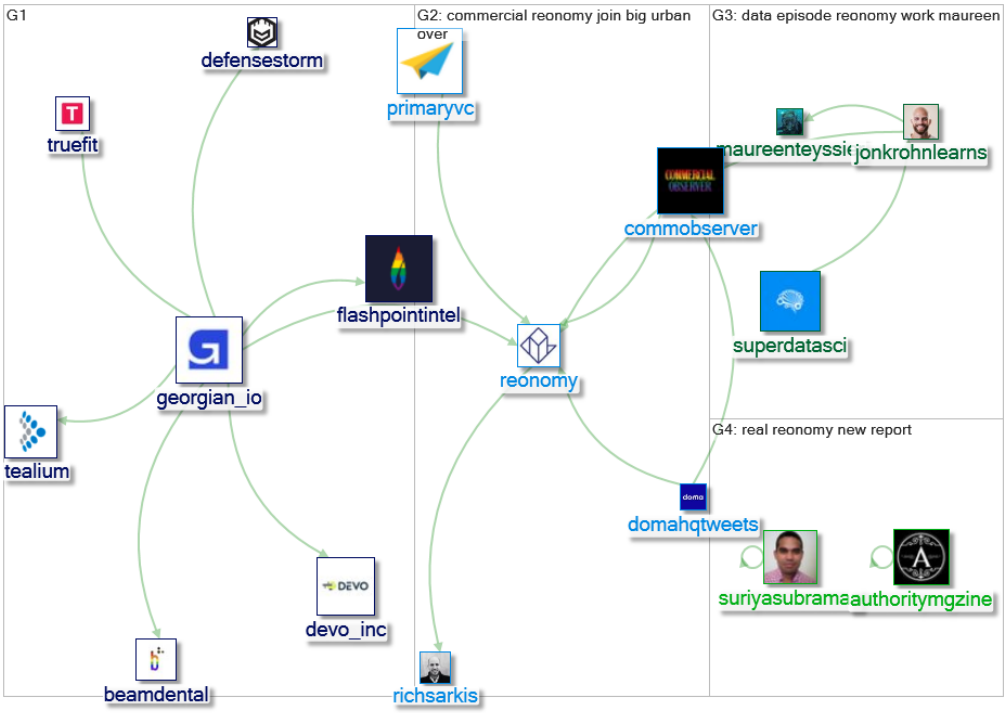
yFiles [9] je nástroj pro tvorbu různých typů grafů (včetně vývojových, viz Obr. 5.4) podporující několik různých programovacích jazyků, včetně C# a WPF v něm. Má bohatou dokumentaci a je aktivně vyvíjena, bohužel ale zdarma poskytuje jenom možnost vyzkoušení na 60 dní a pak je k dispozici pouze v placené verzi. Nejlevnější zvolitelná verze bez podpory stojí 10 000 dolarů, což je výrazně přes hranici rozpočtu tohoto projektu.



Obrázek 5.4: Ukázka vývojového diagramu vytvořeného v yFiles.

5.5 NodeXL

NodeXL [5] je addon pro Microsoft Excel, vytvořený ve WPF. Přidává možnost vytvářet síťové grafy na základě dat a to včetně customizace uzlu (například obrázkem jako je na Obr. 5.5). Zdrojový kód byl ještě v roce 2020 přístupný v open source licenci a bylo možné si jej upravit i pro jiné použití, dnes již se ho nedaří dohledat. Úpravy kódu by ovšem byly nejspíš obtížné a nejedná se proto o ideální řešení.



Obrázek 5.5: Ukázka grafu vytvořeného v NodeXL z dat ze sociální sítě Twitter.

5.6 Satsuma

Satsuma [46] je minimalistická grafová knihovna pro .NET s velmi jednoduchým a příjemným ovládáním. Je k dispozici v rámci licence *zlib*, která umožňuje její plné použití pro komerční i nekomerční účely. Zásadní problém této knihovny je to, že jde o knihovnu v první řadě grafovou a její

grafické funkce spočívají pouze ve vykreslení grafu v nejjednodušší možné formě. Nelze tedy jednoduše změnit tvary uzlů, ani rozložení grafu. Z tohoto důvodu tato knihovna není pro účely nástroje pro tvorbu a interpretaci vývojových diagramů vhodná.

5.7 ZedGraph

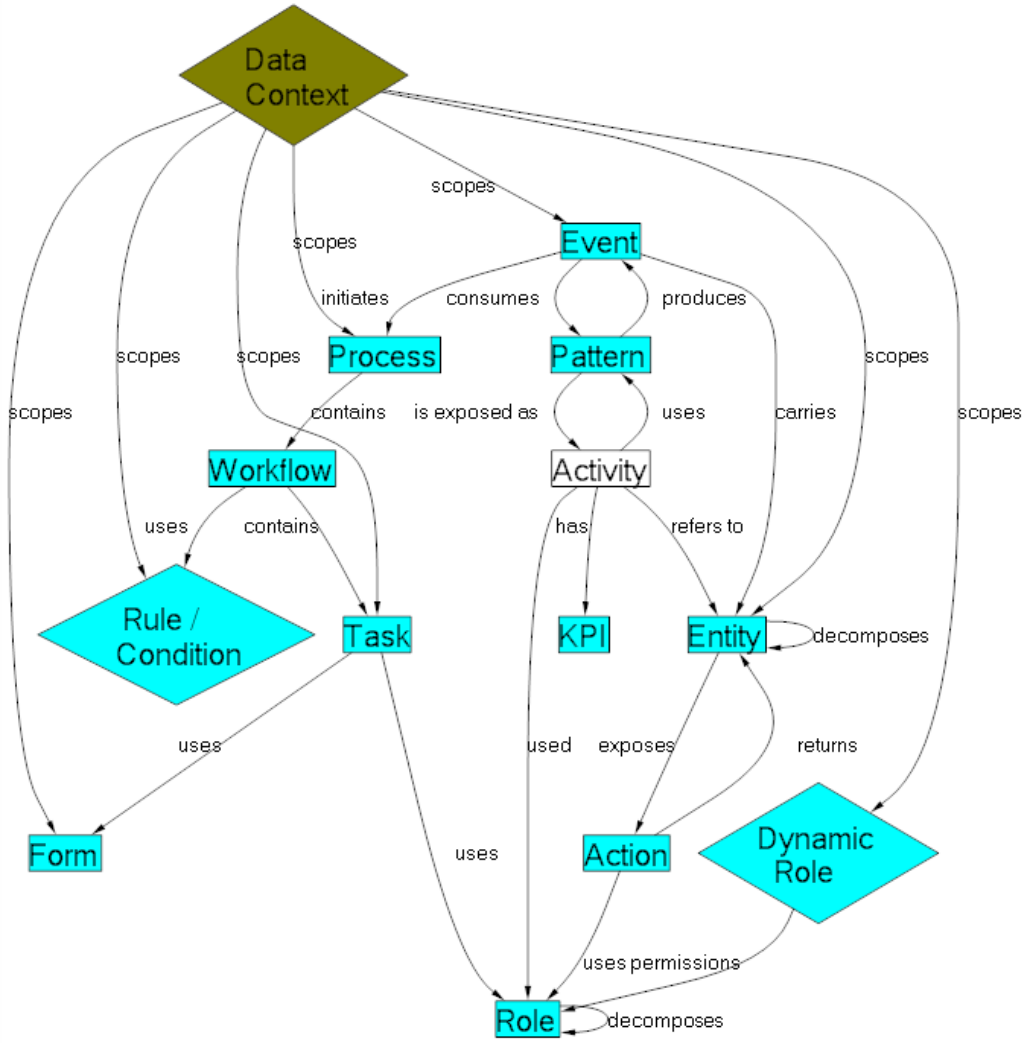
ZedGraph [18] je grafická knihovna pro .NET, která umožňuje vykreslování grafů - jde ovšem o diagramy (charts) nikoliv o grafy. Zdrojové kódy jsou k dispozici ve formě open source, nebyly ale od roku 2008 aktualizovány. Tato knihovna je z těchto důvodů pro použití zcela nevhodná.

5.8 MSAGL

Microsoft Automatic Graph Layout (MSAGL) je grafická grafová knihovna pro Windows Forms, WPF a Javascript s velmi jednoduchým systémem tvorby grafu. Vyvíjí se od roku 2007 v původním názvu GLEE (Graph Layout Execution Engine) a na GitHubu je stále updatovaný, i když poslední update na NuGetu je z roku 2018. MSAGL je dostupný jako open source pod MIT licencí. Umožňuje několik implicitních tvarů uzlu a hran, možnosti vytvoření dalších tvarů, změnu barev (včetně vložení obrázku), změnu hranice uzlů a hran a různá nastavení rozložení grafu. Ke grafu umožňuje zoomovat, funkce undo a redo a dokonce umožňuje vytvářet podgrafy a ty dle potřeby kolabovat a expandovat. Příklad grafu vytvořeného v MSAGL je vidět na Obr. 5.6.

K MSAGL neexistuje regulérní dokumentace, na GitHubu je ovšem knihovna uveřejněna s velkým množstvím jednoduchých příkladů, které vysvětlují její funkčnost. Kód knihovny z nich vypadá na první pohled jasně, jednoduše, a hlavně lehce upravitelný pro případné další rozšíření.

V rámci analýzy provedené na začátku vývoje se u knihovny MSAGL nenašel žádný jasný problém, který by ji znevýhodňoval před ostatními z hlediska použití pro implementaci nástroje pro tvorbu a interpretaci vývojových diagramů, a naopak se pro zadaný účel jevila jako zdaleka nejlepší volba.



Obrázek 5.6: Ukázka grafu vytvořeného v MSAGL.

5.9 Shrnutí výběru

Na základě průzkumu použitelných grafických grafových knihoven pro C# se ukázala MSAGL jako jediná knihovna vyhovovující ve všech kategoriích (shrnuto na Obr. 5.7). Důležité také bylo, že byla ze zkoumaných jedinou knihovnou, která byla přímo určená právě pro vizualizaci grafu vytvořeného přímo v programu bez žádného uživatelem ovládaného editoru. Vzhledem k tomu bylo očekáváno, že bude nutné udělat do kódu této knihovny pouze minimální zásahy v porovnání s ostatními, které buď slouží jako uživatelské editory, nebo slouží pro vizualizaci specifického typu grafů.

Jméno	Kód	Update	WPF	Dokumentace	Účel	Cena
NShape	✓	2017	×	✓	Editace grafu	Zdarma
Graph#	✓	2011	✓	×	Závislosti v grafu	Zdarma
GraphX for .NET	×	2020	✓	×	Vizualizace grafu	Zdarma
GraphX for .NET PRO	×	2020	✓	?	Vizualizace grafu	3 000 dolarů
yFiles	×	2021	✓	✓	Editace grafu	10 000 dolarů
NodeXL	×	2021	✓	×	Generuje grafy v Excelu	39 dolarů
Satsuma	✓	2017	✓	~	Práce s teoret. grafem	Zdarma
ZedGraph	✓	2008	~	×	Vytváření charts	Zdarma
MSAGL	✓	2021	✓	~	Vizualizace grafu	Zdarma

Obrázek 5.7: Tabulka souhrnně porovnávající různé grafické knihovny v C#.

Během implementace se ovšem ukázalo, že MSAGL má celou řadu drobných, ale obtížně řešitelných problémů, které samy o sobě jsou pouze nepříjemné, ale společně zamezují v použití MSAGL očekávaným způsobem. MSAGL například nemá žádný mechanizmus, jak si pamatovat pozici a úroveň přiblížení a bohužel dokonce ani způsob, jak je programově nastavit. To by nebyl výrazný problém, pokud by při každém vytvoření grafu nebyl tento graf vycentrován s úrovní přiblížení obsahující všechny uzly a zároveň nebyl základní mechanizmus pro updatování grafu jeho opětovné vytvoření od základu.

MSAGL navíc nenabízí žádný spolehlivý způsob, jak graf programově upravit na prezenční vrstvě. Některé příklady obsahují tuto funkcionalitu, ovšem pro jiný typ grafu (nižší úroveň) a při použití této funkcionality se vizuálně změní graf jen při některých způsobech interakce, což není žádným způsobem zdokumentováno.

Úpravu MSAGL také komplikuje fakt, že se snaží striktně dodržovat třívrstvou strukturu do takové míry, že aby bylo možné tyto vrstvy používat na sobě zcela nezávisle, i přesto, že na sebe mají tyto vrstvy funkční závislost. Není tak možné na prezenční vrstvě měnit rozložení grafu, ke kterému dochází pouze ve střední grafové vrstvě. Na druhou stranu není také možné z grafové vrstvy explicitně aktualizovat po změně rozložení prezenční vrstvu (kromě opětovného vytvoření grafu na prezenční vrstvě, což vede ke ztrátě informace o pozici a přiblížení). Navíc neexistuje způsob, jak na prezenční vrstvě přistupovat k jednotlivým objektům prezenční vrstvy a jsou uloženy jen v některých případech jako objekty typu object v rámci grafové vrstvy.

Zvláště problematické je také použití podgrafů, které v rámci MSAGL nejsou samy považovány za uzly a jejich prohledávání je nutné implementovat zvlášť. MSAGL má navíc problém už se třetí úrovní zanoření podgrafů, a i po opakovaných pokusech o opravu následné podgrafy umísťuje na grafové vrstvě buď do hlavního grafu nebo se je na grafové vrstvě vůbec nedaří dohledat.

5.10 Vlastní vizualizátor

Snaha opravit problémy MSAGL vedla vzhledem ke komplexitě knihovny k vytváření kaskády nových bugů, a i tak se dařilo problémy pouze zmírnit nebo obcházet. Oprava MSAGL se ukázala jako podstatně náročnější než vytvoření nového vizualizátoru přímo pro potřeby nové aplikace, proto bylo rozhodnuto ušetřit čas tím, že se knihovna MSAGL odstraní a nahradí vlastní implementací.

Při analýze a implementaci nového vizualizátoru se vycházelo jednak z povětšinou negativních zkušeností získaných z práce s MSAGL, jednak ze snahy unifikovat grafovou strukturu na logické i prezenční úrovni. Zároveň byla snaha použít již existující součásti technologie WPF a v případě potřeby je rozšířit. Vizualizátor by implementován odděleně od samotné GUI, kde je pouze prezentován na *Canvas* jako množina grafických objektů typu *Shape*, i od samotného logického grafu, jehož vlastnosti pouze vizualizuje. Vykreslování samotných grafických elementů (uzlů, hran, podgrafů a podpůrných prvků) není plně odděleno od výpočtu jejich rozložení, to zrychluje vývoj, i případné úpravy za cenu mírně horší přehlednosti kódu. Více k tomu vysvětluje Sekce 6.4

6 Implementace

6.1 Popis funkcí

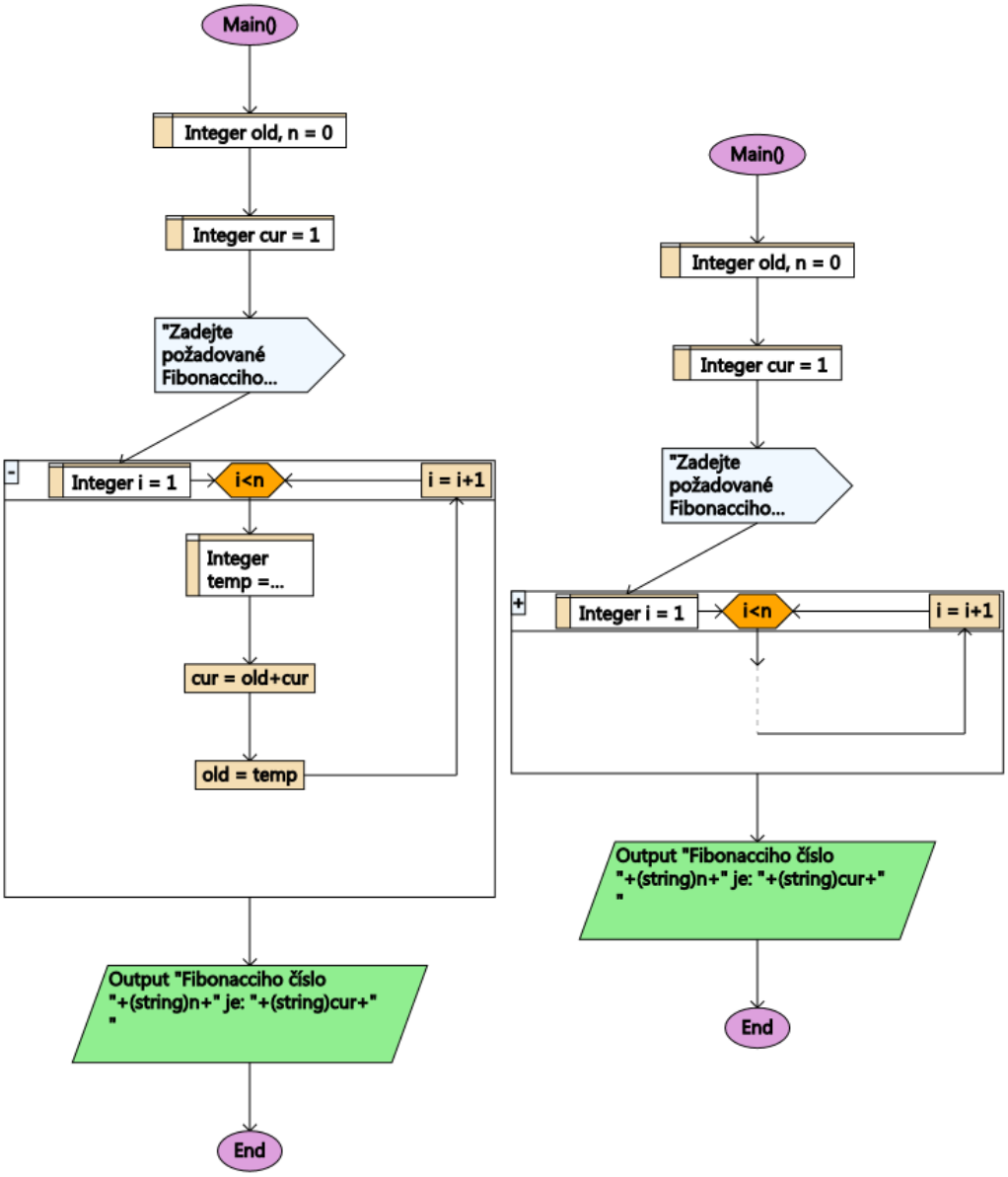
Nová aplikace si bere za vzor Flowgorithm (sekce 3.1). Přidávat nové uzly do vývojového diagramu bude umožňovat kliknutím na příslušnou hranu, kam ji uživatel chce přidat, a editovat uzel umožní dvojklikem na něj. Pro zachování jednoduchosti a minimalizaci možnosti, že si uživatel sám způsobí obtíže v přehlednosti grafu, nebude mít uživatel možnost měnit rozložení grafu dáno mimo posouvání celým grafem a přidávání/odstranění jednotlivých uzlů.

Mezi typy uzlů budou všechny používané Flowgorithmem, tedy deklarace, přiřazení, volání podprogramu, podmínka, cykly do, while a for, komentář a konzolový vstup a výstup. K tomu navíc poskytne uzly upravující postup v grafu - break, continue a return. Všechny uzly bude možné kopírovat a mazat kliknutím na uzel pravým tlačítkem myši - stejným způsobem je bude možné zakomentovat a dát na něj breakpoint. Cykly a podmínky budou fungovat jako podgrafy, které bude možné kolabovat a expandovat pro potřebu přehlednosti (Obr. 6.1).

Uzly, které obsahují výraz - tedy přiřazení, volání podprogramu, podmínky, cykly, výstup apod. - budou mít tento výraz na místě kontrolován na správnost syntaxe a při interpretaci vyhodnotitelný, uživatel tak bude na případnou chybu syntaxe upozorněn už při psaní výrazu. Výraz bude odpovídat redukovanému výrazu v jazyce C# a bude tedy umožňovat použití literálů, proměnných, základních matematicko-logických operací, volání podprogramů (včetně integrovaných metod z knihovny Math a některých dalších) a bude vyžadovat explicitní konverzi datových typů ve stylu cast (datový typ v závorce před proměnnou) nebo použitím nativní metody. Výstupem výrazu bude vždy nepojmenovaná proměnná.

Aplikace poskytuje datové typy integer, real, char, boolean, string a jejich n-rozměrná pole. Bitová velikost hodnot těchto proměnných je 64 bitů pro integer (long), 64 bitů pro real (double), 16 bitů pro char (char), 8 bitů pro boolean (bool). Velikost polí se momentálně určuje při deklaraci, a to množinou literálů představujících velikosti jednotlivých dimenzí pole.

Přesný popis způsobu užívání aplikace je vysvětlen v následující sekci.



Obrázek 6.1: Ukázka kolabování podgrafu cyklu v nové aplikaci.

6.2 Případy použití

Aplikace Flowcha-n představuje vývojové prostředí a jako takové má jeden velký účel, kterým je sloužit ke psaní funkčních programů, konkrétně v případě Flowcha-n v podobě vývojových diagramů. Příklady způsobu užití lze rozdělit na následující části: Editace těla metody, Správa metod, Uložení a načtení programu a Spuštění programu. S aplikací pracuje jen jediná lidská role, kterou je uživatel, respektive programátor.

6.2.1 Editace těla metod

Editace těla metod probíhá přímo v hlavním menu a může tedy probíhat právě jen tehdy, když se tam uživatel nachází. Skládá se z Přidání uzlu, Smazání uzlu, Zkopírování uzlu a Editace uzlu. Tyto akce jsou ekvivalentem editování těla metody v rámci běžného programovacího jazyka.

Přidání uzlu

1. Uživatel zvolí levým tlačítkem myši hranu ve vývojovém diagramu, na kterou chce přidat nový uzel.
2. Vybraná hrana je zvýrazněna.
3. Aplikace dá uživateli nabídku možných typů uzlu (*NodeChooser*).
4. Uživatel zvolí typ uzlu, který chce přidat.
5. Vývojový diagram je aktualizován. Uživatel vidí v diagramu nový uzel, jehož typ odpovídá vybranému typu, na místě hrany, který zvolil. Zvýraznění vybrané hrany je zrušené. Do nového uzlu vede hrana z uzlu, odkud vedla původní vybraná hrana, stejně tak z nového uzlu vede hrana, která vede do místa, kam vedla původní vybraná hrana.

Omezení:

- Hrany, která si uživatel může zvolit, jsou interaktivně zvýrazněny.
- Uzel lze přidat pouze na existující hranu mezi uzly.
- Nelze přidat uzel na hranu mezi dvěma řídícími uzly, jako je podmínka a spojnice v hlavičce běžného cyklu (viz Sekce 6.4) nebo podmínka, inicializace a iterace ve for-cyklu. Toto místo není pro uživatele zvýrazněno.

Smazání uzlu

1. Uživatel zvolí pravým tlačítkem myši uzel, který chce smazat.
2. Aplikace dá uživateli nabídku možných operací nad tímto uzlem.
3. Uživatel vybere smazání uzlu.
4. Vývojový diagram je aktualizován. Vybraný uzel a hrany s ním v přímém kontaktu ve vývojovém diagramu chybí a jsou nahrazeny novou hranou, která zajišťuje celistvost grafu.

Omezení:

- Nelze smazat řídící uzly. Tedy, nelze smazat Terminály, kolabovací tlačítka, spojnice, výpustky, ani uzly pro inicializaci a iteraci ve For-cyklu. Pokud uzel podporuje nabídku operací, operace smazání je zobrazena jako nepovolená a nelze ji vybrat.

Zkopírování uzlu

1. Uživatel zvolí pravým tlačítkem myši uzel, který chce zkopírovat.
2. Aplikace dá uživateli nabídku možných operací nad tímto uzlem.
3. Uživatel vybere zkopírování.
4. Aplikace si zapamatuje, že tento uzel je zkopírován po dobu spuštění aplikace.
5. Uživatel zvolí pravým tlačítkem myši hranu ve vývojovém diagramu, na kterou chce uzel zkopírovat.
6. Aplikace dá uživateli nabídku možných operací nad touto hranou.
7. Uživatel vybere vložení zkopírovaného uzlu.
8. Vývojový diagram je aktualizován. Uživatel vidí v diagramu nový uzel, jehož vlastnosti odpovídají zkopírovanému uzlu v době, kdy byl zkopírován. Do nového uzlu vede hrana z uzlu, odkud vedla původní vybraná hrana, stejně tak z nového uzlu vede hrana, která vede do místa, kam vedla původní vybraná hrana.

Alternativa 1:

- Pokud uživatel už někdy předtím uzel zkopíroval pomocí nabídky operací nad uzlem, může začátek procesu přeskočit a rovnou přejít na výběr hrany, na kterou chce uzel zkopírovat.

Alternativa 2:

- Uživatel může místo přes nabídku operací nad hranou (pravé tlačítko) vybrat vložení uzlu také v nabídce možných typů uzlu (*NodeChooser* – levé tlačítko). Výsledek je identický. V nabídce je navíc ukázaná podoba zkopírovaného uzlu.

Omezení:

- Hrany, na která si může uživatel uzel vložit, jsou interaktivně zvýrazněny.
- Uzel lze vložit pouze na existující hranu mezi uzly.
- Nelze vložit uzel na hranu mezi dvěma řídícími uzly, jako je podmínka a spojnice v hlavičce běžného cyklu (viz Sekce 6.4) nebo podmínka, inicializace a iterace ve for-cyklu. Toto místo není pro uživatele zvýrazněno.
- Nelze kopírovat a vkládat uzly, který uživatel nemůže přidat přes nabídku uzlů. Tedy, nelze smazat Terminály, kolabovací tlačítka, spojnice, ani výpustky. Tyto uzly uživateli vůbec nenabídnou nabídku akcí.

Editace uzlu

1. Uživatel zvolí dvojklikem levého tlačítka uzel, který chce editovat.
2. Aplikace dá uživateli nabídku vlastností uzlu. Nabídka odpovídá typu zvoleného uzlu a obsahuje jeho současné vlastnosti.
3. Uživatel změní vlastnosti uzlu dle potřeby. Viz Sekce 6.2.1.
4. Pokud změna některé z vlastností nebyla platná, aplikace uživatele upozorní varovnou zprávou, která vysvětluje, v čem je problém.
5. Uživatel uloží změny.
6. Vývojový diagram je aktualizován. Uživatel vidí, že zeditovaný uzel má nyní vlastnosti, které mu nastavil.

Alternativa 1:

- Pokud uživatel uložil uzel s neplatnými vlastnostmi, uzel je ve vývojovém diagramu označen jako chybný. V tomto stavu zůstane, dokud jej znovu nezedituje. Program je možné spustit a pokud dojde k tomuto chybnému uzlu, pokusí se jej interpretovat stejně jako běžný uzel.

Alternativa 2:

- Uživatel místo uložení změn změny zruší. V tomto případě se uzel navrátí do stavu, ve kterém byl na začátku editace.

Omezení:

- Nelze editovat uzly, který uživatel nemůže přidat přes nabídku typů uzlu. Tedy, nelze editovat Terminály, kolabovací tlačítka, spojnice, ani výpustky. Pro tyto uzly se nabídka vlastností neotevře. V případě specificky Terminálu se otevře místo nabídky vlastností uzlu nabídka vlastností metody. O tom viz Sekce 6.2.2.

Změna vlastností uzlu

Ke všem změnám vlastností uzlu dochází v nabídce vlastností uzlu, kam je možné se dostat před dvojklik levým tlačítkem na uzel (viz Sekce 6.2.1). Typ uzlu omezuje, které vlastnosti uzel má a případně i některá další omezení této vlastnosti. V případě, že vlastnost není platná pro daný typ uzlu, uživateli se v nabídce vůbec nezobrazí. Pokud uživatel vyplní vlastnost neplatnou hodnotou, je na to upozorněn varovnou zprávou specifikující daný problém.

- Datový typ – Výběr z nabídky datových typů. Představuje datový typ proměnné v daném uzlu. Vždy má hodnotu a každá z možností v nabídce je validní. Vlastnost je platná pro uzly typu Deklarace a Parametr.

- Dimenze – Kladná přirozená čísla včetně nuly oddělená čárkou. Představuje seznam velikostí pole ve všech jeho dimenzích. Prázdná hodnota znamená, že datový typ není polem. Vlastnost je platná pro uzly typu Deklarace a Parametr.

- Do-smyčka – Zaškrtávací políčko (pravda/nepravda). Představuje možnost, jestli je daný cyklus typu „do“ - tedy jestli vykonává první běh těla cyklu před podmínkou. Vlastnost je platná pro uzly typu Loop.

- Inicializace – Výběr z nabídky uzlů (Přiřazení, Deklarace a Void – tedy žádný uzel). Představuje typ uzlu, který je použit při inicializace tohoto uzlu For. Vlastnosti inicializačního uzlu je možné editovat zvlášť, buď otevřením jeho nabídky vlastností přímo z této nabídky, nebo dvojklikem na existující uzel z vývojového diagramu. Vlastnost je platná pro uzly typu For.

- Iterace – Výběr z nabídky uzlů (Přiřazení, Deklarace a Void – tedy žádný uzel). Představuje typ uzlu, který je interpretován na konci každé iterace tohoto uzlu For. Vlastnosti iteračního uzlu je možné editovat zvlášť, buď otevřením jeho nabídky vlastností přímo z této nabídky, nebo dvojklikem na existující uzel z vývojového diagramu. Vlastnost je platná pro uzly typu For.

- Typ skoku – Výběr z nabídky typů skoku (Break, Continue, Return). Představuje typ skoku, tedy instrukce měnící průběh interpretace kódu. Viz Sekce 6.3.5. Vlastnost je platná pro uzel typu Skok.

- Nová řádka - Zaškrtávací políčko (pravda/nepravda). Představuje výběr, jestli má dojít po výpisu k odřádkování. Vlastnost je platná pro uzly typu Vstup a Výstup.

- Reference - Zaškrtávací políčko (pravda/nepravda). Představuje výběr, jestli daný parametr je referenční. Vlastnost je platná pro uzel typu Parametr.

- Text – Textový řetězec představují výraz interpretovaný parserem – musí jej tedy být možné interpretovat. V případě Podmínky, Smyčky a uzlu For jde o podmínku, v případě Přiřazení o přiřazovanou hodnotu, v případě Deklarace o základní hodnotu, v případě Vstupu a Výstupu o prompt (text vypsaný do konzole), v případě Skoku typu Return o návratovou hodnotu a v případě Komentáře a o jeho obsah. V případě Deklarace, Skoku a Vstupu je povolena prázdná hodnota. V případě komentáře je vypnuta kontrola parsovatelnosti. Vlastnost je platná pro všechny typy uzlů s výjimkou Parametru a Skoku typu Break a Continue.

- Until-smyčka - Zaškrtávací políčko (pravda/nepravda). Představuje možnost, jestli je daný cyklus typu „until“ - tedy jestli místo pokračování smyčky, dokud platí podmínka, nemá smyčka naopak pokračovat do té doby, dokud podmínka platit nezačne. Vlastnost je platná pro uzly typu Loop.

- Proměnná – Textový řetězec představující názvy proměnných, kterých se daná operace týká, oddělené čárkou. Hodnota je nejprve rozdělena na jednotlivé proměnné a ty jsou následně zkoušeny na parsovatelnost. Vlastnost je platný pro uzly typu Přiřazení, Deklarace, Parametr a Vstup. V případě Parametru ovšem není možné vypsat najednou více názvů proměnných než jednu.

6.2.2 Správa metod

Správa metod probíhá v nabídce správy metod, do které je možné se dostat přes položku *Method Management* ve hlavním menu. Skládá se z Otevření metody, Přidání nové metody, Smazání metody, Zkopírování metody a Editace hlavičky metody. Tyto akce jsou ekvivalentem stejnojmenných operací v běžném programovacím jazyce.

Otevření metody

1. Uživatel vybere metodu v seznamu metod kliknutím.
2. Uživatel vybere možnost otevření metody.
3. Správa metod uloží současný stav a zavře se.
4. Pokud ve správě metod došlo k nějaké úpravě, tyto úpravy jsou uloženy.
5. Aplikace připne vybranou metodu pod hlavní menu a otevře její tělo jako vývojový diagram. Ten je následně možné editovat (viz Sekce 6.2.1).

Alternativa 1:

- Uživatel vybere metodu v seznamu metod dvojklikem. V tom případě se přeskočí nutnost vybrat možnost otevření metody.

Alternativa 2:

- Pokud metoda je již připnutá pod hlavní menu, není třeba otevírat správu metod. Stačí ji otevřít kliknutím na její název.

Omezení:

- Nelze otevřít nativní metody. Ty mají při výběru tuto možnost zobrazenou jako zakázanou a nelze ji vybrat.

Editace hlavičky metody

1. Uživatel vybere možnost editovat metodu.
2. Aplikace otevře uživateli nabídku vlastností metody (*MethodEditor*). Těmito vlastnostmi jsou název metody, datový typ návratové hodnoty a seznam parametrů.
3. Uživatel změní vlastnosti metody dle potřeby:

(a) Uživatel změní textový řetězec názvu metody na libovolný neprázdný řetězec bez bílých znaků.

(b) Uživatel vybere z nabídky datových typů datový typ návratové hodnoty.

(c) Uživatel změní velikost dimenze návratové hodnoty.

(d) Uživatel změní množinu parametrů pomocí těchto úprav:

   i. Editace parametru
      A. Uživatel vybere parametr, který chce editovat.
      B. Uživatel klikne na tlačítko pro editaci parametru.
      C. Aplikace dá uživateli nabídku vlastností parametru. Parametr se považuje za typ uzlu. Jeho editace je identická s editací uzlu, viz Sekce 6.2.1.
      D. V případě uložení vlastností parametru je stav parametru v nabídce vlastností metody upraven, v opačném se vrací do předchozího stavu.
   ii. Přidání parametru
      A. Uživatel vybere místo v seznamu parametrů, kam chce parametr přidat.
      B. Uživatel klikne na tlačítko pro přidání parametru.
      C. Aplikace dá uživateli nabídku vlastností parametru, který chce přidat. Parametr se považuje za typ uzlu. Jeho editace je identická s editací uzlu, viz Sekce 6.2.1.
      D. Pokud uživatel zruší editaci parametru, parametr není přidán.
      E. Pokud uživatel uloží vlastnosti parametru, parametr je přidán do nabídky vlastností metody.
   iii. Smazání parametru
      A. Uživatel vybere parametr, který chce smazat.
      B. Uživatel klikne na tlačítko pro smazání parametru.
      C. Parametr je odstraněn ze seznamu.
   iv. Posun parametru nahoru
      A. Uživatel vybere parametr, který chce posunout nahoru.
      B. Uživatel klikne na tlačítko pro posunutí nahoru.
      C. Parametr si vymění pozici s parametrem, který je umístěný hned nad ním.

v. Posun parametru dolů
   A. Uživatel vybere parametr, který chce posunout dolů.
   B. Uživatel klikne na tlačítko pro posunutí dolů.
   C. Parametr si vymění pozici s parametrem, který je umístěný hned pod ním.

vi. Kopírování parametru
   A. Uživatel vybere parametr, který chce zkopírovat.
   B. Uživatel klikne na tlačítko pro kopírování parametru.
   C. Aplikace parametr zkopíruje a pamatuje si ho ve stavu, v jakém byl zkopírován, po dobu otevření této nabídky vlastností metody. Tlačítko pro vložení parametru změní vizuálně svůj stav a je stisknutelné.
   D. Uživatel vybere místo v seznamu parametrů, kam chce kopii parametru přidat.
   E. Uživatel klikne na tlačítko pro vložení parametru.
   F. Aplikace vloží kopii na vybrané místo.
   G. Aplikace upozorní uživatele na duplicitu parametrů.
   H. Uživatel zedituje kopii parametrů dle vlastní potřeby.

4. Pokud změna některé z vlastností nebyla platná, aplikace uživatele upozorní varovnou zprávou, která vysvětluje, v čem je problém.

5. Uživatel uloží změny v nabídce metody.

6. Metoda je ve správě metod aktualizována a v seznamu ji je možné dohledat se změněnými vlastnostmi.

7. Uživatel uloží změny ve správě metod.

8. Správa metod se zavře. Vývojový diagram je aktualizován pro případ, že je změněna v současnosti otevřená metoda.

Alternativa 1:

- Pokud uživatel uložil metodu s neplatnými vlastnostmi, při uložení se zavře nabídka vlastností metod a stav správy metod se vrátí do původního stavu. Není tedy tímto způsobem možné uložit metodu v chybovém stavu.

Alternativa 2:

- Pokud je správu metod v neplatném stavu, například kvůli neplatnému stavu některé z metod nebo jejich duplicitě, pak při uložení správy metod je správa metod běžným způsobem ukončena, ale seznam metod se vrátí do stavu, ve kterém byl před otevřením správy metod. Není tedy tímto způsobem možné uložit seznam metod v chybovém stavu.

Alternativa 3:

- Uživatel může místo uložení vlastností metody změny zrušit. V tom případě změny nejsou uloženy. V případě přidání nové metody to znamená, že metoda není přidána.

Alternativa 4:

- Uživatel může místo uložení správy metody změny zrušit. V tom případě se správa metod zavře. Všechny metody se vrátí do stavu před otevřením správy metod.

Alternativa 5:

- Hlavičku metody je možné také editovat přes dvojklik na uzel typu Terminál. Editace funguje běžným způsobem a v případě uložení se změny rovnou reflektují přímo ve vývojovém diagramu metody.

Omezení:

- Nelze editovat nativní metody. Ty mají při výběru tuto možnost zobrazenou jako zakázanou a nelze ji vybrat.
- Název metody nesmí být prázdný nebo obsahovat bílé znaky. Uživatel je na neplatný název upozorněn varovnou zprávou se specifikací problému.
- Pokud uživatel vybere návratový typ *Void*, metoda je procedurou, a tedy nemá návratovou hodnotu.
- Dimenze návratové hodnoty musí být přirozené kladné číslo nebo nula (v takovém případě se nejedná o pole). Uživatel je na neplatnou hodnotu dimenze návratové hodnoty upozorněn varovnou zprávou se specifikací problému.
- Dimenze návratové hodnoty akceptuje absenci vstupu jako hodnotu nula.

- V seznamu parametrů nesmí být duplicitní parametry. Uživatel je na duplicitu parametrů upozorněn varovnou zprávou se specifikací problému.
- Žádný z parametrů v seznamu nesmí mít prázdný název nebo název obsahující bílé znaky. Uživatel je na neplatný název parametru parametrů upozorněn varovnou zprávou se specifikací problému.
- Není možné posunout nahoru parametr, pokud nad parametrem žádný parametr není. V takovém případě je posunutí nahoru zakázáno, na tlačítko posunutí nahoru nejde kliknout a jeho vizuální stav to reflektuje.
- Není možné posunout dolů parametr, pokud pod parametrem žádný parametr není. V takovém případě je posunutí dolů zakázáno, na tlačítko posunutí dolů nejde kliknout a jeho vizuální stav to reflektuje.

Smazání metody

1. Uživatel vybere metodu v seznamu metod kliknutím.
2. Uživatel vybere možnost smazání metody.
3. Metoda je smazány ze seznamu metod.
4. Uživatel uloží stav správy metod.
5. Správa metod se zavře. Pokud byla smazaná metoda připnuta pod hlavním menu, je z tohoto místa odstraněna.

Omezení:

- Nelze smazat nativní metody. Ty mají při výběru tuto možnost zobrazenou jako zakázanou a nelze ji vybrat.

Zkopírování metody

1. Uživatel vybere metodu v seznamu metod kliknutím.
2. Uživatel vybere možnost zkopírování metody.
3. Správa metod metodu zkopíruje a bude si ji pamatovat po dobu tohoto otevření správce metod.
4. Uživatel klikne na vložení metody.

5. Zkopírovaná metoda ve vložena do seznamu ve stavu, v jakém byla zkopírována.
6. Pokud jsou v seznamu dvě nebo více identických metod, je na to uživatel upozorněn zprávou specifikující tento problém.
7. Uživatel zkopírovanou metodu upraví přes editaci hlavičky metody.
8. Uživatel uloží stav správy metod.

Alternativa 1:

- Pokud uživatel duplicitu neopraví a uloží správu metod, správa metod se zavře bez uložení. Všechny změny, které uživatel ve správě metodu udělal, jsou tím zrušeny.

Omezení:

- Nelze kopírovat a vkládat nativní metody. Ty mají při výběru tuto možnost zobrazenou jako zakázanou a nelze ji vybrat.
- Možnost vkládat je zobrazena jako zakázaná a nelze ji vybrat, pokud není žádná metoda zkopírována.

Přidání nové metody

1. Uživatel vybere možnost přidat metodu.
2. Aplikace otevře uživateli nabídku vlastností metody (*MethodEditor*).
3. Uživatel změní vlastnosti metody dle potřeby viz Sekce 6.2.2.
4. Uživatel uloží změny v nabídce metody.
5. Metoda je ve správě metod aktualizována a v seznamu ji je možné dohledat se změněnými vlastnostmi.
6. Uživatel uloží změny ve správě metod.
7. Správa metod se zavře.
8. Změny jsou uložené a při příštím otevření správa metodu obsahuje přidanou metodu.

Alternativa 1:

- Již existující metodu je možní importovat z jiného vývojového diagramu. Viz Sekce 6.2.3.

Alternativa 2:

- Pokud je správu metod v neplatném stavu, například kvůli neplatnému stavu některé z metod nebo jejich duplicitě, pak při uložení správy metod je správa metod běžným způsobem ukončena, ale seznam metod se vrátí do stavu, ve kterém byl před otevřením správy metod. Není tedy tímto způsobem možné uložit seznam metod v chybovém stavu.

Alternativa 3:

- Uživatel může místo uložení vlastností metody změny zrušit. V tom případě změny nejsou uloženy. V případě přidání nové metody to znamená, že není metoda přidána.

Alternativa 4:

- Uživatel může místo uložení správy metody změny zrušit. V tom případě se správa metod zavře. Všechny metody se vrátí do stavu před otevřením správy metod.

Omezení:

- Platí omezení viz Sekce 6.2.2.

6.2.3 Uložení a načtení

Uložení a načtení probíhá přes položky menu *File* v hlavním menu. Kromě uložení a načtení samotného programu (vývojových diagramů) je tu možné také importovat metody z jiného programu a přeložit program do určeného programovacího jazyka.

Uložení programu

1. Uživatel vybere položku „*Save as...*“.
2. Uživatel vybere místo na disku, kam chce soubor uložit, a název souboru, pod kterým jej chce uložit.
3. Uživatel zvolí formát Flowcha-n programu „.fcha“.
4. Uživatel potvrdí uložení.

5. Na zvoleném místě na disku se vytvoří soubor se zvoleným názvem. Obsahuje současný stav programu a je možné jej načíst jako program. Aplikace na pozitivní výsledek nebo selhání upozorní zprávou.

Alternativa 1:

- Pokud na zvoleném místě soubor se stejným názvem již existuje, je přepsán.

Alternativa 2:

- Pokud uživatel zvolí místo formátu „.fcha“ formát programu Flowgorithm „.fprg“, je nejprve upozorněn na to, že tento formát není plně podporován. Následně se aplikace pokusí uložit současný obsah programu v tomto formátu stejně jako v běžném případě. Vytvořený soubor je možné otevřít jak aplikací Flowcha-n, tak aplikací Flowgorithm.

Alternativa 3:

- Pokud uživatel zvolí místo formátu „.fcha“ formát C# programu „.cs“, je program uložen v podobě běžného C# programu. Flowcha-n nedovede otevřít soubor v tomto formátu.

Načtení programu

1. Uživatel vybere položku „*Open*“.
2. Uživatel zvolí formát souboru, tedy „.fcha“ pro Flowcha-n program nebo „.fprg“ pro Flowgorithm.
3. Uživatel vybere soubor na disku, který chce načíst.
4. Uživatel potvrdí načtení.
5. Aplikace se pokusí načíst soubor ve zvoleném formátu. V případě neúspěchu uživatele upozorní na specifickou chybu zprávou. V opačném případě načte všechny metody z daného souboru a přepíše jimi seznam metod původního programu.
6. Aplikace poté zavře všechny otevřené metody a otevře metodu *Main*.

Omezení:

- Aplikace nedokáže načíst program v jiném formátu než „.fcha“ nebo „.fprg“.

Import metod

1. Uživatel vybere položku „*Import metod*“.
2. Uživatel zvolí formát souboru, tedy „.fcha“ pro Flowcha-n program nebo „.fprg“ pro Flowgorithm.
3. Uživatel vybere soubor na disku, který chce načíst.
4. Uživatel potvrdí načtení.
5. Aplikace se pokusí načíst soubor ve zvoleném formátu. V případě neúspěchu uživatele upozorní na specifickou chybu zprávou. V opačném případě načte všechny metody z daného souboru a přidá je k existujícím metodám současného programu. Metody je možné najít ve správě metod.

Omezení:

- Aplikace nedokáže načíst program v jiném formátu než „.fcha“ nebo „.fprg“.

6.2.4 Spouštění programu

Obsluha spouštění programu probíhá pomocí tlačítek v hlavním menu. Umožňují program spustit, zastavit a debugovat.

Spuštění programu

1. Uživatel klikne na tlačítko pro spuštění „▷“.
2. Aplikace změní informaci o stavu aplikace v pravém horním rohu.
3. Tlačítko pro spuštění se změní na tlačítko pro pozastavení „‖“.
4. Do konzole je vypsáno, že se program spouští a současný čas.
5. Aplikace postupně interpretuje program shora od prvního uzlu grafu (bez předchůdce).
6. Pokud narazí na výstup, vypíše jej do konzole.
7. Pokud narazí na vstup, požádá o něj uživatele:
   (a) Aplikace vypíše do konzole prompt.
   (b) Aplikace změní stav v pravém horním rohu na čekání na vstup.

(c) Uživatel dodá vstup do konzole a potvrdí ho.

(d) Aplikace změní informaci o stavu aplikace na spuštění.

(e) Aplikace načte vstup a pokračuje v interpretaci.

8. Pokud interpret nemá žádný další uzel, nebo došel na uzel Return v hlavní metodě, interpretace končí.

9. Aplikace vypíše do konzole, že interpretace skončila a současný čas.

10. Aplikace změní stav tlačítka pro spuštění a informací o současném stavu.

Alternativa 1:

1. Při chybě programu během interpretace označí interpret uzel, kde došlo k chybě.

2. Aplikace k chybnému uzlu přiscrolluje.

3. Aplikace vypíše uživateli aplikace chybu do konzole.

4. Interpretace je ukončena, včetně změny stavů.

Alternativa 2:

- Pokud uživatel během spuštění programu klikne na tlačítko pro zastavení, program se přepne do debugovacího režimu a zastaví se na následujícím uzlu. Dál pokračuje viz Sekce 6.2.4.

Alternativa 3:

- Pokud uživatel během spuštění programu klikne na tlačítko pro ukončení, interpretace je ukončena, a to včetně změny stavu.

Debugování programu

1. Uživatel vybere Debugovací režim.

2. Uživatel vybere pravým tlačítkem myši uzel, od kterého chce začít debugovat.

3. Aplikace dá uživateli nabídku operací nad uzlem.

4. Uživatel vybere možnost přidání breakpointu.

5. Aplikace aktualizuje vývojový diagram a vizuálně zobrazí breakpoint na vybraném uzlu.

6. Uživatel klikne na tlačítko pro spuštění „▷“.

7. Aplikace spustí interpretaci způsobem, který je popsaný v Sekci 6.2.4.

8. Interpret narazí během interpretace na breakpoint, označí uzel, na kterém je, a pozastaví interpretaci.

9. Aplikace přenastaví stav na v breakpointu, tlačítko pro spuštění získá základní tvar „▷“.

10. Aplikace otevře pozorovatele proměnných, ve kterém je vidět současný stav aplikace.

11. Uživatel krokuje aplikaci kombinací z následujících způsobů:

    - Uživatel klikne na tlačítko *Step into* („↓“). V tom případě pokračuje interpretace a zastaví se hned na následující uzlu. Stav se opět přenastaví na v breakpointu a pozorovatel proměnných je aktualizován.
    - Uživatel klikne na tlačítko *Step over* („↷“). V tom případě pokračuje interpretace a zastaví se hned na následující uzlu, který je na stejné nebo vyšší úrovni zanoření, případně na breakpointu. Stav se opět přenastaví na v breakpointu a pozorovatel proměnných je aktualizován.
    - Uživatel klikne na tlačítko *Step out* („↑“). V tom případě pokračuje interpretace a zastaví se až na dalším uzlu na vyšší úrovni zanoření, případně na breakpointu. Stav se opět přenastaví na v breakpointu a pozorovatel proměnných je aktualizován.
    - Uživatel klikne na tlačítko pro spuštění. V tom případě pokračuje interpretace a zastaví se až na dalším breakpointu. Případně pokračuje do konce viz Alternativa 1. Uživatel za běhu změní vývojový diagram přidáním, smazáním, kopírováním nebo editací uzlů, aby opravil nalezenou chybu.

12. Uživatel dokončí interpretaci buď přímo ukončením nebo tím, že nechá interpret úspěšně doběhnout do konce.

13. Pozorovatel proměnných se po ukončení zavře a stav se změní na neběžící.

Alternativa 1:

- Pokud je program spuštěny v debugovacím režimu, ale interpret nikdy nenarazí na breakpoint, program je interpretován procesem identickým v Sekci 6.2.4.

Alternativa 2:

- Program nemusí být spuštěn v debugovacím režimu a nemusí obsahovat breakpoint. Stačí, aby uživatel za běhu klikl na tlačítko pro pauzu. Dojde k zastavení na nejbližším následujícím uzlu, jako při použití tlačítka pro *Step into* („↓“), a přepnutí do debugovacího režimu. Uživatel může pak pokračovat viz Sekce 6.2.4, ale nemůže si takto přímo vybrat, kde je interpretace zastavena.

Alternativa 3:

- Uživatel nemusí vývojový diagram upravovat za běhu. Může interpretaci ukončit a upravovat jej až poté. Pokud si chce ovšem změny ověřit, musí pak program znovu spustit.

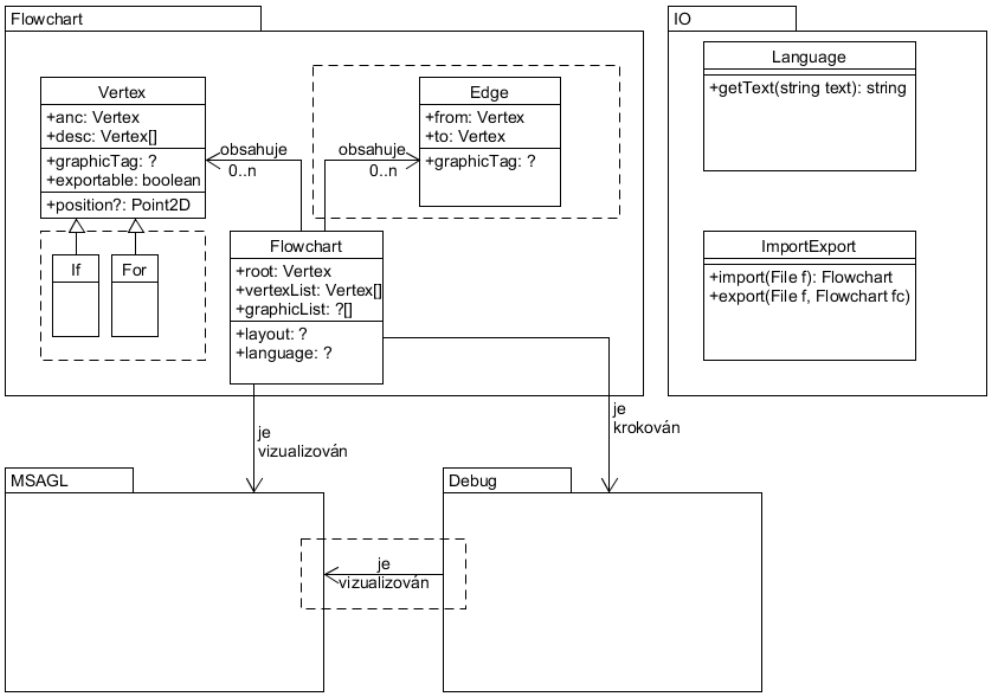
Omezení:

- Pokud je vývojový diagram měněn za běhu, nesmí být měněný uzel, který je právě krokován. Operace nad ním se neprovedou.

6.3 Softwarová architektura

Od počátku vývoje této nové aplikace bylo zamýšleno rozdělit aplikaci na komponenty pro větší přehlednost, modularitu a zjednodušení případných úprav. Rozdělení na Obr. 6.2 vystihuje prvotní myšlenku architektury a obecně struktury této aplikace tak, aby bylo na jediném obrázku pochopitelné, jak se bude vývoj ubírat. Tzn. komponenta na obrázku nazývaná *Flowchart* obsahuje data daného grafu (vývojového diagramu), *MSAGL* pak tyto data vizualizuje a interpret (zde nazývaný *Debug*) umožňuje jejich interpretaci a krokování. Veškeré metody pro jazyky, uložení/načtení a konverze měli být umístěny mimo (komponenta *IO*).

Architektura se během vývoje pochopitelně přizpůsobila novým potřebám a skutečnostem, stejně jako byl uzpůsoben plán vývoje v závislosti na časových možnostech a zjištěné funkčnosti použitých cizích komponent. Přesný stav softwarové architektury aplikace je popsán v podsekcích níže.



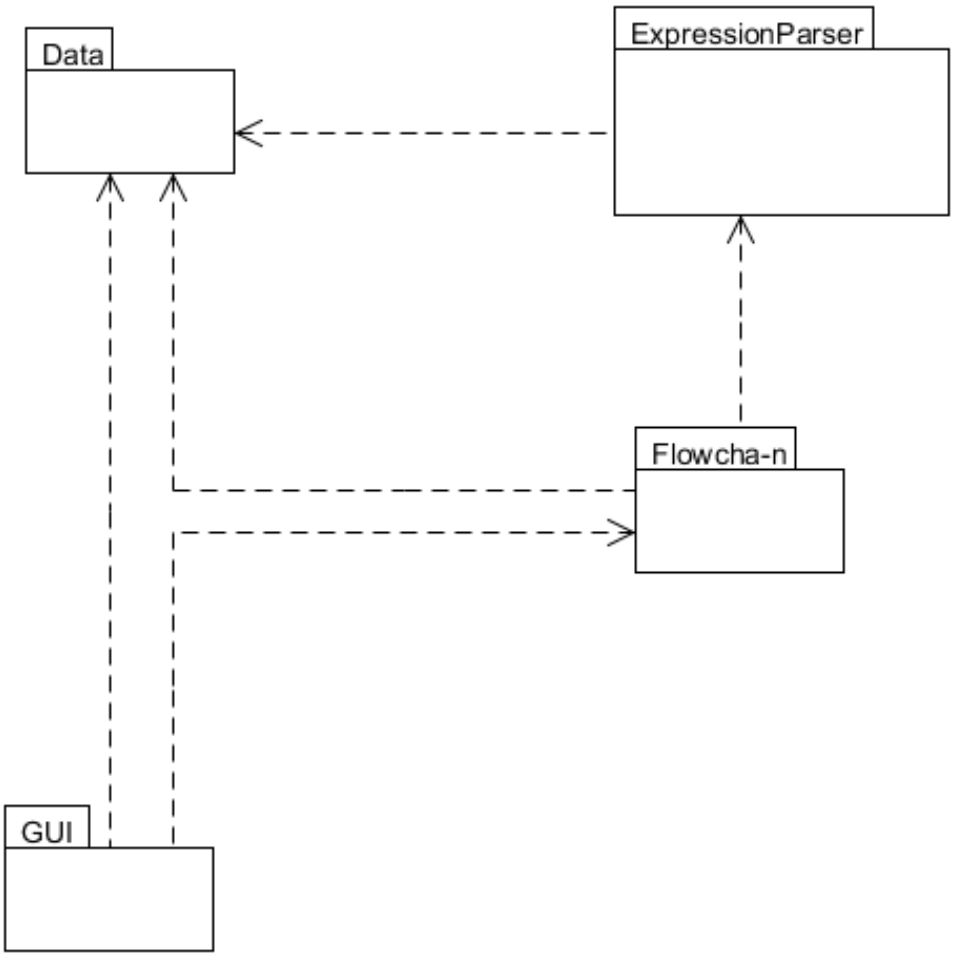
Obrázek 6.2: Doménově-komponentově-třídní model nové aplikace ve stavu, ve kterém byl poprvé navrhnut.

6.3.1 Komponentový model

Skutečnou podobu komponent je možné vidět na grafu závislostí na Obr. 6.3.

Stručně k jednotlivým komponentám:

- Komponenta *Flowcha-n* obsahuje vnitřní strukturu vývojových diagramů (uzlů, grafů a podgrafů) a také jejich interpret a debugger. Jedná se o hlavní jádro aplikace obsahující většinu jeho logiky - proto také sdílí název aplikace. Jde o projekt typu class library.
- Komponenta *Data* byla z komponenty *Flowcha-n* vyčleněna pro zamezení vzniku kruhové závislosti, obsahuje proměnné, jejich datové typy a hodnoty a také operace nad nimi. Také obsahuje pomocné metody a funkcionality, které se používají ve všech částech aplikace, jako je manažer chyb. Jde o projekt typu class library.



Obrázek 6.3: Graf závislostí komponent nově vyvinuté aplikace v podobě package diagramu. Testovací projekty jsou ignorovány.

- Komponenta *ExpressionParser* je parser výrazů postavený nad knihovnou *Antlr*, slouží pro vyhodnocení výrazů v uzlech vývojového diagramu (blíže vysvětleno v sekci 6.3.4). Oddělení této komponenty bylo nutné zvláště proto, že Antlr v C# je nestabilní a někdy u něj nastávají potíže při překladu. Je tedy podstatně jednodušší s ním pracovat, pokud se jedná o nezávislou komponentu, která se testuje zvlášť od zbytku. Jde o projekt typu class library.
- *GUI* obsahuje GUI ve WPF, která slouží k obsluze a vizualizaci celé aplikace. V tomto projektu je také, z větší části odděleně od samotného GUI, implementován vizualizátor vývojových diagramů. Jde o projekt typu Windows Application.

Interpret nebyl oddělen do samostatného projektu[1] od vnitřní reprezentace vývojového diagramu, protože jsou obě tyto komponenty velmi úzce propojené a bylo by obtížné odstranit kruhovou závislost mezi nimi[2]. V aplikaci je i takto nutné řešit část závislostí přes dependency injection a to u pěti metod: *FindVariable* (nalezení proměnné v zásobníku), *AddVariable* (přidání proměnné do zásobníku), *CallMethod* (zavolání metody), *UpdateConsole* (updatování výstupní konzole) a *UpdateCanvas* (updatování prezenční vrstvy vizualizující vývojový diagram). Delegáty na všechny tyto metody jsou umístěny ve třídě *Helper* komponenty *Data*, jejich fyzická implementace je umístěna ve třídě *Interpret* komponenty *Flowcha-n* s výjimkou *UpdateConsole* a *UpdateCanvas*, které jsou přímo v komponentě *GUI* ve třídě *MainWindowViewModel*, která je mezivrstvou GUI (viz Sekce 6.3.3).

6.3.2 Třídní model

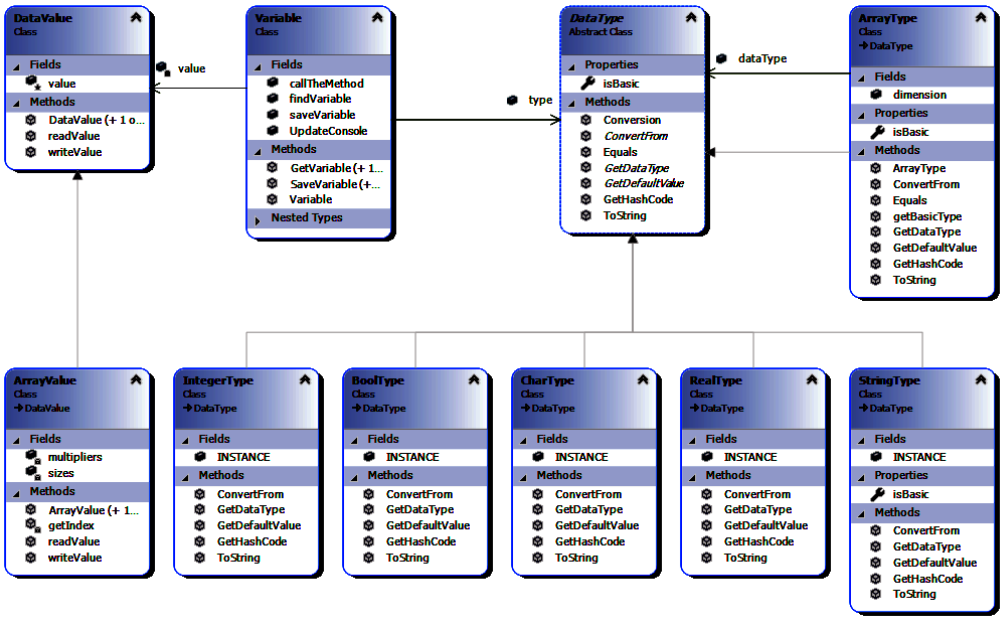
Data

Komponenta *Data* má strukturu odpovídající Obr. 6.4.

- *DataType* je abstraktní třída určující datový typ proměnné. Obsahuje abstraktní metody *ConvertFrom* (konvertuje hodnotu z určeného datového typu na datový typ v this), *GetDataType* (vrací číselník typu) a *GetDefaultValue* (základní hodnota, na kterou je proměnná inicializována), které její subtypy rozšiřují.
- Třídy *IntegerType*, *BoolType*, *CharType*, *RealType* a *StringType* jsou potomky třídy *DataType* zastupující stejnojmenné primitivní datové typy. Jejich hodnoty v rámci C# odpovídají datovým typům long, bool, char, double a string.

[1]Projekt je označení, které se v C# používá pro samostatné moduly, ať už jsou spustitelné nebo jen knihovnami.

[2]Interpretace uzlu je závislá na typu uzlu a je tedy jeho vlastností. Aby mohl interpret uzel interpretovat, musí použít tuto vlastnost, a tedy závisí na uzlech. Interpretace některých uzlů ovšem požaduje vlastnosti specifické pro interpretaci, jako je například přidání rámce do zásobníku při vstupu do podgrafu nebo čekání na uživatelský vstup. Pro odstranění této kruhové závislosti by tak musela být interpretace uzlu oddělena od interpretu, což by buď znamenalo nepřehledné pravidlové řešení, nebo vytvoření množství nadbytečných tříd. To se jeví jako výrazně nadbytečné vzhledem k tomu, že interpret je oproti zbytku projektu drobný.



Obrázek 6.4: Diagram tříd komponenty *Data*.

- *ArrayType* je potomek třídy *DataType* zastupující array. Jeho atribut *dataType* odpovídá datovému typu, který nabývají prvky v poli. Může to být i další pole[3].
- *Variable* je třída představující proměnnou. Obsahuje datový typ *type* (*DataType*), kterého je proměnná a jeho hodnotu *value* (*DataValue*).
- *Helper* je třída pomocných metod, které fungují jako služba ze všech částí aplikace. Obsahuje metody (getter/setter) pro uložení a načtení hodnoty dané proměnné a také delegáty metod, které jsou vkládány ze závislé komponenty přes dependency injection. Třída na obrázku chybí a je naznačena jako součást třídy *Variable*.
- *DataValue* je třída wrapující hodnotu proměnné.
- *ArrayValue* je potomek *DataValue*. Umožňuje načítat a ukládat hodnoty na přesné indexy pole, které je interně reprezentované 1D polem.

[3]Pozor, i v případě vícerozměrného pole jsou všechny hodnoty uchovány v 1D poli. - U hodnoty se nejedná o pole polí.

- *ErrorManager* je třída představující manažér chyb. Kdykoliv dojde během běhu aplikace k chybě, ať už výrazu, interpretace programu, nebo nějaké jiné chyby, chyba je manažerem zaznamenána a později pro uživatele vypsána, nejčastěji do konzole společně s názvem uzlu, kde došlo k chybě.

- Ostatní - Data dále obsahují ve třídě *Utility* metodu *CloneObject* zajišťující reflexní klonování libovolného objektu a metody pro aktivní podmíněné čekání jako ekvivalent *conditional variable* ve třídě *TaskEx*.

Vertex

Třídy typu *Vertex* mají strukturu znázorněnou na Obr. 6.5.

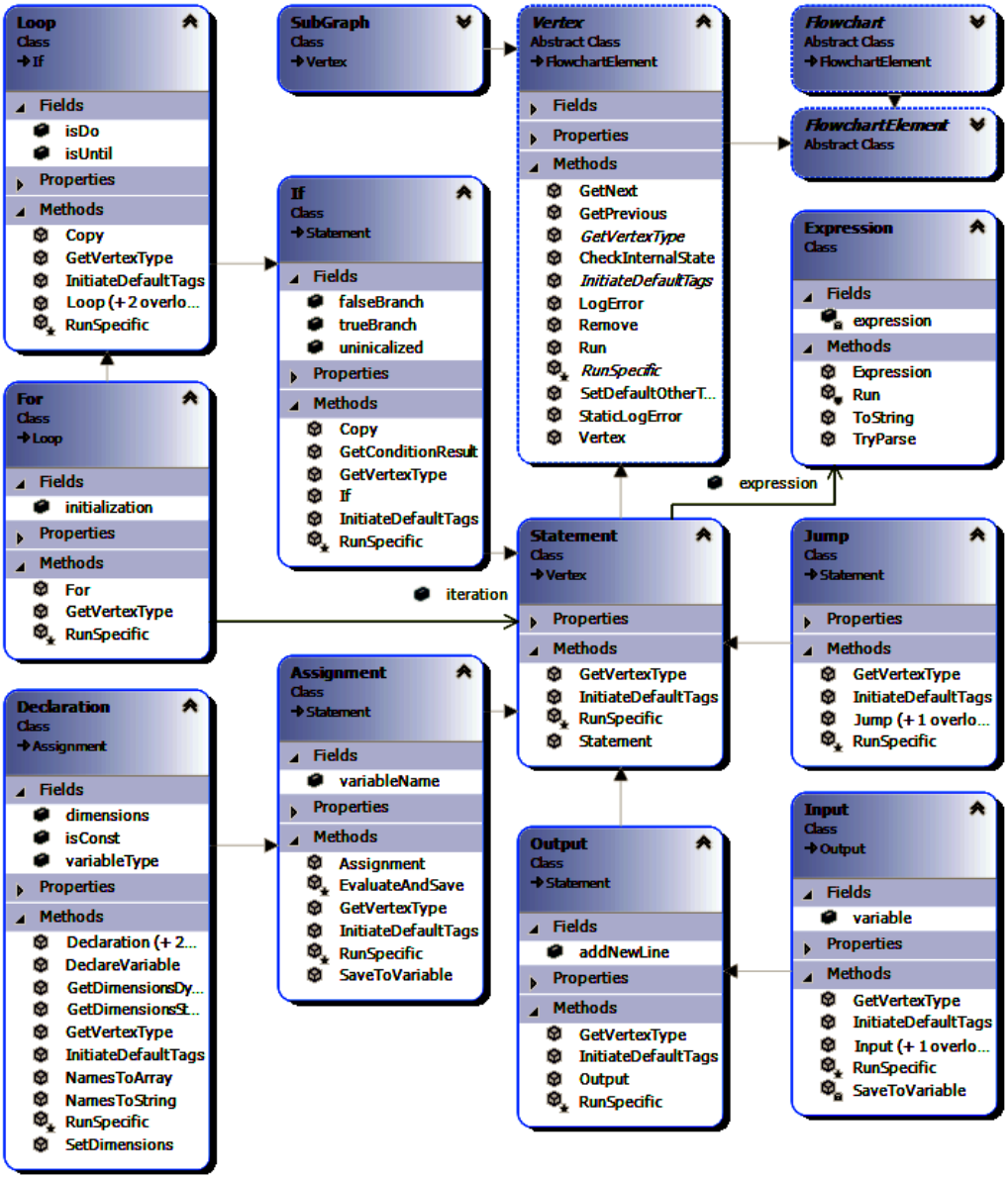
- *FlowchartElement* je abstraktní třída představující libovolný element vývojového diagramu, včetně podgrafů a vývojového diagramu samotného.

- *Flowchart* je abstraktní třída představující celý vývojového diagramu. Specifickou implementací jsou určité metody třídy *Method*.

- *Vertex* je abstraktní třída dědící od *FlowchartElement* vyjadřující uzel vývojového diagramu. Má odkaz na rodiče, tedy nejbližší podgraf, ve kterém se nachází, a dokáže tak získat svého předchůdce a následovníka v tomto podgrafu. Následovník i předchůdce může být *Vertex* nejvíce jeden uzel - větvení a více vláken řeší potomci třídy *Vertex*, které představují podgrafy. *Vertex* také obsahuje několik indikátorů stavu tohoto uzlu a prostředky (tagy), které představují grafické nastavení uzlu pro úpravu grafických komponent (nodů). Abstraktní metoda *Run* představuje interpretaci a debugging instance třídy *Vertex*, potomci si implementují pouze část z něj - *RunSpecific*.

- *SubGraph* je potomek třídy *Vertex*, který představuje podgraf jako sekvenci uzlů - tříd *Vertex*. Slouží pouze pro udržení informace o této sekvenci a její procházení. Při interpretaci postupně prochází všechny členy grafu, dokud nenarazí na *Jump* nebo nedojde na konec podgrafu.

- *Statement* je potomek třídy *Vertex*, který představuje neurčený příkaz, jehož výstup není přiřazen do proměnné - typicky by mělo jít o volání podprogramu. Příkaz je určený výrazem - instance třídy *Expression*. Vzhledem k tomu, že téměř všichni běžní potomci *Vertex* nějaký výraz vyhodnocují, dědí tak od *Statement*.



Obrázek 6.5: Diagram tříd v komponentě *Flowcha-n* s vazbou na třídu *Vertex*.

- *Expression* představuje výraz uložený jako string. Tento výraz může být testován na správnost syntaxe a případně také interpretován pomocí komponenty *ExpressionParser* - jedná se o jedinou třídu, která tuto komponentu používá.
- *Output* je potomek třídy *Statement*, který představuje výstup (do konzole) v rámci vývojového diagramu. Výsledek výrazu *Statement* se připojí k výstupní konzoli.

- *Input* je potomek třídy *Output*, který představuje vstup z konzole. Pokud výraz není prázdný, je použit jako prompt, který se vypíše do konzole před čekáním na vstup (je dobrým zvykem konzolový vstup předcházet žádostí o něj a tímto je tato praktika zjednodušena) - tato funcionalita je zděděna z třídy *Output*. Při interpretaci se na vstupu procházení grafu zastaví, dokud uživatel neukončí vstup tlačítkem enter. - Následně se napsaný řetězec zpracuje jako výraz a uloží do příslušné proměnné.

- *Assignment* je potomek třídy *Statement*, který představuje přiřazení do proměnné. Obsahuje název proměnné, který pokud obsahuje hranaté závorky, předpokládá přítomnost indexů pole.

- *Declaration* je potomek třídy *Assignment*, který představuje deklaraci. Obsahuje název a datový typ proměnné, kterou deklaruje, a případně také velikosti jednotlivých dimenzí. Výraz je použit k inicializaci proměnné zděděné ze třídy *Assignment*. Pokud je prázdný, inicializuje se na defaultní hodnotu datového typu.

- *Parameter* je potomek třídy *Declaration*, který představuje parametr podprogramu. Obsahuje název a datový typ proměnné, kterou deklaruje, informaci, jestli je předávaný hodnotou, nebo parametrem, a případně také velikosti jednotlivých dimenzí. Výraz v případě *Parameter*u v současnosti není použit, v budoucnu se předpokládá jeho použití jako defaultní hodnoty.

- *Jump* je potomek třídy *Vertex*, který představuje nějaký skok v programu - break, continue nebo return. Výraz je v případě možnosti return použit jako návratová hodnota.

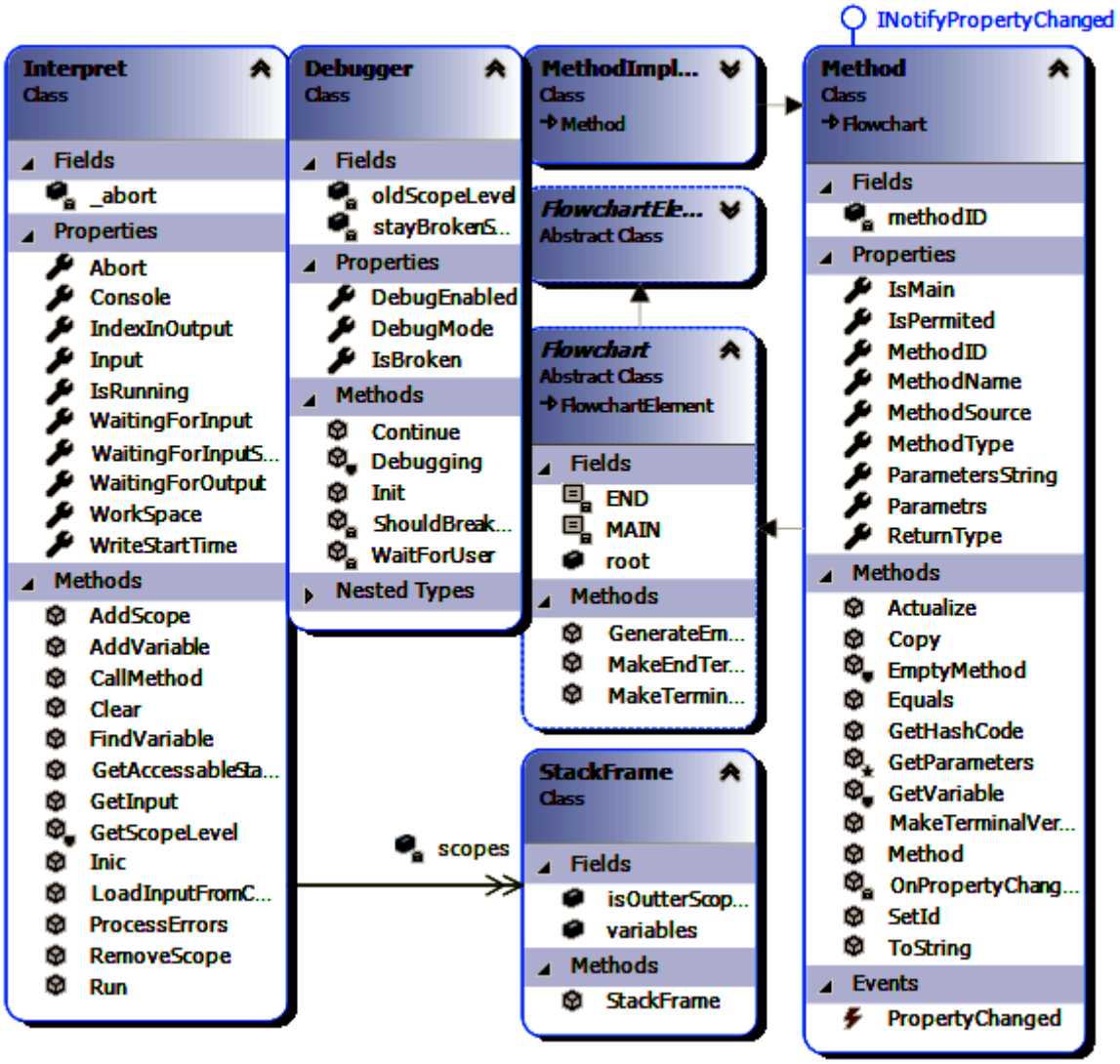
- *If* je potomek třídy *Statement*, který představuje podmínku. Má dva podgrafy odpovídající true a false větvi třídy *SubGraph*. Při interpretaci nejprve vyhodnotí výraz jako podmínku a následně interpretuje odpovídající podgraf.

- *Loop* je potomek třídy *If*, který představuje cyklus. Přes indikátory *isDo* a *isUntil* lze nastavit, o jaký typ cyklu se jedná - možné jsou while, do-while, until a repeat-until. Při interpretaci, pokud platí *isDo*, nejprve interpretuje podgraf představující tělo cyklu. Následně vyhodnotí vyraz představující podmínku, kterou zneguje, pokud platí *isUntil*. Pokud podmínka platí, interpretuje se podgraf představující tělo cyklu vždy znovu. Interpretace skončí předčasně, pokud narazí na *Jump* subtypu *Break* nebo *Return*.

- *For* je potomek třídy *Loop*, který představuje for-cyklus. Oproti *Loop* má navíc dvě instance třídy *Statement* - initialization a iteration. Jedna se vyhodnotí jen jednou na začátku interpretace, druhá se vyhodnotí na konci každé iterace.

Interpret

Třídy přímo související s interpretem mají strukturu znázorněnou na Obr. 6.6.



Obrázek 6.6: Diagram tříd souvisejících s interpretem v rámci komponenty *Flowcha-n*.

- *Flowchart* je třída představující vývojový diagram, která dědí od třídy *FlowchartElement*. Obsahuje odkaz největší přímou sekvenci uzlů v podobě podgrafu třídy *SubGraph*. Obsahuje rovněž metody pro generování prázdného diagramu a vytváření terminálních uzlů (začátku a konce diagramu).

- *Method* je potomek třídy *Flowchart*, který představuje metodu[4] v podobě vývojového diagramu. Má název, vstupní parametry, výstupní datový typ (kde typ *Void* značí, že je metoda procedurou) a další pomocní indikátory. Instanci třídy *Method* je možné interpretovat metodou *GetVariable*, která vrací instanci třídy *Variable*.

- *MethodImplicit* je potomkem třídy *Method*, který představuje nativní metodu C#, která je volána přímo z kódu. Konstantní seznam *Metods* obsahuje všechny defaultně podporované metody - všechny jsou v současnosti statické metody knihovní třídy *Math* a *Convert*. Kromě *MethodImplicit* jsou implementovány také metody z jiných nestatických nativních tříd - *Random*, *String* a *Double* - také získání velikosti pole a třída *MethodAlias*, která zajišťuje kompatibilitu s aplikací Flowgorithm, která používá jiné názvy metod.

- *StackFrame* je třída představující zásobníkový rámec. Každý obsahuje proměnné (instance *Variable*) a informaci, jestli je možné hledat proměnnou s nenalezeným názvem o zásobník dále (tzn. že zásobník před tímto není v jiné metodě). Model paměti je dále popsán ve sekci 6.3.5.

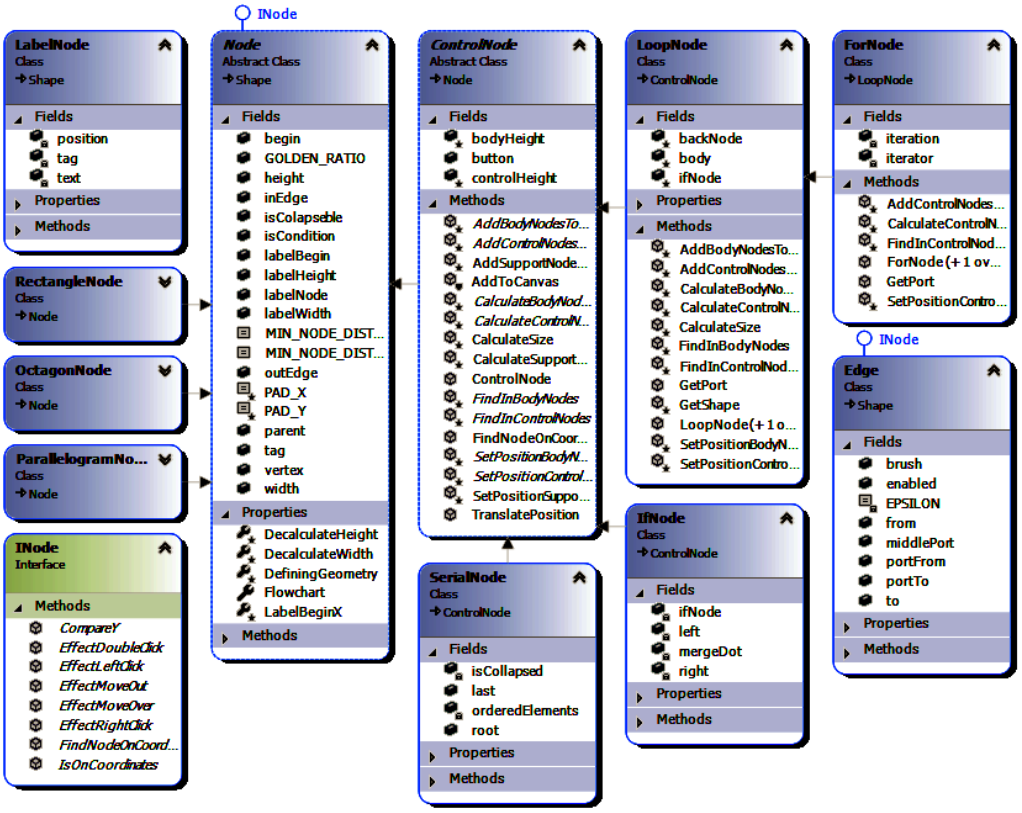
- *Interpret* je třída představující interpret vývojového diagramu. Obsahuje zásobník instancí třídy *StackFrame*, který funguje jako paměťový zásobník, a nástroje pro interpretaci vývojového diagramu/metod a uzlů. Umožňuje interpretaci vývojového diagramu (*Flowchart*) a volání metody (*Method*).

- *Debugger* je třída představující debugovací režim. Rozhoduje, kdy se má interpretace pozastavit na základě nastaveného režimu a uzlu a také kdy má znovu pokračovat.

Vizualizátor

Třídy přímo související s vizualizátorem mají strukturu znázorněnou na Obr. 6.7.

[4]Metoda v současnosti není vztažená k žádnému objektu - aplikace zatím neumožňuje vytvářet vlastní datové typy a objekty. Z terminologického hlediska se tedy v současnosti jedná o funkci.



Obrázek 6.7: Diagram tříd souvisejících s vizualizérem v rámci komponenty *GUI*.

- *INode* je rozhraní představující klikatelný objekt vizualizéru. Deklaruje metody pro detekci hit boxu a změnu objektu vlivem uživatelské interakce.

- *Node* je abstraktní třída dědící od wpf třídy *Shape* a implementující *INode*, která představuje uzel prezenční vrstvy vývojového diagramu. Obsahuje základní výpočet rozměrů uzlu, metodu pro získání nápisu, konstanty pro odsazení v rámci uzlu a mezi uzly, grafické nastavení stylu uzlu a všechny další funkcionality, které jsou společné pro všechny uzly na prezenční vrstvě. Důležité jsou abstraktní metoda *GetShape*, která vrací geometrii daného uzlu pro vizualizaci na prezenční vrstvě, a virtuální metoda *CalculateSize*, která nastavuje velikost uzlu dle velikosti jejího nápisu - opak, tedy velikost nápisu dle maximální velikosti uzlu, řeší property *DecalculateWidth* a *DecalculateHeight*.

- *LabelNode* je třída představující nápis na *Node*. Z *Node* je vyčleněn proto, aby mohl mít rozdílný styl a sama nemá jiné funkce. Efektivně je další *Shape*, který je vytvořen v rámci *Node*.

- *RectangleNode*, *OctagonNode*, *ParallelogramNode* a další třídy z package *shapes* představují potomky třídy *Node* různých tvarů. Tvar se rozhoduje ne na základě typu *Vertex* (logického uzlu), ale na základě nastavitelného tagu. Každá z těchto tříd má vytvoření vlastního tvaru v rámci metody *GetShape* a výpočet své velikosti v rámci *CalculateSize* (případně naopak velikosti nápisu).

- *Edge* je potomek třídy *Shape* implementující *INode*, který představuje orientovanou hranu na prezenční vrstvě. Uchovává si informaci o uzlu, ze kterého vychází a ve kterém končí, a také tři body, kterými prochází (začátek, konec a mezibod). Obsahuje také metody pro detekci hit boxu a řešení interakce uživatele s uzlem.

- *ControlNode* je abstraktní třída dědící od třídy *Node*, která představuje podgraf. Má tři různé části - tělo, hlavičku (označovaná jako control nodes) a podpůrné části (zatím pouze kolabovací tlačítko). Každá z těchto částí má příslušnou abstraktní metodu[5] pro výpočet velikosti, nalezení pozice a přidání na plátno (*Canvas*), které se volají v předem určeném pořadí. Potomkům stačí pouze metody implementovat.

- *SerialNode* je potomek třídy *ControlNode* představující nejjednodušší možný podgraf, který má pouze tělo a představuje sekvenci uzlů. Každý vývojový diagram má alespoň jeden *SerialNode*, který představuje hlavní graf.

- *IfNode* je potomek třídy *ControlNode* představující podmínku a její větve *True* a *False*, které představují tělo podgrafu - dvě instance třídy *SerialNode*. Konec těl je spojen do jedné výstupní části pomocí neinteraktivního spojnicového uzlu. Řídící část tvoří podmínka v podobě samostatného uzlu odpovídajícího stylu logického vrcholu diagramu (*Vertex*). V podgrafu je graficky odlišena hlavička a tělo obdélníky.

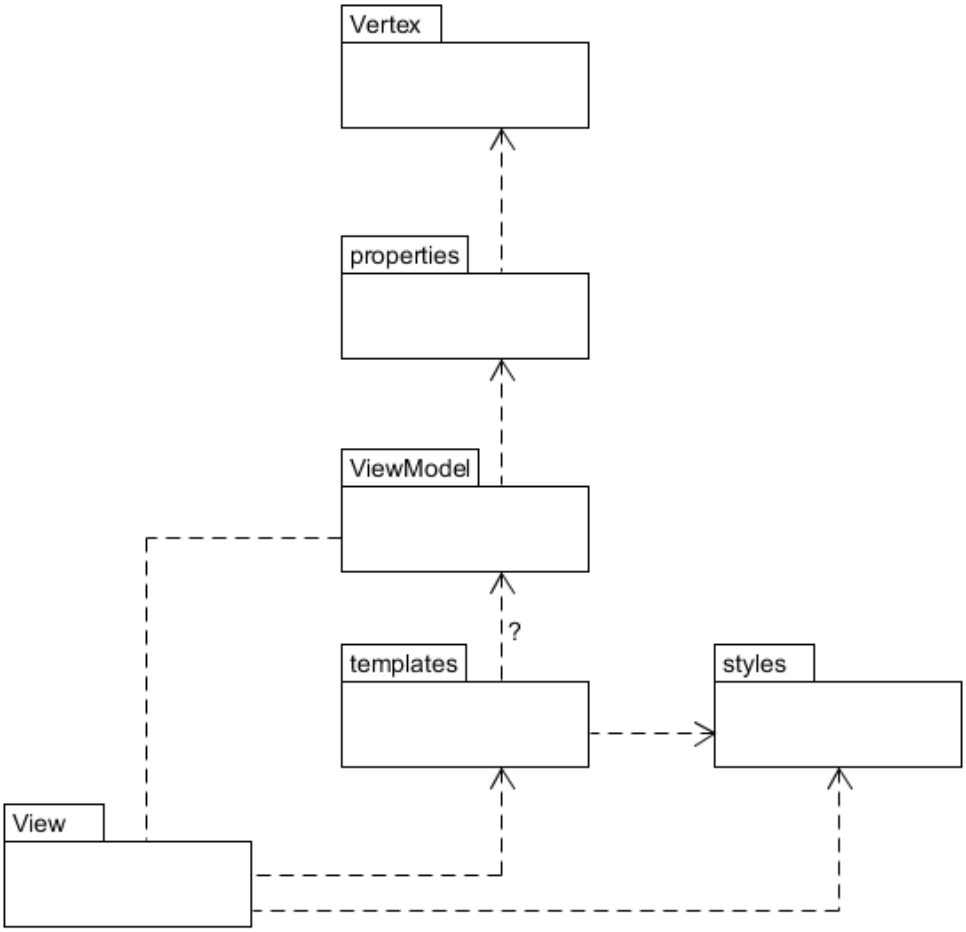
- *LoopNode* je potomek třídy *ControlNode* představující obecný cyklus grafu. Tělo je tvořeno jednou instancí třídy *SerialNode*. Kontrolními elementy jsou podmínka v podobě samostatného uzlu odpovídajícího stylu logického vrcholu diagramu (*Vertex*) a neinteraktivní spojovací uzel, který slouží pouze pro napojení konce těla zpět na podmínku.

[5]Kromě podpůrných částí, ty jsou virtuální a obsahují řešení kolabovacího tlačítka - to je stejné pro všechny podgrafy s výjimkou *SerialNode*.

- *ForNode* je potomek třídy *LoopNode* představující for-cyklus. Liší se pouze v kontrolních elementech, které můžou, dle nastavení, obsahovat příkazový uzel pro inicializace a iteraci (příkaz vykonaný na konci každé iterace). Rozložení a velikost těchto elementů je složitě nastaveno tak, aby se nikdy žádné části grafu nepřekrývaly, a to ani v případě, že v těle cyklu je další podgraf.

6.3.3 GUI

GUI má strukturu, která se do značné míry snaží dodržovat strukturu MVVM. Jejich skutečná struktura je znázorněnou na Obr. 6.8.



Obrázek 6.8: Diagram závislostí částí GUI v rámci komponenty *GUI*.

- *Vertex* je logický uzel grafu, který je součástí komponenty *Flowchan* - není tedy přímou součástí GUI. Z pohledu GUI je ekvivalentem uložených dat, neboť představuje vlastnosti daného uzlu.

- *properties* jsou třídy ze stejnojmenného package, které každá efektivně naplňují činnost modelu na úrovni jedné vlastnosti (jednoho typu dat). Řeší tak načtení původních dat, konverzi mezi daty od uživatele a ukládanými daty, uložení dat a případný rollback, tedy vrácení do původního stavu. Při běžném použití by *Vertex* a *properties* byly součástí jednoho celku, zde je toto rozdělení výhodnější jednak kvůli oddělení logického grafu, jednak kvůli opakovanému použití některých *properties* v různých *View*.

- *ViewModely* odpovídají *ViewModel*ům v rámci MVVM struktury, řeší veškerou logiku GUI a hlavně interakci uživatele. Dle MVVM by měl být *ViewModel* správně zcela nezávislý na View, to se v rámci implementace nepodařilo kompletně zajistit a *ViewModel* sice nepracuje přímo s instancí daného *Window*, ale z technických důvodů získává velikost aplikace z jeho komponenty *Grid* a v případě *MainWindowViewModel* také pracuje přímo s objektem typu *TextBox* přestavujícím konzoli aplikace.

- *styles* jsou třídy typu *ResourceDictionary* ve stejnojmenném package. Obsahují pouze grafické nastavení a rozvržení grafických komponent tak, aby byly vizuálně atraktivnější a přehlednější.

- *templates* jsou třídy typu *ResourceDictionary* ve stejnojmenném package. Každý z nich obsahuje *DataTemplate* odpovídající názvu dané třídy. Jedná se o buď opakované částí elementů *View* použité stejným způsobem na více místech nebo naopak o elementy, jejich rozvržení je závislé na datech a mění se podle nich. Viz Sekce 6.3.3.

- *View* jsou třídy dědící od třídy wpf *Window*, které obsahují samotnou vizuální komponentu a představují prezenční vrstvu v rámci MVVM. Samy o sobě neobsahují žádnou logiku, ale jen rozvržení grafických komponent, které je navíc dále ovlivněno styly a templaty, které používají.

MVVM

Technologie WPF je vytvořena tak, aby maximálně podporovala použití architektury MVVM. Použití jiných architektur, zvláště například takových, které by například využívaly dědičnost na úrovni View, nejsou doporučené a z principu budou s jejich použitím problémy. MVVM je pro většinu malých desktopových aplikací zbytečně složitá a zvláště pro backendové vývojáře zesložiťuje vývoj, neboť používá přístup, na které backendový vývojář není zvyklý, ale které se běžně používají při vývoji frontendu webových aplikací. Některé funkcionality a specifické problémy MVVM navíc nejsou v rámci WPF zcela dořešené a programátor je nucen si je implementovat sám. Flowcha-n je ovšem už dostatečně velká aplikace, aby se použití MVVM mohlo vyplatit, MVVM zároveň kromě technických obtíží přináší i mnoho výhod a výrazně elegantnější kód.

MVVM - Model-View-ViewModel - je třívrstvá architektura vycházející z MVP jejíž specialitou je oboustranná vazba (2-way binding) hodnoty proměnné na prezenční vrstvě a vrstvě kontroléru [44]. To probíhá tak, že je jako vlastnost grafického objektu určena vazba na property s názvem, který je identický názvu ve ViewModelu. Pokud dojde ke změně vlastnosti vlivem interakce uživatele, změní se také příslušným způsobem hodnota property ve ViewModelu. Pokud se změní hodnota property v rámci ViewModelu, změní se také příslušný vizuální objekt v rámci View.

Všechny tři části MVVM od sebe mají být přísně oddělené. Model má pracovat pouze s daty a nemá vůbec vědět o zbytku struktury. ViewModel obsahuje veškerou logiku a properties - k tomu musí implementovat rozhraní *INotifyPropertyChanged* a korektně notifikovat WPF o změně property[6]. ViewModel může používat Model, ale neměl by mít v rámci MVVM žádnou závislost na View. View je prezenční vrstvou MVVM a funguje na stejném principu jako prezenční vrstva webových aplikací - je tak možné importovat styly grafických komponent podobně jako při použití kaskádových stylů a například místo použití dědičnosti (ke které by v rámci View vůbec neměl docházet) se předpokládá použití templatů, které dané rozložení grafických komponent specifikují jinde. View v rámci MVVM nemá obsahovat žádnou logiku, jen pouze grafické objekty, jejich vlastnosti a části.

[6]WPF by si při 2-way bindingu i 1-way bindingu ve směru ke View mělo být schopno při změně property vždy samo zavolat příslušné notifikace. V praxi k tomu nedochází, pokud nebyl přímo zavolán setter a nejedná se o auto-property, k čemuž nemůže dojít, pokud došlo ke změně jen nějaké z částí property nebo property vůbec setter nemá. Skutečný problém je pak v rámci MVVM WPF stav, kdy při bindingu není aktualizováno property při změně na straně View. - Tento problém se zatím podařilo vyřešit pouze částečným porušením vzoru MVVM.

MVVM přináší několik zásadních výhod: logika je přísně oddělená a tím pádem se může testovat zcela samostatně v rámci unit testů a zároveň může jednoduše dědit. Naopak na prezenční úrovni je možné soustředit se pouze na vizuální stránku problému a ignorovat problémy logiky - to také znamená, že je možné View, který má různou logiku, ale používá graficky stejnou část, použít na více místech bez dalších komplikaci a zvětšování problémů právě na vrstvě ViewModelu.

Propojení s Vizualizérem

Z hlediska GUI se uživatel dostane do styku s vizualizérem pouze v rámci *MainWindow*, respektive jeho *MainWindowViewModel*. *MainWindow* obsahuje *TabControl*, v rámci kterých uživatel vybírá záložky představující otevřené metody - tyto záložky jsou třídou *FlowchartTabItem*.

FlowchartTabItem obsahuje název a *Canvas*, na který se vývojový diagram vykresluje. Samotné naklíčování metody (vývojového diagramu) na *FlowchartTabItem* se provádí v rámci třídy *GUIController*.

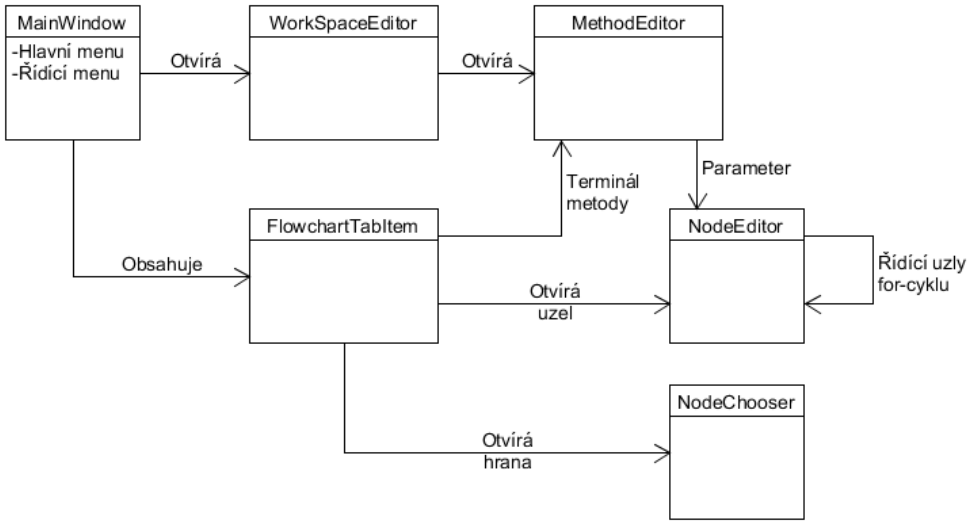
Třídy *GUIController* obsahuje pouze statické prvky a složí jako soubor služeb, v rámci kterých dochází ke změnám vývojového diagramu, kontaktu mezi GUI a Vizualizérem a správou záložek otevřených metod *FlowchartTabItem*. *GUIController* je úzce propojený s *MainWindowViewModel*, který kontroluje právě okno, které obsahuje *Canvas*, na který probíhá vizualizace vývojového diagramu. *MainWindowViewModel* je ale pomocí *GUIController*u odstíněn od přímého kontaktu s Vizualizérem.

Samotné vytvoření vizuálního vývojového diagramu (a tedy i jeho vizuální aktualizace) probíhá v metodě *Visualise* třídy *Visualiser*, která je volána právě pouze v rámci třídy *GUIController*.

Diagram oken

Aplikace se skládá z mnoha částí, se kterými může uživatel interagovat samostatně a tvoří samostatné logické celky. Ty můžeme rozdělit na: Hlavní menu, řídící prvky interpretace, záložky otevřených diagramů, vizualizace vývojového diagramu, editor jednotlivých uzlů (třída *NodeEditor*), správa metod (třída *WorkSpaceEditor*), editor metody (*MethodEditor*) a editor parametru (což je také *NodeEditor*).

Pro uživatele by bylo nepřehledné používat všechny tyto části najednou, a proto jsou v případě potřeby rozděleny do vlastních oken a zpřístupněny způsobem, který je popsán v případech použití (Sekce 6.2) a je inspirován Flowgorithmem. Přehledněji způsob přenosu mezi jednotlivými okny/třídami ukazuje Obrázek 6.9.



Obrázek 6.9: Diagram znázorňující uživatelský přístupu k jednotlivým částem *GUI*.

Jedná se o tyto třídy:

- *MainWindow* - *MainWindow* je už dle názvu hlavní okno, které se spustí po spuštění aplikace. Obsahuje hlavní menu pro správu programu (uložení, načtení apod.), řídící prvky pro interpretaci vývojového diagramu a také záložky s metodami, které jsou třídami *FlowcharTabItem*.

- *FlowchartTabItem* - Instance třídy *FlowchartTabItem* jsou položkami třídy *TabControl* obsažené v rámci okna *MainWindow*. *FlowchartTabItem* odpovídá jedné metodě (*Method*), kterou obsahuje a která je vizualizována na její grafickou plochu *Canvas* jako vývojový diagram tvořený hranami a uzly (včetně mezních značek - terminálů).

- *NodeChooser* - *NodeChooser* je samostatné okno, které se otevře kliknutím na interaktivní hranu. Představuje nabídku uzlů, které je možné na danou hranu vložit.

- *WorkSpaceEditor* - *WorkSpaceEditor* je samostatné okno, které se otevře kliknutím na příslušnou položku v hlavním menu. Představuje správu metod, které je v rámci tohoto okna možné vidět, upravovat, mazat a také přidávat.

- *MethodEditor* - *MethodEditor* je samostatné okno, které se otevře kliknutím na kliknutím na tlačítko Edit nebo Add v okně *WorkSpaceEditor*. Představuje editor hlavičky metody (tělo metody tvoří vývojový diagram znázorněný v rámci položky *FlowchartTabItem*) a jejích vlastností. *MethodEditor* lze také otevřít z vývojového diagramu dvojklikem na terminální uzel - místo editoru uzlu se otevře *MethodEditor*, protože uzel představuje hlavičku metody a tato reakce je tedy logicky implikována okolnostmi.

- *NodeEditor* - *NodeEditor* je samostatné okno představující editor uzlu, ať už jakéhokoliv, a jeho vlastností. Je velmi univerzální a používá se v různých případech. Základní přístup k němu je dvojklikem na uzel ve vývojovém diagramu. Vzhledem k tomu, že parametr metody je také typem uzlu (uzel *Parameter*), je také možné otevřít *NodeEditor* kliknutím na tlačítko Edit nebo Add v okně *MethodEditor*. Speciální případem je pak uzel for-cyklu (*For*), který sám obsahuje ve své hlavičce další uzly - pro inicializaci cyklu a ukončení iterace - tyto uzly je možné editovat otevřením okna *NodeEditor* jak přímo z vývojového diagramu, tak také z okna *NodeEditor* for-cyklu, ke kterému náleží.

Aplikace se tak sestává z velkého množství oken a má složitou strukturu, ale tato struktura je praktického důvodu a umožňuje uživateli se ke každé části aplikace dostat přes dvě okna. Někteří testeři považovali i to za příliš a dali by přednost tomu, kdyby se celá aplikace sestávala jen z jediného okna, v takovém případě by toto okno ovšem mělo mnoho různých částí přístupných najednou, bylo nepřehledné a na jednotlivé části by zůstalo jen málo místa.

6.3.4 Parser výrazů

Důležitou součástí interpretovatelných vývojových diagramů je výraz obsahující literály, proměnné a operace s nimi (včetně případných volání podprogramů). Pro upřesnění pár příkladů těchto výrazů a jejich interpretace:

- $|1 + 1|$ interpretováno jako proměnná typu integer o hodnotě 2.
- $|a < 7|$ interpretováno jako proměnná typu bool o hodnotě true, pokud proměnná a je integer menší než 7.
- $|"Největší\ číslo\ je: "+(string)42|$ interpretováno jako proměnná typu string o hodnotě "Největší číslo je: 42".
- $|Cos(3.14)|$ interpretováno jako proměnná typu real o hodnotě přibližně -1.

Tyto výrazy můžou být velmi složité a pro správnou funkčnost interpretu musí být jednoznačně interpretovatelné. V rámci aplikace se o parsování výrazu stará komponenta *ExpressionParser*, která je nezávislá na samotném interpretu a využívá knihovnu ANTLR.

Knihovna ANTLR

Knihovna ANTLR [38] je nástroj pro parsování výrazů na základě bezkontextové levo-levé gramatiky v *Extended Backus-Naur form* pro něj vytvořené. Na základě této gramatiky ANTLR generuje překladač obsahující lexikální, syntaktickou a omezeně sémantickou analýzu, korektním výstupem překladu může být strom, který je možný procházet a interpretovat dle potřeby.

Knihovnu ANTLR je v rámci C# relativně složité nakonfigurovat, má totiž více verzí od různých autorů, které jsou rozdělené do více částí a fungují vždy jen s určitou verzí .NET. V rámci tohoto projektu se osvědčila kombinace *Antlr4.Runtime.Standard 4.7.2*, *Antlr4BuildTasks 1.0.8*, *System.CodeDom 5.0.0* v rámci projektu typu class library s frameworkem *.NET 5.0*. Knihovna byla původně používána s *.NET Standard 2.0*, to ale vedlo k problémům kvůli tomu, že zbytek aplikace používal jiný framework. Po přechodu na *.NET 5.0* se tyto problémy již neprojevily.

Gramatika

Vytvořená gramatika byla založená na upravené a redukované gramatice jazyka C#. Ve zjednodušené podobě je gramatika ilustrována na listingu 1.

```
expression
: OPEN_PARENS expression CLOSE_PARENS                #parenthesis
| IDENTIFIER (OPEN_BRACKET expression CLOSE_BRACKET)+ #index
| expression (OP_INC | OP_DEC)                       #post_unary
| op_pre expression                                  #pre_unary
| OPEN_PARENS datatype CLOSE_PARENS expression       #cast
| expression (STAR | DIV| MOD) expression           #binary_op_multiply
| expression (PLUS| MINUS) expression               #binary_plus
| expression shift  expression                      #binary_shift
| expression compare expression                     #compare
| expression (OP_EQ | OP_NE) expression             #equal
| expression bit_wise expression                    #binary_bit
| expression (OP_AND | OP_OR) expression            #and
| expression QUESTION expression COLON expression   #ternary
| IDENTIFIER OPEN_PARENS
(expression (COMMA expression)*)?
CLOSE_PARENS                                        #method
| INTEGER_LITERAL                                   #integer
| REAL_LITERAL                                      #real
| CHARACTER_LITERAL                                 #char
| BOOL_LITERAL                                      #bool
| STRING_LITERAL                                    #string
| IDENTIFIER                                        #indetifier
;
```

Listing 1. Zjednodušená podoba použité gramatiky. Snippet tokenu "expression"

Gramatika bere v úvahu pořadí operací tak, jak funguje v rámci jazyka C#. Listing 1 ukazuje toto pořadí operací postupně: přednost závorek, přístup do pole (přes indexy), unární inkrementace a dekrementace, ostatní unární operace (včetně například negace), přetypování (cast), binární operace na úrovni násobení, binární operace na úrovni plus, bitové posuny, porovnávání, rovnost a nerovnost, bitové operace, logický and a or, ternární operátor, metoda, literály datových typů a nakonec identifikátor (proměnná).

Implementace

Výraz je rozparsován a je vytvořen parsovací strom. Za předpokladu, že během překladu nedojde k (syntaktické) chybě, je tento strom následně procházen a během průchodu vytvořen strom výrazů, kde každý vyraz lze buď vyjádřit přímo jako proměnnou (například "1"nebo "a"), nebo jako proměnnou vyjádřit na základě vyjádření podvýrazů, které výraz obsahuje ("1+1"nebo "(int)a/2"). Pro operace navíc platí následující:

- Každá operace je platná jenom s předem daným výčtem povolených datových typů na vstupu (týto výčty jsou níže).
- Pokud má operace více proměnných na vstupu a nejedná se o ternární operátor, metodu nebo výběr z pole, všechny mají stejný datový typ[7].
- Pokud se jedná o ternární operátor, první vstupní proměnná je datového typu bool. Druhá a třetí proměnná jsou libovolného datového typu, ovšem oba shodného.
- Pokud se jedná o metodu, vstupní datové typy jsou dány jejími parametry.
- Pokud se jedná o výběr z pole, první parametr je vždy typu pole a všechny následující typu integer.
- Pokud jde o operátor porovnání, rovnosti nebo nerovnosti, výstupným datovým typem je bool.
- Pokud jde o ternární operátor, výstupní datový typ odpovídá datovému typu druhé a třetí proměnné.
- Pokud jde o operaci konverze (cast), výstupní datový typ odpovídá typu konverze.
- Pokud jde o operaci výběr z pole, výstupní datový typ odpovídá datovému typu položek v poli.
- Pokud jde o operaci zavolání metody, výstupní datový typ odpovídá datovému typu určenému návratovou hodnotou.
- Ve všech ostatních případech je výstupní datový typ shodný se vstupním datovým typem.

[7]Speciálně v případě použití přepínače *Implicit Conversion* v případě opaku nejde o chybu - v takovém případě se proměnná užšího datového typu automaticky zkonvertuje na širší datový typ a operace následně může pokračovat stejným způsobem.

Operace mají následující výčty povolených datových typů[8]:

- Inkrementace a dekrementace - integer, real, char.
- Unární plus a mínus - integer, real, char.
- Negace - bool.
- Konverze - integer, real, char, bool[9]. Dále je povolena ze stringu do stringu.
- Násobení, dělení, modulo - integer, real, char.
- Binární mínus - integer, real, char.
- Binární plus - integer, real, char, string.
- Levý a pravý shift - integer, char.
- Porovnání - integer, real, char.
- Bitové operace and[10], or a xor - integer, char, bool.
- Bitová negace - integer, char.
- Logické operace and a or - bool.
- Ve všech ostatních případech jsou povolené všechny datové typy, pokud to není v rozporu s jiným pravidlem v této subsubsekci.

6.3.5 Interpret

Interpretace vývojového diagramu spočívá v postupné interpretaci jeho uzlů v pořadí dané návazností uzlů od počátku (uzel, kterému žádný nepředchází). Níže je určené chování jednotlivých uzlů odpovídajícím stejnojmenným třídám:

- *Statement* - Uzel pouze interpretuje výraz v něm vložený (viz sekce 6.3.4). Pokud obsahuje volání metody, vstupuje do ní - stejně jako vždy, když interpretuje výraz.

[8]Operace nad charem se chovají jako 16bitový integer a nejspíš nejsou ve většině případů užitečné. Některé z nich budou možná v budoucnu redukovány

[9]Ve směru z datového typu. Konverze je povolena do všech základních datových typů (integer, real, char, bool, string), chování odpovídá chování při konverzi v jazyce C#.

[10]Speciálně u operace bitového And je navíc povolen datový typ string a proběhlá operace je identická s operací plus (tedy dojde ke spojení řetězců). K této úpravě došlo pro zachování kompatibility s Flowgorithm.

- *Assignment* - Interpretuje výraz stejně jako *Statement* a následně jeho hodnotu uloží do proměnné (nebo proměnných - viz *Declaration*) v něm určené.
- *Declaration* - Přidá do zásobníku novou proměnou se zadaným jménem, datovým typem a defaultní hodnotou. Pokud je vyplněný výraz, zároveň ho interpretuje a hodnotu rovnou do této proměnné uloží. V kolonce se jménem proměnné může být jmen vyplněno více, pokud jsou oddělena čárkou, to se pak chápe jako vytvoření několika stejných proměnných s různými jmény. V případě pole je na všechny jeho indexy při inicializaci uložena stejná hodnota.
- *Output* - Interpretuje výraz a připojí jeho hodnotu k textu výstupní konzole.
- *Input* - Počká na zmáčknutí klávesy enter a následně uloží uživatelem nově napsaný text z konzole do proměnné s určeným názvem (případně do více proměnných naráz). Pokud je vyplněný výraz, interpretuje jej a vypíše do konzole před čekáním na zadání vstupu.
- *SubGraph* - Chová se jako samostatný vývojový diagram. Postupně interpretuje své uzly od počátku. Tento postup ukončí předčasně, pokud narazí na uzel typu *Jump* (break, continue nebo return). Před začátkem interpretace *SubGraph*u je přidán do zásobníku nový zásobníkový rámec a po skončení jen odstraněn.
- *If* - Interpretuje výraz. Dle jeho pravdivosti vybere jeden ze svých uzlů třídy *SubGraph* (true/false) a ten interpretuje.
- *Loop* - Má výraz představující podmínku a *SubGraph*, který představuje tělo. Dle nastavení instance v různém pořadí opakovaně interpretuje výraz a následně *SubGraph*, dle pravdivosti/nepravdivosti interpretovaného výrazu. Pokud narazí na Break nebo Continue, přestává předčasně.
- *For* - Stejné jako *Loop*, jen má navíc jeden *Statement*, který interpretuje předtím, než začne všechno ostatní, a druhý, který interpretuje vždy po interpretaci *SubGraph*u. Jedná se tedy o obecný for-cyklus jehož vzorem je for-cyklus jazyka C.

- continue - Pokud se při interpretaci vyskytne tento uzel, skončí interpretace současného posledního *SubGraph*u předčasně, ale vše dál pokračuje beze změny. Mimo *SubGraph*u cyklu (*Loop*/*For*) by se neměl vyskytnout, pokud se tak stane, funguje identicky s uzlem typu return bez návratové hodnoty, což u metody, která není procedurou, vyvolá chybu za běhu programu.

- break - To samé jako continue, jen ukončí předčasně interpretaci celého uzlu *Loop*/*For*. Mimo cyklus platí to samé, co u continue.

- return - Předčasně ukončí interpretaci současné poslední metody, nebo případně celého vývojového diagramu, pokud žádná metoda právě není interpretována. U metody se jedná o korektní ukončení.

- *Method* - Nejprve podobně jako *SubGraph* přidá do zásobníku nový rámec. Následně do zásobníku přidá proměnné odpovídající vstupním parametrům názvem a hodnotou. Pokud je proměnná předávána hodnotou, je vytvořena nová proměnná a obsah se kopíruje. Pokud je předávána odkazem, do hodnoty nové proměnné se uloží odkaz na již existující proměnnou v jiném zásobníkovém rámci. Následně interpretuje své uzly stejně jako *SubGraph*, dokud nenarazí na return nebo v případě procedury na konec diagramu. Výraz returnu se interpretuje a navrací jako návratová proměnná. Po skončení se opět odstraní zásobníkový rámec.

Paměťový model

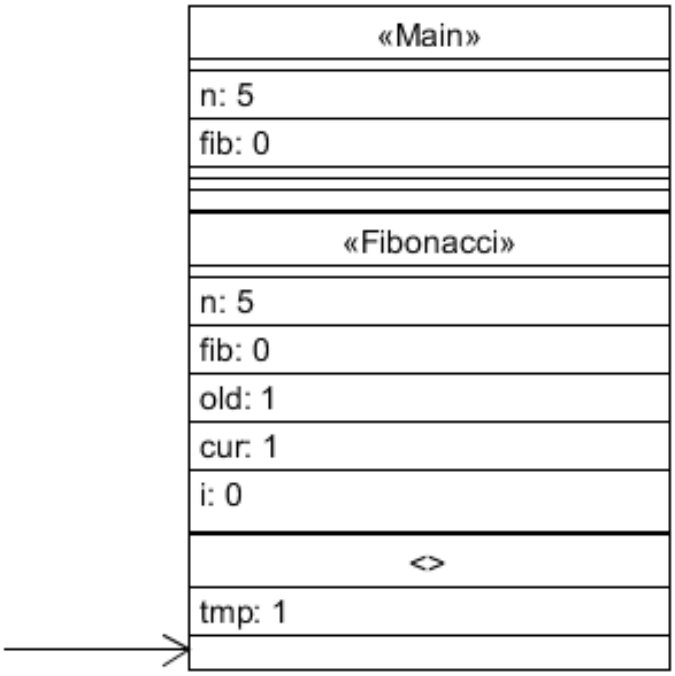
Interpret používá pro potřeby interpretace vývojového diagramu virtuální zásobník obsahující aktuálně známé pojmenované proměnné v zásobníkových rámcích. Tento zásobník neobsahuje nepojmenované proměnné, ani návratovou hodnotu nebo pozici, s nimi se pracuje jen jako s obyčejnými proměnnými, které jsou předávány pouze interně v rámci rekurze. Interpret si také uchovává seznam známých metod, které můžou být volány (v rámci vyhodnocení výrazů).

Zásobníkový rámec se přidává při vstupu do těla cyklu, větve podmínky anebo do metody a při výstupu z těchto částí se pak tento rámec opět odstraňuje. Rozdíl mezi vstupem do metody a do větve podmínky nebo těla cyklu je ten, že z rámce metody se při hledání proměnné interpret dál nevrací.

Interpret neobsahuje žádnou zvláštní strukturu, která by odpovídala haldě, data jsou na haldě samotného programu - proměnné mají přímo odkaz na svůj datový typ a hodnotu, která má interní podobu proměnné object, která obsahuje skutečnou hodnotu proměnné.

Nepojmenované proměnné jsou jen dočasné a pracuje se s nimi pouze v rámci vyhodnocování výrazu, jehož jednotlivé části jsou vyhodnocovány interním rekurzivním procházením parsovacího stromu díky *ANTLR*. Parametrické proměnné se přidají do zásobníku během vstupování do metody. Návratová hodnota je vracena jako bezejmenná proměnná (třída *Variable*) pomocí metody *Run* třídy *SubGraph* a metody *GetVariable* třídy *Method* - metoda je tedy vyhodnocena v rámci výrazu, ve kterém je volaná, pomocí interní rekurze.

Příkladem stav zásobníku na Obr. 3.1, by při zastavení na řádku "*cur = tmp*"(pokud by nešlo o vývojový diagram ve Flowgorithmu, ale v novém programu) vypadal, jak ukazuje Obr. 6.10.



Obrázek 6.10: Stav zásobníku nového programu, pokud by vývojový diagram vypadal jako na Obr. 3.1 a program se zastavil a řádku "*cur = tmp*".

6.4 Vizualizace vývojového diagramu

K vizualizaci, tedy vytvoření vizuálního vývojového diagramu, dochází v rámci Vizualizátoru (Sekce 6.3.2). Má tři fáze - v rámci první dojde k vytvoření samotných objektů a spočítání jejich velikosti, v rámci druhé dochází k nalezení pozice na platně (*Canvas*), kam mají být umístěna, a nakonec ve třetí fází dochází k samotnému vložení na plátno.

6.4.1 Vytvoření *INodů*

Prvním vytvořeným *INodem* je vždy *SerialNode*, který představuje hlavní graf. Ten následně vytváří všechny uzly a hrany v něm a určuje jejich pozici na ose y na základě jejich velikostí - pozice y je neměnná. Vytvoření každé instance třídy *Node* má stejný základ:

1. Uložení základních vlastností uzlu na základě vstupních parametrů.
2. Nastavení grafického stylu (barev, čárkování hranice apod.)
3. Vytvoření nápisu uzlu a nalezení velikostí jeho obdélníku.
4. Výpočet velikosti celého uzlu na základě velikostí nápisu nebo v případě podgrafu uzlů v něm. - V takovém případě jsou tyto uzly vytvořeny stejným způsobem. Konstruktor uzlu se přitom vybírá pomocí třídy *ShapeFactory* na základě jednak grafického stylu udávajícího tvar u běžného uzlu, jednak samotného typu *Vertex* v případě podgrafu.

6.4.2 Translace pozice *Nodů*

Takto vytvořené uzly jsou v rámci souřadnice x vycentrované na střed plátna, protože před vytvořením větví podgrafu nejde předem říct, jak budou široké, a tedy kde by měly předem mít pozici. V rámci metody *TranslatePosition* tak každý *Node* dostane své centrum v rámci osy x a příslušně si nastaví svou souřadnici. Stejně tak zavolá rekurzivně *TranslatePosition* nad všemi svými uzly v případě, že jde o podgraf a následně nastaví pozice hran, které vedou mezi jeho uzly. Nastavit pozice hran lze až v této fázi, protože předtím neznáme souřadnici x.

6.4.3 Přidání na plátno

Takto vytvořené *INode* mají správné pozice, ale nejsou ještě na plátně. Na plátno jsou dány pomocí metody *AddToCanvas*, která se identickým způsobem rekurzivně volá přes všechny *INode*, které *Node* obsahuje. Až při této fázi také dochází k vytvoření vizuálního nápisu (*LabelNode*). Pokud došlo k zastavení interpretace na tomto daném uzlu (udává stav logického uzlu *Vertex*), pak také dochází k automatickému scrollování k tomuto uzlu.

K přidání na plátno by mohlo docházet najednou spolu s translací pozice, pro přehlednost operací k ní ovšem dochází zvlášť.

6.4.4 Nastavení tvaru

Každý *INode* zároveň dědí od třídy *Shape* a implementuje její property *DefiningGeometry*, které vytváří geometrii objektu. Tu u běžného uzlu řeší metoda *GetShape*, která rovnou vytvoří geometrii uzlu jako uspořádanou množinu cest. V případě podgrafu tato metoda není užita a k vytváření geometrie se používá přímo *DefiningGeometry*. Podgrafy totiž samy obsahují pouze ohraničení hlavičky a těla grafu, zbytek geometrie tvoří uzly, které obsahují a přidávají na plátno metodou *AddToCanvas*.

6.4.5 Detekce hit boxů

Třída *Shape* sama o sobě zajišťuje detekci dotyku kurzoru myši s instancí třídy *Shape* na plátně - tato možnost byla jednou z důvodů, proč byl *Shape* použit a část vývoje byla teto vlastnost využívána. Bohužel jsou ale hit boxy *Shape* nastavené tak, aby nedetekovaly dotyk jen podle pozice kurzoru a objektu, ale hlavně také podle obarvených pixelů. To je problematické například u *Shape* s průhledným pozadím, které tak je možné vybrat jen na hranici nebo u úzkých objektů jako jsou hrany, které mají jenom pár pixelů na šířku, ale vybrat by je mělo být snazší.

Detekci lze upravit, nejedná se ovšem o vlastnost třídy *Shape* a způsob je natolik komplikovaný, že se ukázalo jako podstatně jednodušší řešení napsat od základu zcela vlastní detekci hit boxu. Výrazným zjednodušením byl k tomu také fakt, že nebylo třeba vytvářet další strukturu pro prohledávání objektů na plátně, protože vývojový diagram sám obsahuje grafový strom.

Při každém pohybu kurzoru na plátnu vývojového diagramu pohyb zaregistruje *FlowchartTabItem*, který zavolá metodu *FindNode* třídy *GUIController* k s pozicí kurzoru. Ta volá nad hlavním grafem metodu *FindNodeOnCoordinates*, která rekurzivně volá stejnojmennou metodu svých uzlů a hledá interaktivní uzel, nad kterým je kurzor. Pokud je takový uzel nalezen, je nad ním zavolána metoda příslušící dané akci uživatele[11].

Nedochází však k procházení celého grafu, ale pouze té části, obsah jejichž bounding boxu koliduje s pozicí kurzoru. K nalezení uzlu tak dochází v nejhůře logaritmickém čase, navíc pro praktické použití je nepravděpodobné velká úroveň zanoření podgrafu, tu tak můžeme považovat pro praktické účely za konečnou a k vyhledávání tak efektivně dochází v konstantním praktickém čase.

6.5 Shrnutí stavu aplikace

Po dlouhém vývoji se podařilo implementovat všechny důležité části a zároveň vychytat téměř všechny nalezené bugy. Některé části byly v průběhu vývoje plánovány nebo navrhnuty, ale zatím nebyly dokončeny a jsou tak možným směrem, kterým se aplikace může zlepšovat. Jmenovitě jde o:

- Sémantická typová kontrola před spuštěním
- Export do obrázku
- Přesun uzlů přes Drag'n'Drop
- Tooltip přímo nad uzly diagramu
- Vstup a výstup, který by umožnil souborové čtení a zápis přímo v rámci interpretace vývojového diagramu
- Další vylepšení GUI

Zároveň do podařilo implementovat velké množství funkcionalit, které Flowgorithm nemá:

- Podmínky a cykly je možné kolabovat a expandovat
- Vstup má možnost rovnou vypsat prompt

[11]Operace nad uzlem tedy odpovídají mechanizmům, které dochází k jejich vyvolání, a nejsou od sebe odděleny. Případný loose coupling těchto mechanizmů by výrazně znesnadňoval orientaci v kódu a patrně vedl ke kompletní dedikované mezivrstvě mezi GUI a Vizualizátorem, což se zdá být v současnosti nadbytečné.

- Vstup a výstup jsou zřetelně odlišné - barevně, tvarem a výstup začíná slovem "Output". Editor vstupu navíc vypadá jinak a upozorní uživatele na chybu, pokud by nevyplnil název proměnné, do které se má vstup vložit.
- Při deklaraci je možné proměnnou inicializovat
- Při deklaraci je možné celé pole inicializovat na jednu hodnotu
- Umožňuje unární inkrementaci (i++)
- Syntaxe je kontrolována už při psaní
- For-cyklus je podstatně otevřenější C-based for-cyklus
- Deklarace, přiřazení a vstup umožňují uložit stejnou hodnotu do více proměnných naráz
- Metody je možné importovat z jiného souboru
- Program se na základě testování ovládá intuitivněji

7 Vývoj

Vývoj aplikace je možné rozdělit do několika fází:

1. První průzkum, v rámci kterého bylo zjištěno, jaká je situace, co bude účelem diplomové práce, a byly získány základní předpoklady pro analýzu.
2. Analýza, v rámci které došlo k návrhu aplikace včetně architektury a použitých technologií.
3. Vývoj interpretu, v rámci kterého byla vytvořena komponenta *Flowcha-n* (viz Sekce 6.3.2) a byl stanoven způsob uživatelského užití v částech, které aplikaci rozporují s Flowgorithmem.
4. Vývoj Alfa verze, v rámci které byla vyvinuta první funkční verze, došlo k jejímu testování a pokusu o obhajobu.
5. Vývoj vlastního vizualizátoru, který nahradil dále nepoužitelnou knihovnu *MSAGL* (viz Sekce 5.8).
6. Vývoj první Beta verze, která byla určena pro vyzkoušení vyučujícími z fakulty strojní z hlediska nutných úprav pro použití ve výuce.
7. Vývoj druhé Beta verze, která měla být použita v rámci výuky v letním semestru 2022 na fakultě strojní (k čemuž bohužel nedošlo).
8. Vývoj plné verze, která byla testována studenty v rámci předmětu UUR a dalšími zájemci.

7.1 První průzkum

K první událost týkající se vývoje aplikaci Flowcha-n došlo 14. 2. 2020 v rámci prvního setkání s vyučujícími z fakulty strojní. V rámci diskuze bylo vysvětleno, že vyučující v rámci předmětu TI relativně spokojeně využívají pro výuku algoritmizace aplikaci Flowgorithm, ale že mají k aplikaci několik výhrad, zvláště nemožnost použít vícerozměrné pole a nemožnost importovat metody z jiného souboru Flowgorithmu. Zároveň jim byla představena jako alternativa aplikace PS Diagram (viz Sekce 3.3.3), která ovšem byla také viděna jako suboptimální. Na základě diskuze vyšlo najevo, že s autorem Flowgorithmu probíhala dlouhodobá snaha o spojení pro účel spolupráce

na vývoji Flowgorithmu, která ale byla doposud neúspěšná. Vyučující z fakulty strojní tak vyjádřili velký zájem na vytvoření vlastní obdoby aplikace Flowgorithm, která by byla v C#, využívala moderní technologie (WPF) a umožnila překlad vývojového diagramu do kódu C#.

Ze setkání byl 18. 2. 2021 zapsán výstup na MS Teams[1], který se stal na základě dohody primárním komunikačním kanálem. Zásadní problém v tu dobu činil rozsah práce - nakonec byl zvolen jako kompromis, tedy tak, aby nebyl zaměřený jen na analýzu, ale aby byl v limitované formě funkční a následující rok bylo možné ho případně doladit. 19. května pak došlo k vytvoření zadání práce, které by tento kompromis mělo reflektovat.

Už dne 18. 2. také došlo k prvním průzkumům grafických knihoven a prozkoumání funkčnosti aplikace Flowgorithm jakožto vzoru pro novou aplikaci[2].

7.2 Analýza

Čas nutný na zhotovení vlastního vizualizátoru byl odhadován na 60 člověkohodin, tento odhad byl ovšem výrazně podhodnocený, neboť nebral v úvahu různé stavy a nastavení vývojového diagramu a počítal s konstantní velikostí uzlů. 19. 5. 2020 bylo tak během setkání ohledně obsahu zadání diplomové práce rozhodnuto, že pro vizualizaci vývojových diagramů bude z časových důvodů použita externí knihovna.

Během května, června a července bylo úsilí, kromě jiných školních povinností, soustředěno na nalezení vhodné grafické knihovny schopné zmíněnou funkci zastat a také první návrh vnitřní struktury aplikace. Ty byly naznačeny v rámci druhé schůzky s vyučujícími z fakulty strojní, ke které došlo 15. 7. 2020.

Během druhé schůzky[3] bylo hlavně vyjasněno, jaké by měly být použity datové typy (int, float, string, boolean a jejich pole - alespoň 1D a 2D), jak by měly být vyhodnocovány v rámci výrazů (povolena jen explicitní konverze), že je u parametrů metod třeba umožnit nastavení předávání hodnoty odkazem nebo hodnotou a že aplikace má být z grafického hlediska

[1] K dispozici v souboru *Vysledky/Záznamy/zaznam1.txt*

[2] Flowgorithm nepoužívá žádnou grafickou knihovnu pro vykreslování vývojových diagramů, ani pro načítání a ukládání vývojových diagramů ve XML formátu. Nebylo tedy možné jej brát za vzor i v tomto ohledu.

[3] K dispozici v souboru *Vysledky/Záznamy/zaznam2.txt*

minimální v tom smyslu, že nemá umožnit uživatele dělat zbytečné akce, kterými by mohl zmást sám sebe, jako například hýbání s uzly a hranami nebo změnu grafického stylu uzlů. Ze schůzky byl vytvořen záznam výstupu na MS Teams.

16. 9. 2020 byl vytvořen harmonogram vývoje beroucí v úvahu studijní povinnosti zhotovitele a doporučení vedoucího práce. Plán počítal s podrobnou analýzou během volného času probíhající až do února 2021, po které měly následovat dva měsíce intenzivní implementace, dva týdny testování a tři týdny psaní textu práce. Tento plán maximalizoval množství času na zpracování diplomové práce a byl dostatečně flexibilní, aby zmírňoval nečekané událost a problémy během vývoje. Není tedy pravda, že by špatný technický stav alfa verze byl důsledek absence časového plánu nebo jeho nerealističnost.

Analýza pokračovala během září a října. Došlo k sepsání specifikace nové aplikace (Sekce 4), analýzy existujících grafických knihoven (Sekce 5), prozkoumání a analýzy normy ISO (Sekce 2.3) a také analýzy Flowgorithmu a podobných již existujících nástrojů (Sekce 3). Analýza byla komunikována s vedoucím práce nedlouho po nasdílení harmonogramu a 18. 10. 2020, u příležitosti vytvoření gitu projektu, poslána na git.

7.3 Vývoj interpretu

V únoru začala práce na tvorbě interpretu, tedy logické částí aplikace obsahující datové typy, uzly a systém interpretace. Tato část byla, ač ještě bez parseru výrazů, dokončena před třetí schůzkou s vyučujícími z katedry strojní, kde byla prezentována a ke které došlo 12. 3. 2021.

Během třetí schůzky[4] byly také dořešené implementační detaily týkající se interpretace, zvláště použití výrazů, konverze datových typů, použití prvků break, continue a return, či nativních metod, které bude muset aplikace za základu podporovat (na základě diskuze se usoudilo, že budou nutné jen metody C# třídy *Math*). Zároveň bylo dohodnuto testovat aplikaci v květnu v rámci předmětů UUR prostřednictvím vedoucího práce nebo PPA2 na základě domluvy s garantem předmětu.

Práce ovšem byly přerušeny ve prospěch dokončení studijních povinností, jejichž konečný termín byl bližší. Tyto studijní povinnosti byly zahrnuty v plánu a bylo i počítáno s mírným překročením časové dotace určené dle osnov.

[4]K dispozici v souboru *Vysledky/Záznamy/zaznam3.txt*

Parser výrazu byl dokončen na konci dubna, kdy také došlo k požádání o pozdější termín odevzdání diplomové práce, k čemuž bylo vyhověno.

7.4 Alfa - verze 0.1-0.2

V rámci května probíhala práce na vizualizaci vývojových diagramů pomocí knihovny *MSAGL* - právě v této fázi došlo k největšímu zaseknutí. Problémy, ke kterým s knihovnou *MSAGL* docházelo, jsou zkráceně popsány v Sekci 5.9. Na zprovoznění vizualizace vývojových diagramů s *MSAGL* bylo v květnu využito přinejmenším 120 člověkohodin a ani poté nebyla situace ani zdaleka uspokojující.

Dne 26. 5. 2021 byla vytvořena částečně funkční verze bez GUI a s nutností po každé úpravě grafu znovu manuálně přiblížit a přiscrollovat na původní místo. GUI, velmi minimalistické, ale technicky funkční, bylo vytvořeno během následujícího týdne. Další týden byl věnován opravě *MSAGL*, aby nebylo nutné po každé úpravě diagramu ručně vracet přiblížení a pozici do původního stavu, se smíšenými výsledky.

12. 6. 2021 byla hotová první funkční verze (označení 0.1) a práce přešla na psaní textu práce. Nedokončené části práce a části s velkým množstvím bugů (podprogramy, pole, for-cyklus + kopírování a mazání uzlů) byly osekány, čímž vznikla verze 0.2 aplikace. 21. a 22. 6. proběhlo testování s předem domluvenými dobrovolníky, které ukázalo na nestabilitu a vizuální nepřívětivost aplikace. Nalezené bugy, u kterých to z časových důvodů bylo možné, byly 21. 6. opraveny. 23. 6. byl dopsán text práce, 24. 6. byl dokončen poster a práce byla v nejzazší den odevzdána.

7.4.1 Testování verze 0.2

Účastníkům testování byl dodán .rar soubor obsahující přenositelné verze Flowgorithmu i nové aplikace a formulář k vyplnění s pokyny a instrukcemi pro syntaxi výrazů, ovšem ne ovládání aplikací. Formulář je možné najít v souboru *Vysledky/Odpovědi Testování/Formulář Alfa.pdf.pdf*. Přesné odpovědi účastníků testování jsou v souboru *Vysledky/Odpovědi Testování/-Testování Alfa.pdf*.

Tohoto testování se zúčastnili pouze 4 zájemci. Navzdory očekávání žádný z účastníků testování nepoužil formát dotazníku. Dva odpisovali textově přes aplikaci *Discord*, jeden odpověděl pouze slovy a jeden napsal první polovinu poznámek na konec formuláře a druhou polovinu řekl osobně slovy. K jednotlivým účastníkům:

Účastník M1

Účastník je programátor z povolání, který prošel systematikou výukou programování na střední i vysoké škole. Vývojové diagramy běžně nepoužívá a s žádným nástrojem typu Flowgorithmu před testováním nepřišel do styku.

Účastník se negativně vyjádřil ke stabilitě aplikace, negativně hodnotil nemožnost vytvářet v této verzi aplikace pole, chybějící barevné zvýraznění hran, na které lze dát uzel, a některé další drobnosti.

Všechny tyto nedostatky se podařilo opravit až ve verzi 0.4 (Sekce 7.5).

Účastník J1

Účastník je programátor z povolání, který prošel systematikou výukou programování na vysoké škole. Vývojové diagramy běžně nepoužívá a s žádným nástrojem typu Flowgorithmu před testováním nepřišel do styku.

Účastník reagoval dysfemisticky na graficky negativní stránku aplikace - celkový vzhled, chyběly mu obrázky ve výběru uzlů a vadilo mu několik problémů vzniklých používání knihovny *MSAGL*. Taktéž negativně hodnotil stabilitu a chybné vyhodnocení syntaxe výrazu[5].

Účastníkovi citelně chyběl tooltip, předzaškrtnuté přidávání nové řádky na konec a možnost přidat nový řádek v promptu vstupu.

Možnost přidat nový řádek byla dodána ten samý den. Zbytek problémů bylo možné odstranit až s nahrazením knihovny *MSAGL*. Zpětná vazba přispěla k tomu, že bylo rozhodnuto v rámci vlastního vizualizéru neimplementovat funkcionalitu přiblížení.

Účastník M2

Účastník je programátor z povolání, který prošel systematikou výukou programování na vysoké škole. Vývojové diagramy běžně nepoužívá a s aplikací typu Flowgorithm přišel před testováním do styku, ale aktivně ji nepoužíval.

[5]Problém spočíval v tom, že gramatika neobsahovala ukončovací znak, a *Antlr* tak vyhodnocoval jen počáteční korektní část výrazu. Problém se podařilo najít a opravit až pro verzi 1.0 aplikace.

Účastník se vyjádřil hlavně k funkcionalitě aplikace, problémy se stabilitou a grafickou stránkou zhodnotil stroze. Vadila mu nemožnost použít $\backslash n$ v rámci výrazu pro odřádkování[6], přegenerování grafu po některých akcích a že postfixové unární operace dekrementace a inkrementace (i++) se chovají identicky se svou prefixovou verzí[7].

Účastník také negativně hodnotil nemožnost mazat uzly a nutnost ručně přetypovat datové typy. Pozitivně se ovšem vyjádřil o přítomnosti unární inkrementace a možnosti kolabovat uzly. Způsob spojování textových řetězců mu nevyhovuje, ale považuje ho za smysluplnější v porovnání s Flowgorithmem. Zásadní problém vidí v konzolovém vstupu - netušil totiž, kam jej zapsat.

Většina poznámek tohoto účastníka byla příliš specifická a příliš technického charakteru. Nutnost přetypování ve verzi 1.0 odstraňuje přepínač *Implicit Conversion*, byl ovšem implementován pro kompatibilitu s Flowgorithmem. Konzole je od verze 1.1 zešedlá a nelze do ní psát, pokud se nečeká na vstup.

Účastník A1

Účastník nikdy neprošel systematickou výukou programování. Má za sebou osm let praxe jako skladník/prodavač ve společnosti Jysk, vývojové diagramy mu nic neříkají a s aplikací typu Flowgorithm před testováním nepřišel do styku.

Účastníkovi se ze začátku podařilo paradoxně lépe ovládat novou aplikaci, protože měla větší šipku, na kterou tak bylo při náhodném klikání větší šance kliknout. Při práci dále pokračoval s dokumentací Flowgorithmu, protože pro něj bylo obtížné pochopit, jak fungují výrazy v obou aplikacích. Pozitivně hodnotil, že nová aplikace umožňuje při deklaraci proměnnou inicializovat, negativně pak orientaci v její GUI a nemožnost mazat uzly.

[6]S tím se původně počítalo, ale tato funkcionalita se později ukázala jako zbytečná pro programátora-začátečníka. Nový řádek je možné ve výrazu přidat enterem.

[7]Toto chování stále nebylo změněno. Tato změna je v rámci knihovny Antlr a současné implementace netriviální, žádný jiný účastník v rámci všech testování tuto mechaniku nechtěl použít a běžný programátor často nezná funkcionální rozdíl mezi těmito operacemi. Nejedná se tak o nutnou funkcionalitu.

7.5 Vlastní vizualizátor - verze 0.3-0.4

Vzhledem k velkému množství problémů s knihovnou *MSAGL* bylo rozhodnuto tuto knihovnu nahradit vlastním systémem pro vizualizaci vývojových diagramů. *MSAGL* byl kompletně odstraněn 12. 9. 2021 a 22. 9. 2021 byla vytvořena první spustitelná verze vizualizátoru schopného vizualizovat prázdný diagram. 23. 9. již mohl vizualizovat všechny tvary a podmínku, 24. 9. mohl vizualizovat běžnou smyčku, 26. 9. kolabování uzlů a 30. 9. dokonce i forcyklus (verze 0.3).

Během října se vývoj soustředil hlavně na odstranění bugů, údržbu kódu a implementaci načítání a ukládání vývojového diagramu ve formátu .fprg. V listopadu se pokračovalo podobným způsobem, bylo přidáno načítání a ukládání v interním formátu .fcha, který na rozdíl od .fprg zvládne ukládat aktuální stav jednotlivých nodů, multidimenzionální pole a obecný forcyklus.

V tomto stavu (verze 0.4) měla být aplikace zhodnocena v rámci čtvrté schůze s vyučujícími z fakulty strojní, k té ovšem nedošlo kvůli postupnému onemocnění účastníků a jejich členů rodiny virem COVID-19. Schůze byla odročena postupně až na konec ledna. K testování tak v této fázi nedošlo.

7.6 Beta - verze 0.5-0.6

V polovině listopadu začala práce na analýze GUI - nového a korektnějšího způsobu jeho implementace a prozkoumání správného užití MVVM struktury. 21. 12. začala práce na implementaci a byly vytvořeny *properties* (model). 22. 12. byl implementován *ViewModel* pro editor uzlů a přidány tooltipy a názvy pro jednotlivé uzly. 23. 12. byl unifikován výpis erroru v rámci editoru uzlů a přidána hlavička editoru s obrázkem uzlu, který byl později použit i v ostatních oknech. 29. 12. byla dokončena úprava prezenční vrstvy editoru uzlů, aby podporovala MVVM, byla stabilnější a graficky přehlednější. 30. 12. došlo k unifikaci všech částí aplikace na *.NET 5.0*, což vedlo ke zpomalení překladů aplikace a zvětšení její velikosti, ale pomohlo vyřešit problémy s nestabilitou některých částí a teoreticky by také mělo umožnit s emulátorem Wine kompatibilitu s unixovými operačními systémy[8].

[8]Testování ukazuje, že to zatím není možné. Viz Sekce 7.11.1.

31. 12. bylo přepracováno GUI výběru uzlů do MVVM. 3. 1. 2022 bylo do MVVM přepracováno i hlavní okno a celá aplikace tak od toho dne podporuje MVVM. 4. 1. Byla dokončena první použitelná verze správy metod a editoru metody. 6. 1. byl implementován systém pro záložky otevřených metod, který umožnil mít více metod otevřených najednou a rychle mezi nimi přepínat. 9. 1. byl dokončen debugovací režim a watcher proměnných.

V této verzi byla aplikace testována od 10. 1. vedoucím práce a vyučujícími z katedry strojní (viz Sekce 7.6.1), ze zpětné vazby bylo jasné, že aplikace není pro běžné užití dostatečně stabilní, je nutné odstranit běžně se vyskytující bugy, a že aplikace kvůli rozdílné syntaxi nedokáže korektně interpretovat všechny vývojové diagramy vytvořené ve Flowgorithmu.

7.6.1 Testování verze 0.6

Vedoucí práce

Celá zpětná vazba je k dispozici v souboru *Vysledky/Odpovědi Testování/-TestBeta1.txt*.

Vedoucí práce důkladně prozkoumal aplikaci hlavně z hlediska funkcionality. Kritizoval absenci tlačítka pro vytvoření nového prázdného programu a scrollbaru konzole, špatnou detekci dotyku myši[9] a rozvržení oken aplikace a nekonzistenci chybových hlášek.

Také mu chybělo vysvětlení funkce defaultní hodnoty u pole, přesné použití vypsání hodnoty pole v uzlu výstupu a editor for-cyklu mu přišel vyloženě neintuitivní - doporučil jej simplifikovat. Znovu upozornil na problém už z verze 0.2, kdy výraz akceptuje nevalidní výraz proto, že přijme jen jeho validní začátek. Problematická pro jen byla absence chybového výpisu při pádu. Narazil také na překrývání nápisů při některých nastaveních uzlu podmínky.

Vedoucí práce v této fázi vývoje doporučil soustředit se hlavně na stabilitu aplikace a lepší řešení výpisu chybových stavů aplikace. Autor aplikace se doporučením řídil. Všechny zmíněné části byly postupně opraveny.

[9]Změna byla plánovaná od verze 0.3

Vyučující z fakulty strojní

Celá zpětná vazba je k dispozici v souboru *Vysledky/Odpovědi Testování/-TestBeta2.txt*.

Vyučující měli s použitím aplikace výrazné problémy pramenící z toho, že se automaticky snažili používat aplikaci stejně jako Flowgorithm, i když Flowcha-n požaduje explicitní konverzi datových typů. Vyšlo také najevo, že velkou část svých výukových materiálů nemůžou použít, přestože nová aplikace dokáže soubory formátu .fprg otevřít, protože Flowgorithm používá mírně odlišné názvy a syntaxi výrazů a navíc provádí implicitní konverzi datových typů. Tento problém byl vyřešen ve verzi 0.8 aplikace.

7.7 Beta - verze 0.7

Snaha se následně přesunula na odstranění bugů, zvýšení stability a management chyb. 30. 1. byl přidán *ErrorManager*, který zajišťuje, že chyba je uživateli vypsána do konzole. 1. 2. byl implementován vlastní systém hit testů, čímž byl vyřešen problém s příliš malým bounding boxem hran a průhlednými částmi uzlů.

4. 2. 2022 došlo ke čtvrté schůzce, kde byla představena již výrazně stabilnější verze. Kritika byla přesunuta na grafickou stránku aplikace, zvláště použití příliš kontrastních barev, rozložení GUI a případně některé grafické bugy (povětšinou nápisy zasahující do jiných grafických objektů). Vyučující se rozhodli odmítnout použít aplikaci v rámci aktuálního akademického roku ve výuce, protože by už nestihli přepracovat výukové materiály tak, aby odpovídaly nové aplikaci, a také proto, že aplikace v aktuálním stavu nebyla dle jejich názoru pro studenty dostatečně atraktivní.

Ze čtvrté schůze nebyl proveden záznam, ani výpisky. Schůzi doprovázaly technické problémy a jeden z účastníků byl nemocný.

7.8 Plná verze - 0.8-0.9

10. 2. byla přidána kompletní podpora pro metody používané aplikací Flowgorithm a režim *Implicit Conversion*, díky čemuž se stala aplikace pro formát .fprg plně kompatibilní. 15. 2. byl přidán plnohodnotný uzel pro komentář (verze 0.8). 16. 2. došlo ke schůzi s vedoucím práce, který je specialistou na GUI, a dodal tak expertní doporučení ohledně jeho přepracování[10]. Tato

[10]Z této, i následující schůze byly vytvořeny jen těsnopisné poznámky zaměřené na velké množství konkrétních problémů a bugů, na které vedoucí práce upozornil.

schůze byla opakována 15. 3. (verze 0.9) a na jeho základě vznikl 20. 3. návrh testovacího scénáře pro studenty UUR. Zbytek února byl zaměřený hlavně na odstranění bugů. Během března došlo k úpravě grafické stránky GUI, změně barev, přidání obrázků, zlepšení rozložení a velikostí a interaktivnosti grafických objektů.

7.9 Plná verze - 1.0

30. 3. byl změněn systém terminálů (začátku a konce vývojového diagramu) tak, aby byly terminály generovány pouze dynamicky a nikdy nebyly ukládány. Také došlo k vylepšení sloupců objektu *ListView*, aby bylo možné podle nich seřadit řádky a aby byl *ListView* rozložený na celou šířku dle velikosti sloupců. Ve stejný den byl také studentům online zpřístupněn testovací scénář s aplikací v aktuální stavu (verze 1.0). 31. 3. byl kompletně předělán systém templatů tak, aby zároveň odstranil redundanci a zároveň umožnil customizaci všech datových verzí editoru uzlů.

7.9.1 Testování verze 1.0

Testování probíhalo podle testovacího scénáře, který byl studentům zpřístupněn online[11]. Scénář má dvě hlavní částí - implementaci algoritmu výpočtu Fibonacciho čísla a debugování uměle rozbité verze Insertion Sortu - účastníci měli možnost vyplnit jednu z nich, nebo i obě. Účastníkům úmyslně nebyly dány žádné instrukce ani manuál pro ovládání aplikace, pokud si o ně výslovně nepožádali.

Dle dohody byli o vyplnění testovacího scénáře požádáni studenti předmětu UUR - Úvod do uživatelského rozhraní - vedoucím práce, který je zároveň garantem předmětu. Vzhledem k tomu, že předmět je zaměřený právě na tvorbu GUI, nabízel testovací scénář studentům možnost procvičit se v hledání slabých stránek existujícího GUI a případného možného zlepšení. Testovací scénář tak zároveň posloužil i jako výukový materiál a studenti měli možnost jeho vyplněním získat body v rámci předmětu UUR.

[11] Aktuální verze k dispozici na *https://forms.gle/wkz6FB9PEKtJAdGe6*

Studenti byli poprvé informováni o testovacím scénáři 4. 4. a 7. 4. byli vedoucím práce připomenuti. První vyplnění bylo zaznamenáno až 11. 4. V reakci na nízkou rychlost vyplňování v rámci studentů UUR autor aplikace 13. 4. sám požádal studenty prvního a druhého ročníku přes aplikaci *Discord* o vyplnění scénáře a případné informování, pokud jim ve vyplnění scénáře něco brání, je to příliš dlouhé a obtížné nebo nechtějí scénář vyplnit z jiného důvodu. Ten samý den byly registrovány dvě další vyplnění.

Ze zpětné vazby vychází, že některým studentům vadilo, že se k online dotazníku musí připojit přes Google účet. To je nutné, protože dotazník jinak neumožňuje ukládat respondentům soubory, jako jsou screenshoty nebo uložené vývojové diagramy. Každý student ovšem takový účet má[12], není tedy nutné jej vytvářet.

Několik studentů také reagovalo tím, že jim aplikaci nejde spustit, přestože dle instrukcí nainstalovali v instrukcích specifikovanou verzi .NET. Ukázalo se, že problém tkvěl v nainstalování špatného druhu - studenti z neznámého důvodu (snad rozložení dané webové stránky?) stahovali 32bitovou verzi, která ovšem funguje pouze na 32bitovém operačním systému. Všichni zájemci ovšem zatím měli 64bitové systémy.

Velká část studentů upozornila na zmíněné dva problémy a už se k dotazníku nevrátila. Dokonce i po soukromém vysvětlení těchto záležitostí. Testování verze 1.0 tak má pouze 3 účastníky.

Účastník A2

Účastník je studentem bakalářského studia na FAV, již se setkal s vývojovými diagramy i podobnou aplikací typu Flowgorithm, ale aktivně je nepoužíval. Účastník znal autora práce, neboť opakovaně využíval jeho pomoci v rámci Student Support Centre, mohl tak být více motivovaný se testování zúčastnit a aplikaci kladně hodnotit.

Účastník vyzkoušel pouze debugovací část, kterou nezvládl celou dokončit. Účastník měl problém jednak s tím, že nedokázal identifikovat, jestli program běží v nekonečné smyčce, nebo ne, jednak s tím, že se snažil odstranit řídící uzel for-cyklu, což mu aplikace nepovolila a upozornila jej na problém zdánlivě nesouvisející chybovou hláškou.

Účastník by upřednostnil možnost místo kopírování posouvat myší již existující uzly na jiné místo a kritizoval vertikální neskladnost vývojového diagramu oproti běžnému kódu.

[12]Orion účet je propojen přes Gapps, viz zde: *https://tinyurl.com/25eka2hd*

Nalezené problémy byly obratem opraveny, řídící uzly již není možné mazat a byly přidány další podpůrné identifikátory běhu aplikace - výpis při začátku programu a změna tvaru start/stop tlačítka. Vertikální neskladnost je problém očekávaný už od prvního návrhu a částečně jej řeší možnost kolabovat uzly a rozdělit program na více podprogramů. Přidání Drag'n'Drop čistě pro přesun uzlů dává z uživatelského hlediska smysl, jednal by se ovšem o nečekaný zásah do funkcionality aplikace, který je netriviální implementovat.

Účastník A3

Účastník je studentem bakalářského studia na FAV, již se setkal s vývojovými diagramy, ale ne s podobnou aplikací. Účastník splnil obě části testovacího scénáře. Měl nereplikovatelné problémy s tím, že se při psaní neaktualizovala textová políčka a dle vyjádření jde možná o problém specifické grafické karty nebo jejího nastavení. Ze začátku také nevěděl, jak aplikaci ovládat, a upřednostnil by přítomnost nápovědy, i bez ní ovšem oba úkoly splnil. Ocenil by klávesové zkratky, které byly poté také přidány.

Účastník měl na aplikaci jako na výuková nástroj pozitivní názor, preferuje jej ovšem spíš pro vizualizaci algoritmů než jako programovací jazyk. Vyzdvihuje přehlednost, ale negativně hodnotí, že je práce s aplikací pomalejší než psaní kódu.

Účastník J2

Účastník je studentem bakalářského studia na FAV, již se setkal s vývojovými diagramy, ale ne s podobnou aplikací.

Účastník splnil pouze debugovací úlohu. Během debugování měl problém s tím, že byl watcher proměnných překrýván, chybně si myslel, že může přímo smazat samostatně jeden z řídících uzlů for-cyklu (stejně jako účastník A2) a spuštěná interpretace mu zůstala spuštěná i po načtení jiného programu. Všechny tyto problémy byly později vyřešeny - Watcher je nyní vždy nejsvrchnější okno, uživatel fyzicky nemůže odstranit uzel for-cyklu mimo jeho editor (možnost je zašedlá) a při inicializaci programu se provede kontrolní ukončení interpretace.

Účastníkovi se aplikace líbí, ale priorizoval by výuku spíše v Java nebo C, navíc by osobně doporučil umístit Watcher proměnných do společného okna s diagramem. Paradoxně si myslí, že by v aplikaci mohl psát programy rychleji než v jazyce Java, v rozporu s názorem většiny ostatních účastníků.

7.10 Plná verze - 1.1

Dne 1. 4. byly přidány obrázky do editoru uzlu smyčky ukazující nastavený typ uzlu a interaktivní tlačítka, která se označí po přejetí myší. 2. 4. byly přidány interaktivní uzly iterace a inicializace přímo do editoru for-cyklu. 10. 4. byla změněna většina stylů běžných objektů, aby byla více atraktivní a interaktivní. 13. 4. byly opravena část bugů reportovaná účastníky testování a aplikace byla aktualizována (verze 1.1).

7.10.1 Testování verze 1.1

Testovací scénář byl upraven tak, aby obsahoval novou verzi aplikace a také aby informoval o tom, že je nutné stáhnout 64bitovou verzi .NET. Jinak v této fázi nedošlo k výrazné změně.

V této fázi se zúčastnili dva uchazeči, nejednalo se ale bohužel o studenty, ale o už zkušené programátory. Celý vyplněný dotazník spolu s odpovědmi ze Sekce 7.9.1 jsou v souboru *Vysledky/Odpovědi Testování/Testování aplikace Flowcha-n.csv*[13].

Účastník R1

Účastník má za sebou vysokou školu v oboru a několik let praxe, vývojové diagramy zná a používá, ale o podobné aplikaci zatím jen slyšel. Účastník splnil úkol na implementaci a zkusil také debugovací úkol, který ale nedokončil.

Účastník má problém s přesnou formou výrazu a chybovým upozorněním na jeho syntaxi, zejména má problém s nedostatečným vysvětlením explicitní konverze. Problém mu dělá také konzole, která momentálně korektně akceptuje pouze vstup z konce nebo začátku položky - ideální by bylo vytvářet zvláštní konzole čistě jenom pro vstup uživatele, je ovšem otázka, kam ji umístit.

Účastník má také stejně jako účastník A2 problém zjistit, jestli aplikace běží, nebo ne, přestože v této verzi již spuštění aplikace mělo být patrné z textového indikátoru stavu, z konzole i z podoby tlačítka start/pause - účastník možná měl staženou starší verzi, protože záznam byl odeslán jen den po aktualizaci. Také by doporučil přesouvat uzly přes Drag'n'Drop stejně jako účastník A2 a kritizuje problematickou práci s podgrafem, kde není jasné, která část je interaktivní.

[13]Osobní informace všech účastníků byly vymazány. Nebudou nikde použity ani šířeny, s výjimkou pro určení studentů předmětu UUR, kteří mají nárok na bonusové body.

Účastník pozitivně hodnotí, že není pro psaní programu v aplikaci nutné znát žádnou syntaxi, ale stejně jako většina účastníků považuje práci s ním za zdlouhavou. Účastník je také jediný, kterému se podařilo vyvolat pád aplikace, a to velmi rychlým opakováním spouštění interpretace, při kterém se interpretace nestihla dost rychle ukončit - jednalo se o kritickou sekci, kterou si autor aplikace původně neuvědomil.

Účastník také upozorňuje, že při vyplnění celého dotazníku se odpovídá na stejné otázky dvakrát v reakci na každý úkol zvlášť, což považuje za duplicitu. To bylo změněno při testování verze 1.2 aplikace. Dále má opakovaný problém s překrýváním watcheru, zvýraznění vybraného uzlu považuje za nevýrazné (později bylo změněno) a chybí mu možnost psát do watcheru proměnných vlastní interpretovatelné výrazy jako *pole[i]*, možnost přidat klíč generátoru náhodných čísel a možnost vypsat stav proměnných do logu. - Poslední tři možnosti zatím nebyly implementovány a nejspíš nejsou pro běžného uživatele podstatné.

Účastník F1

Účastník má za sebou vysokou školu v oboru a několik let praxe, vývojové diagramy zná a používá, a dokonce už používal i aplikaci podobného typu. Účastník splnil pouze úkol na implementaci.

Upozornil opět na nedlouho poté vyřešenou chybu s odstraněním řídícího uzlu podgrafu a také na další chybu, u které účastník ale bohužel nenahrál program ve stavu, v jakém by ji bylo možné replikovat. Navíc evidentně také použil ještě verzi 1.0, protože z obrázků je vidět, že po spuštění není tlačítko start změněno na tlačítko pause.

Účastník nemá rád aplikace tohoto typu, považuje je za svazující a hůře pochopitelné, a tedy si nemyslí, že by byly vhodné pro výuku. Tuto aplikaci specificky považuje za těžkopádnou, chybí mu návod a nelíbí se mu rozdělení aplikace do různých oken - preferoval by otvírání postranních panelů ve stejném okně.

7.11 Plná verze - 1.2

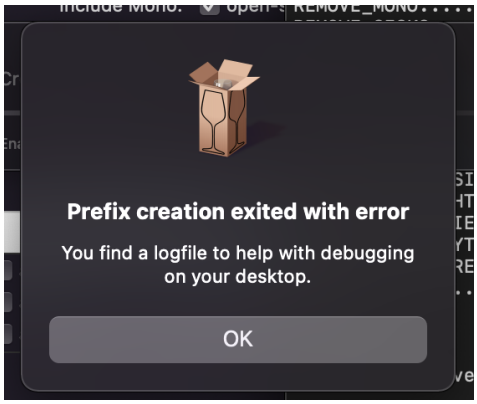
Od 15. 4. se při zastavení na určitém uzlu (error nebo pause) k tomuto uzlu přiscrolluje. 17. 4. byla přidána možnost přeložit diagram do kódu C# programu. 23. 4. byl zaveden systém pro automatické uložení při spuštění a nucené vybrání souboru pro uložení diagramu před prvním spuštěním, také byly přidány klávesové zkratky. Ve stejný den byl také upraven testovací scénář pro studenty, aby byl kratší a méně náročný (verze 1.2).

Od 24. 4. začala práce na přepracování textu práce. 29. 4. aplikaci vyučující z fakulty strojní pozitivně ohodnotili. Vývoj se už pak soustředil jen na odstranění bugů, kvalitu kódu a drobné funkcionality zdůrazněné účastníky testování. 12. 5. byla přidána ikona. 13. 5. se podařilo získat účastníky alfa testování, aby aplikaci retestovali. 15. 5. byla přidána lokalizace do češtiny a aplikace byla dokončena s s verzí 1.3.

7.11.1 Testování verze 1.2

Dotazník nebyl vytvořený nový, ale byl pouze upravený starý dotazník tak, že jej část byla přesunuta do samostatné sekce z obou částí a část vypuštěna. Debugovací úloha, kde studenty mátl uměle vyvolaný nekonečný cyklus, byla upravena a tento úmyslný bug byl vypuštěn. K aplikaci byl navíc připojen instalační soubor .NET (jehož šíření licence umožňuje) specifický pro většinu běžných operačních systémů a procesorů, takže v tomto kroku dále studenti nemohli udělat chybu. 23. 4. byli studenti opět kontaktováni přes aplikaci *Discord*, tentokrát už ale bohužel bez reakcí.

Autor aplikace se dále pokusil osobně požádat o vyplnění testovacího scénáře studenty, kterým během posledních dvou semestrů výrazně pomohl v rámci *Student Support Centre*, většina studentů se ale bohužel neozvala. Jeden student dotazník vyplnil, ale aplikaci dále příliš nezkoušel (jen přidání, odstranění a editaci uzlů) a pozitivně zhodnotil, že si může díky vývojovým diagramům lépe představit, co má program vykonat. Další dva studenti se pokoušeli aplikaci zprovoznit na systému *MacOS* pod emulátorem *Wine*, jak je ale vidět na Obrázku 7.1, nebyli příliš úspěšní.



Obrázek 7.1: Studenti snažící se spustit aplikaci s emulátorem *Wine* skončili s touto chybovou hláškou.

Dalšími účastníky tak nebyli studenti, ale účastníci původního testování a další různí dobrovolníci. Až na účastníka M2 a S1 byli všichni kontaktováni osobně a svou odpověď vyjádřili přes *Discord* nebo jinou platformu[14]. Odpovědi je možné vidět v souboru *Vysledky/Odpovědi Testování/Testování aplikace Flowcha-n2.csv*. Soubor obsahuje i původní výsledky.

Účastník M2

Účastník se zúčastnil testování Alfa verze a mohl tak pozitivně zhodnotit pokrok aplikace.

Negativně hodnotil práci s konzolí, u které doporučil načítat vstup z nového okna sloužícího pouze pro vstup, což je stále analyzováno. Dále mu vadila nutnost použít breakpoint pro zastavení na chybě a problémy vzniklé kopírováním právě debugovaného uzlu - oba problémy byly vyřešeny.

Doporučil by přidat aplikaci ikonu (ten samý den byla přidána), přidat našeptávání proměnných a grafické znázornění textové položky, ve které je chyba (bylo přidáno obratem).

Pozitivně se vyjádřil hlavně k funkcionalitě *Implicit Conversion*, možnosti upravovat graf během debugování a dokončeným funkcionalitám z minula.

Účastník J1

Účastník se zúčastnil testování Alfa verze a mohl tak pozitivně zhodnotit pokrok aplikace.

Upřednostnil by, pokud by změna řídící proměnné for-cyklu vedla na automatickou změnu i v ostatních uzlech, pokud by při vnoření for-cyklu byl název řídící proměnné automaticky odlišný od proměnné vnějšího cyklu a pokud by program po skončení interpretace zároveň vypsal, jak dlouho program běžel (poslední bod byl ve verzi 1.3 implementován).

Účastník měl zároveň problém najít Break (který sdílí uzel s Return) a byl překvapený, že nemůže žádným způsobem vykreslovat 2D grafiku (patrně v něm 2D pole vyvolává očekávání jeho vykreslení). Považuje při tom za špatné designové rozhodnutí umožnit uživateli všechna okna roztahovat dle potřeby - doporučuje je raději zamknout na určitou velikost (což byla varianta během vývoje diskutovaná a zavrhnutá, protože stejně bylo třeba umožnit velikost komponent nastavit pro potřeby různých oken a aplikace by mohla na různých rozlišeních při jedné velikosti působit nepatřičně).

[14]Jejich odpovědi byly pro konzistenci do dotazníku uměle přidány.

Účastník M1

Účastník se zúčastnil testování Alfa verze a mohl tak zhodnotit vývoj aplikace. Pro jeho potřeby ovšem byla aplikace vývojem ovlivněna negativně.

Účastník přiznal, že není zběhlý v angličtině, a proto pro něj byly popisky aplikace a tooltipy nesrozumitelné. Jedná se o jediného z účastníků, který tento problém poznamenal. Původně, i na základě reakcí od kolegů z fakulty strojní, se předpokládalo, že bude stačit do začátku jednojazyčná verze aplikace. Na základě této zpětné vazby byla ve verzi 1.3 přidána čeština.

Účastník uvedl, že se mu s aplikací pracuje hůře než s Flowgorithmem, kterému rozumí a na rozdíl od této aplikace má lištu navozující dojem windowsovského ribbonu. To mělo efekt na i zcela nesouvisející úkony, jako je přidání uzlu nebo otevření jeho editace. Aplikaci tak účastník hodnotil jako horší než programovat v běžném programovacím jazyce, protože stejně je třeba znát jeho příkazy.

Účastníkovi se podařilo najít tři drobné bugy, které byly obratem opraveny. Zároveň měl problém s ukládáním souboru na disku, kde nemá plná práva, což způsobilo, že ze začátku ani účastník nemohl vytvořený program interpretovat. Účastník by ocenil přidat sémantickou kontrolu, funkci Undo a již zmíněnou češtinu.

Účastníci A1, G1 a F2

Tito účastníci nemají žádné nebo jen minimální zkušenosti s programováním. Účastník A1 se zúčastnil již testování alfa verze a při té příležitosti dokázal naprogramovat výpočet Fibonacciho posloupnosti v aplikaci Flowgorithm. Účastník F2 je inženýr, ovšem v nesouvisejícím oboru a s programováním se sice dostal do styku, ovšem uvedl, že jej nemá v oblibě.

Účastník F2 si vyžádal uživatelskou příručku a podle ní dokázal vytvořit program, který načte číslo od uživatele a pak jej vypíše. Stěžoval si při tom, že se mu všechno označuje červeně jako špatně, aniž by věděl proč[15]. Celá trojice účastníků se shodla na tom, že aplikace je pro běžné lidi příliš složitá na pochopení a že je lepší ji cílit na osoby, které mají základní povědomí o algoritmizaci.

[15]Většina uzlů je při vytvoření prázdná a jsou po vytvoření nastavené jako zakomentované. Při uložení uzlu dochází k jeho odkomentování a tedy se poté jeví, jako že se v něm nově objevila chyba.

Účastník A1 byl méně úspěšný než minulý rok s Flowgorithmem. Minulý rok ovšem vyvíjel podle tutoriálů Flowgorithmu a měl upřímný zájem se naučit programovat, tento rok aplikaci testoval z časových důvodů večer před spaním a bez žádných rad nebo návodů. Nejde tedy jednoznačně říct, že by Flowgorithm byl pro účastníka A1 v tomto ohledu lepší než nová aplikace.

Účastník S1

Účastník S1 je programátor webových aplikací s pětiletou praxí bez vysokoškolského vzdělání.

Pro účastníka byla práce s aplikací náročná - trvalo mu dvě hodiny napsat Fibonacciho posloupnost a zdebugovat Insertion Sort, debug byl ovšem i tak proveden špatně s nově zanesenou chybou. Pro práci si vyžádal uživatelskou příručku a později také nejnovější verzi aplikace, když narazil na chybu se zavíráním Watcheru.

Účastník by ocenil funkci Undo a doporučil ukazovat vývojový diagram a odpovídající generovaný kód programovacího jazyka vedle sebe pro navození lepšího pochopení. Také si myslí, že uživatel, který se dobře nevyzná ve vývojových diagramech a programování, potřebuje lepší návod pro pochopení základní funkčnosti aplikace.

7.12 Shrnutí vývoje

Aplikace si prošla dlouhým vývojem, během kterého byla opakovaně konzultována už od fáze konceptu. Bohužel nejnovější testování ukazuje, že některé souvislosti a funkcionality nebyly buď správně pochopeny anebo prozkoumány. Mnohé problémy, které evidentně musí být platné i pro vzorový Flowgorithm, tak byly nalezeny až při testování hotové nové aplikace.

Na základě zpětné vazby z testování se tak například zdá, že by aplikace byla pro použití výrazně jednodušší, pokud by povolovala Drag'n'Drop - například pro přesouvání uzlu. Drag'n'Drop byl ovšem již na základě druhé schůze zavrhnut jako zbytečná funkcionalita, která by studenta mohla zbytečně mást, nebylo tak s ní počítáno a struktura aplikace této funkcionalitě nebyla přizpůsobena.

O aplikaci bohužel v současné době nejde říci, že by ji mohl použít skutečně každý. Účastníci bez povědomí o vývojových diagramech měli výrazné problémy s prací s aplikací a hlavní menu aplikace se navíc ukazuje jako výrazně neintuitivní.

Tyto informace nebyly ze začátku vývoje patrné a dokreslují fakt, že software se zpravidla vyvíjí, nikoliv vyrábí, a že tedy i se sebelepším průzkumem a návrhem je třeba počítat se změnami aplikace během vývoje. U většiny částí se tak stalo a na základě zpětné vazby se zdá, že aplikaci lze použít pro účely výuky a mnozí testeři o ní mají pozitivní mínění. Aplikace je v současném stavu konkurenceschopná.

7.12.1 Porovnání s existujícími nástroji

Při vývoji fungoval jako vzor Flowgorithm - bylo snahou dosáhnout jeho vlastností a překonat je. V některých případech, jako je velké množství jazyků a layoutů, do kterých lze aplikaci přepnout, to bylo považováno za zbytečné. V důležitých ohledech, jako je vizuální a funkční stránka aplikace, už se jí ale v průměru podařilo Flowgorithmu vyrovnat při zároveň umožnění přidání nových funkcionalit.

Oproti tomu PS-Diagram nebyl z pohledu vyučujících z fakulty strojní považovaný za zajímavý, neboť mu chybělo velké množství funkcí, které Flowgorithm má, zvláště možnost vytvářet vlastní podprogramy a vizuální přívětivost (uzly PS-Diagramu jsou jen jednobarevné). PS-Diagram ovšem umožňuje Drag'n'Drop, což neumí ani nová aplikace, ani Flowgorithm, a místo různých oken umožňuje editovat uzly jednoduše v jiném panelu. V mnoha ohledech se tak například zdá, že účastník testování F1 má zkušenosti právě s PS-Diagramem, a že by jej osobně preferoval.

PS-Diagram má ale v první řadě tu nevýhodu, že jeho výrazy jsou závislé na interpretaci v rámci JavaScriptu a není tak možné je mimo to upravovat, což zhoršuje možnost dalšího rozšíření. Hlavním přínosem autora je vizualizátor vývojového diagramu, ne vizuální programovací jazyk, jako je tomu v případě Flowcha-n. Tento rozdíl je výrazný a lze jej dobře poznat na čase nutném k vytvoření překladače do jiného programovacího jazyka. K přidání podpory jazyka Java do PS-Diagramu došlo v rámci bakalářské práce jistého Jozefa Š. ze Slovenska [12], v rámci Flowcha-n ovšem pro přidání překladu vývojového diagramu do jazyka C# stačilo jediné odpoledne.

Proti Flowgorithmu a PS-Diagramu mluví také čas - oba jsou už dlouho hotové produkty, které za sebou mají 8, respektive 10 let od vydání. Flowcha-n má oproti tomu dva roky od prvních návrhů a stále má kam růst. Přesto již nyní může ostatním konkurovat.

7.12.2 Vývoj v číslech

Vývoj aplikace nebyl přímo zaznamenáván, je ale možné ho ovšem odhadnout nepřímo na základě vypracovaných dokumentů, textové komunikace, různých záznamů a gitu. Celý vývoj byl popsán v předchozích částech této celé kapitoly. Changelog gitu je možné najít v souboru *Vysledky/gitChangelog.txt*. Tabulka 7.1 ukazuje tento vývoj přehledně v číslech.

Měsíc [16]	Hod. [17]	Práce	Verze	Soub. [18]	LOC+ [19]	LOC- [20]
2 2020	24	Analýza	-	-	-	-
5 2020	30	Analýza	-	-	-	-
10 2020	48	Text	-	-	-	-
2 2021	60	Kód	0	?[21]	?	?
5 2021	60+	Kód	0.0	![22]	!	!
6 2021	144	Kód	0.1	56	1905	139
6 2021	15	Kód	0.2	61	2467	95
6 2021	70	Text	-	-	-	-
9 2021	125	Kód	0.3	93	3092	1505
11 2021	72	Kód	0.4	47	1481	339
12 2021	80	Kód	0.5	108	20675	2721
12 2021	40	MVVM	-	-	-	-
1 2022	72	Kód	0.6	112	3639	2162
2 2022	40	Kód	0.7	97	2017	868
2 2022	32	Kód	0.8	97	4807	231
3 2022	20	Kód	0.9	64	454	318
3 2022	40	Kód	1.0	147	2373	1639
4 2022	48	Kód	1.1	136	3177	1326
4 2022	24	Kód	1.2	61	1567	415
4-5 2022	120	Text	-	-	-	-
5 2022	36	Kód	1.3	291	9960	10839

Tabulka 7.1: Tabulka ukazující vývoj aplikace Flowcha-n v čase a číslech.

[16] Měsíc a rok, kdy došlo k danému vývoji

[17] Odhadovaný počet hodin na základě známých údajů.

[18] Počet dle gitu modifikovaných souborů.

[19] Počet dle gitu přidaných řádek (všech).

[20] Počet dle gitu odstraněných řádek (všech).

[21] Byl použit jiný git, na který už není přístup.

[22] Commit obsahuje celý MSAGL a jeho 500+ souborů. To by výrazně poškodilo statistiky, není do nich tedy zahrnut.

Celkem bylo od verze 0.0 do verze 1.3 změněno dle gitu 404 souborů, 41432 řádek bylo přidáno a 6415 odstraněno[23]. Dle odhadů bylo na vývoj použito více než 1200 hodin, z toho alespoň 332 hodin na analýzu a psaný textu práce a 868 hodin na psaní kódu.

Počet řádků kódu je dle konsenzu většiny programátorů jedna z nejhorších metrik produktivity programátora, protože v ní vynikají zvláště programátoři píšící velké množství redundantního kódu, který se velmi špatně udržuje. Během vývoje byla naopak snaha počet řádek minimalizovat a kromě jiného byl často používán i integrovaný jazyk LINQ[30].

Počet řádků kódu je ale bohužel jednu z mála lehce prokazatelných metrik produktivity. Obecně se uvádí, že programátor napíše 5-100 řádek (včetně komentářů) za hodinu. Pokud bychom brali v úvahu pro programátora velmi nadprůměrných 53 řádků/hodinu, 41000 řádků by odpovídalo 775 hodin, což je více, než je ve sledovaných fázích vývoje odhadováno.

Git ovšem počítá i komentáře, prázdné řádky a generovaný kód. *Visual Studio* u projektu uvádí 20 365 řádků čistého zdrojového kódu vytvořeného programátorem, který je použit při spuštění, což by při 30 řádcích čistého kódu za hodinu průměrného programátora odpovídalo 679 hodinám práce, což už je výrazně méně, než je odhadováno.

V rámci mírně zastaralého *The Mythical Man-Month*[15] se uvádí, že programátor napíše v průměru 10 logických řádků kódu za den. *Visual Studio* uvádí, že se aplikace skládá z 6281 IL (logických řádek sestavených z kompilovaných instrukcí). IL je aproximace LLOC založená na kompilovaném a optimalizovaném kódu. Použitý čas podle této metriky vychází na 628 pracovních dnů. V rámci .NET se většina vývojářů shoduje na tom, že v průměru jim vychází přibližně 10 LLOC za hodinu. To by odpovídalo 628 hodinám práce.

Všechny tyto metriky tak ukazují, že bylo při práci na aplikaci vynaloženo nadstandardní množství času.

[23]Nejedná se o sumu čísel v tabulce, ale o rozdíl těchto dvou verzí. V případě opakovaných úprav stejného řádku v rámci vývoje je proto řádek započítán pouze jednou. Změna formátu ve verzi 1.3 se tak neprojevila na tomto celkovém součtu.

8 Testování

8.1 Unit testy

Byly vytvořeny tři velké kategorie unit testů: Testy parseru výrazů, testy interpretu vývojových diagramů a integrační testy založené na interpretaci celých již existujících vývojových diagramů, které jsou načteny a interpretovány.

8.1.1 Unit testy parseru

Pro každou aritmetickou operaci (dělení, násobení, modulo, plus, mínus) a jejich složité kombinace byl vytvořen test pro každý základní datový typ (integer, real, char) a to jako pro proměnné, tak pro literály. String je dál stejným způsobem testován na konkatenaci. Dále byl testován operátor rovnosti na datovém typu integer.

8.1.2 Unit testy interpretu

V rámci unit testů interpretu byl vždy nejprve vytvořen vývojový diagram s korektním obsahem, a ten následně interpretován. Tímto způsobem se testují kombinace:

- deklarace, přiřazení a výstup
- deklarace, podmínka (obě větve) a výstup
- dvě deklarace, forcyklus, který sčítá číselnou řadu a výstup
- deklarace, přiřazení a výstup s voláním nativní metody (Sin, DivRem)

8.1.3 Integrační testy

Místo soustředění na velké množství specifických testů pro jednotlivé malé částí aplikace se testování aplikace soustředilo hlavně na integrační testy, které jsou z principu výrazně citlivější a mají zásadní výhodu ve způsobu tvorby. Ve chvíli, kdy aplikace dokázala načíst vývojový diagram, bylo možné testovací příklady vytvářet přímo v aplikaci a pouze je v rámci unit testů načítat.

Aplikace tak obsahuje 8 integračních testů zkoušející základní funkčnosti aplikace. Dále 13 komplexních testů využívající funkční programy ve formátu .fprg, které představují výukové materiály používané vyučujícími z fakulty strojní. Testovací program hledající prvočísla vytvořený jako příklad možné funkcionality a také testy překladu do programovacího jazyka C#, které C# kód také interpretují.

Početně těchto testů není mnoho, svou složitostí by ovšem měly být schopné najít i podstatně složitější případy chyb, než samostatné nebo krátké unit testy.

8.2 Testování GUI

Jsou nástroje, které by umožnily testovat GUI automaticky, například *Selenium*, jejich použití ovšem tkví v porovnání grafického výstupu aplikace se vzorem, nikoliv v posouzení kvality User Expirience. Problémy GUI aplikace navíc tkvěly hlavně ve špatném grafickém cítění autora a obtížného posouzení znělosti návrhu před možností si ho proklikat uživateli. Vytvoření interaktivní makety by navíc byla složitostí srovnatelná s implementací funkčního GUI aplikace, proto bylo efektivně možné vizuální návrh GUI testovat až po implementaci.

Manuální testování aplikace, včetně testován GUI, ať už testery nebo stakeholdery, probíhalo opakovaně v rámci celého vývoje aplikace a toto testování je popsáno v rámci Sekce 7.

9 Závěr

Podařilo se navrhnout, implementovat a otestovat aplikaci, která umožňuje vytvářet a následně interpretovat vývojové diagramy jako program. Aplikace je plně funkční a připravena pro použití. Stále zůstávají možnosti, jak ji zlepšit, ale již nyní je zvláště díky některým novým funkcionalitám lepší než v současnosti nejpoužívanější nástroje svého typu, jako je například Flowgorithm. Je tak vidět, že zkušenosti z praxe o tom, jak studenti pracují s Flowgorithmem a jak se následně učí programovat v běžném programovacím jazyce, napomohly k několika dobrým úpravám, které aplikaci posunuly dál.

Během vývoje došlo k několika změnám, které zajistily, že část předem nasbíraných informací již nebyla aktuální (například autor PS-Diagramu, který původně zdánlivě nechtěl na svém nástroji s nikým spolupracovat, vydal PS-Diagram jako open source právě pro to, aby jej mohli studenti vylepšit například v rámci kvalifikačních prací) a podobným způsobem - nečekaně a ne vždy příjemně - se také vyvíjela i nová aplikace.

Důležité je hlavně to, že se u aplikace počítá s dalším vývojem a má proto navržené a připravené velké množství prvků pro budoucí úpravy a případné přidané funkcionality. Nebude proto pro další modifikace nutné rozsáhle modifikovat kód aplikace.

Ve vývoji aplikace je v plánu dále pokračovat a, i když se ji nepodařilo tento akademický rok nasadit pro výuku, nezbývá již žádný důvod, proč by neměla v únoru roku 2023 v předmětu Technické informatika ve strojírenství nahradit v současnosti používaný Flowgorithm.

Literatura

[1] *ASME standard; operation and flow process charts.* American Society of Mechanical Engineers, 1947.

[2] *Subway Fare Card Machine Flow Process Chart.jpg* [online]. US Federal Aviation Administration, 2008. [cit. 2021/06/12]. Flow Process Charts. Dostupné z: `https://commons.wikimedia.org/wiki/File:Subway_Fare_Card_Machine_Flow_Process_Chart.jpg`.

[3] *ISO 5807:1985.* ISO, Technical Committee: ISO/IEC JTC 1/SC 7, 1985. Dostupné z: `https://www.iso.org/standard/11955.html`.

[4] *NShape* [online]. https://www.dataweb.de/, 2011. [cit. 2021/08/12]. Dostupné z: `https://code.google.com/archive/p/nshape/`.

[5] *NodeXL* [online]. The Social Media Research Foundation, 2010. [cit. 2021/08/12]. Dostupné z: `https://archive.codeplex.com/?p=nodexl`.

[6] *Scratch Statistics* [online]. MIT Media Lab, 2021. [cit. 2021/06/12]. Scratch: Community statistics at a glance. Dostupné z: `https://scratch.mit.edu/statistics/`.

[7] *Visual Logic* [online]. PGS Systems, LLC, 2015. [cit. 2021/06/12]. Visual Logic: The Website. Dostupné z: `https://www.visuallogic.org/`.

[8] *Programming 4 Girls* [online]. 2017. [cit. 2021/06/12]. info@programming4girls.com. Dostupné z: `https://www.programming4girls.com/index.html`.

[9] *yFiles* [online]. yWorks, 2020. [cit. 2021/08/12]. Dostupné z: `https://www.yworks.com/products/yfiles`.

[10] Alan Kay, A. G. D. I. *Squeak* [online]. 1996. [cit. 2021/08/12]. Dostupné z: `https://squeak.org/`.

[11] Axel Böttcher, F. K. *Informatik für Ingenieure: Grundlagen und Programmierung in C.* München: Oldenbourg, 2012. ISBN 978-3-486-70527-0.

[12] Bartyzal, M. *PS-Diagram* [online]. 2012. [cit. 2022/05/09]. Dostupné z: `http://www.psdiagram.cz/`.

[13] Bartyzal, M. *Simulace průchodu programu vývojovým diagramem pro výuku algoritmizace* [online]. Jihočeská Univerzita v Českých Budějovicích, Pedagogická fakulta, 2012. [cit. 2021/06/12]. Bakalářská práce. Dostupné z: https://theses.cz/id/3tdsf6/.

[14] Briol, P. *BPMN, the Business Process Modeling Notation Pocket Handbook*. Lulu.com, 2008. Dostupné z: https://books.google.cz/books?id=1MPoUVrK7-8C. ISBN 9781409202998.

[15] Brooks, F. P. *The Mythical Man-Month: Essays on Softw.* Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1978. ISBN 0201006502.

[16] Carlisle, M. *RAPTOR* [online]. 2019. [cit. 2021/06/12]. Dostupné z: https://raptor.martincarlisle.com/.

[17] Carlisle, M. C. et al. RAPTOR: Introducing Programming to Non-Majors with Flowcharts. *J. Comput. Sci. Coll.* April 2004, 19, 4, s. 52–60. ISSN 1937-4771.

[18] Champion, J. *ZedGraph* [online]. 2005. [cit. 2021/08/12]. Dostupné z: https://sourceforge.net/projects/zedgraph/.

[19] Chung, J. *Stencyl* [online]. Stencyl, LLC, 2011. [cit. 2021/08/12]. Dostupné z: http://stencyl.com/.

[20] Cook, D. *Flowgorithm* [online]. 2014. [cit. 2020/08/06]. Dostupné z: http://www.flowgorithm.org.

[21] Dann, W. et al. Mediated transfer: Alice 3 to Java. 02 2012. doi: 10.1145/2157136.2157180.

[22] Gilbreth, F. – Gilbreth, L. – Mechanical Engineers, A. S. *Process Charts*. author, 1921. Dostupné z: https://books.google.cz/books?id=dULWGwAACAAJ.

[23] Hartree, D. *Calculating Instruments and Machines*. University Press, 1950. Dostupné z: https://books.google.cz/books?id=8Xe4AAAAIAAJ.

[24] Ivana Durdilová, L. S. *Animace uživateli vytvořených interaktivních vývojových diagramů pro výuku algoritmizace* [online]. 2006. [cit. 2012/04/18]. Dostupné z: http://lab.uzlabina.cz/~projekty/index.htm.

[25] Kay, A. *Etoys* [online]. 1996. [cit. 2021/08/12]. Dostupné z: https://etoysillinois.org.

[26] Kelleher, C. – Pausch, R. – Kiesler, S. Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, s. 1455–1464, New York, NY, USA, 2007. Association for Computing Machinery. doi: 10.1145/1240624.1240844. Dostupné z: https://doi.org/10.1145/1240624.1240844. ISBN 9781595935939.

[27] Kendrick, B. *Web Turtle* [online]. 2017. [cit. 2021/06/12]. Dostupné z: http://www.sonic.net/~nbs/webturtle/.

[28] Lavoie, M. *Logic of Algorithms for Resolution of Problems* [online]. 2004. [cit. 2021/08/12]. Dostupné z: http://www.marcolavoie.ca/larp/en/description/description.htm.

[29] Lessner, D. *Vývojový diagram míchaná vajíčka.svg* [online]. 2014. [cit. 2021/06/12]. Úvodní příklad vývojového diagramu s popisky. Dostupné z: https://popelka.ms.mff.cuni.cz/~lessner/mw/index.php/Soubor:V%C3%BDvojov%C3%BD_diagram_m%C3%ADchan%C3%A1_vaj%C3%AD%C4%8Dka.svg.

[30] Microsoft. *Language Integrated Query (LINQ) (C#)* [online]. Microsoft Build, 2022. [cit. 2022/05/13]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.

[31] Midkin, S. *Drakon* [online]. 2019. [cit. 2021/06/12]. Drakon IDE. Dostupné z: https://app.drakon.tech.

[32] Midkin, S. *DrakonHub* [online]. 2022. [cit. 2022/04/24]. Drakon Documentation. Dostupné z: https://drakonhub.com/read/docs.

[33] Moskal, B. – Lurie, D. – Cooper, S. Evaluating the Effectiveness of a New Instructional Approach. *SIGCSE Bull.* March 2004, 36, 1, s. 75–79. ISSN 0097-8418. doi: 10.1145/1028174.971328. Dostupné z: https://doi.org/10.1145/1028174.971328.

[34] Palesz. *Graph#* [online]. 2011. [cit. 2021/08/12]. Dostupné z: https://archive.codeplex.com/?p=graphsharp.

[35] Paranoid. *File:Dutch cryo Moscow.png* [online]. KrioRus, 2007. [cit. 2021/06/12]. A DRAKON diagram describing a transportation of a cryonics patient. Dostupné z: https://en.wikipedia.org/wiki/DRAKON#/media/File:Dutch_cryo_Moscow.png.

[36] Parondžanov, V. *Icons of DRAKON language* [online]. wikipedia.org, 2013. [cit. 2022/04/28]. Dostupné z: https://en.wikipedia.org/wiki/DRAKON#/media/File:Icons_of_Visual_Programming_Language_--DRAKON--.png.

[37] Parondžanov, V. *Macroicons of DRAKON language.* [online]. wikipedia.org, 2013. [cit. 2022/04/28]. Dostupné z: https://en.wikipedia.org/wiki/DRAKON#/media/File:Macroicons_of_Visual_Programming_Language_DRAKON.png.

[38] Parr, T. *ANother Tool for Language Recognition* [online]. 1992. [cit. 2021/08/12]. Dostupné z: https://www.antlr.org/.

[39] Pattis, R. E. *Karel The Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons, 1981. ISBN 0-471-59725-2.

[40] Pausch, R. *Alice* [online]. Carnegie Mellon University, 1998. [cit. 2021/08/12]. Dostupné z: https://www.alice.org/.

[41] Philipp, F. *Multiple Branching.svg* [online]. 2012. [cit. 2021/06/12]. Example of a Nassi–Shneiderman diagram. Dostupné z: https://en.wikipedia.org/wiki/Nassi%E2%80%93Shneiderman_diagram#/media/File:Multiple_Branching.svg.

[42] Skarabo, M. *BPMN - A Process with normal flow* [online]. 2014. [cit. 2021/06/12]. Example of a Business Process Model and Notation for a process with a normal flow. Dostupné z: https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation#/media/File:BPMN-AProcessWithNormalFlow.svg.

[43] Smirnov, A. *GraphX for .NET* [online]. 2015. [cit. 2021/08/12]. Dostupné z: https://archive.codeplex.com/?p=graphx.

[44] Smith, J. *Patterns - WPF Apps With The Model-View-ViewModel Design Pattern* [online]. MSDN Magazine, 2009. [cit. 2022/04/28]. Dostupné z: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern.

[45] Snoza, I. *Interaktivní dynamické vývojové diagramy* [online]. Univerzita Pardubice, Katedra softwarových technologií., 2009. [cit. 2021/06/12]. Bakalářská práce. Dostupné z: http://dspace.upce.cz/handle/10195/34891.

[46] Szalkai, B. *Satsuma* [online]. 2013. [cit. 2021/08/12]. Dostupné z: https://sourceforge.net/projects/satsumagraph/.

[47] Wally Feurzeig, C. S. S. P. *Logo* [online]. 1967. [cit. 2021/08/12]. Dostupné z: https://el.media.mit.edu/logo-foundation/what_is_logo/index.html.

[48] Wilensky, U. *NetLogo* [online]. 1999. [cit. 2021/08/12]. Dostupné z: https://ccl.northwestern.edu/netlogo/.

[49] Yoder, C. M. – Schrag, M. L. Nassi-Shneiderman Charts an Alternative to Flowcharts for Design. *SIGMETRICS Perform. Eval. Rev.* January 1978, 7, 3–4, s. 79–86. ISSN 0163-5999. doi: 10.1145/1007775.811104. Dostupné z: https://doi.org/10.1145/1007775.811104.