Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Vizuální programovací jazyk

2017

Bc. Zdeněk Vídeňský

Vedoucí práce: Mgr. Tomáš Kühr, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Zdeněk Vídeňský
Název práce:	Vizuální programovací jazyk
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2017
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	Mgr. Tomáš Kühr, Ph.D.
Počet stran:	66
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Zdeněk Vídeňský
Title:	Visual programming language
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2017
Study field:	Computer Science, full-time form
Supervisor:	Mgr. Tomáš Kühr, Ph.D.
Page count:	66
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Tato diplomová práce se zabývá návrhem a implementací vizuálního programovacího jazyka vhodného pro výuku algoritmizace a základů programování mladších dětí. O interpretaci jazyka se stará serverová část napsaná v ASP.NET Core Frameworku, která komunikuje s klientskou částí programu poskytující uživatelsky přívětivé vývojové prostředí pro tento jazyk. Vývojové prostředí je napsané v jazyce JavaScript s využitím knihoven React a Redux a je dostupné z internetových prohlížečů.

Synopsis

This master's thesis deals with design and implementation of visual programming language for teaching basics of algorithms and programming for younger children like students on elementary and high school. Server side part of system is written in ASP.NET Core Framework which takes care of interpreting program and communicating with client side part of system which provides user friendly development environment. This part is written in JavaScript using React and Redux libraries to be accessible in internet browsers.

Klíčová slova: vizuální programovací jazyk; programovací jazyk; ASP.NET; .NET Core; React; Redux; výukový program; výuka programování

Keywords: visual programming language; programming language; ASP.NET; .NET Core; React; Redux; education program; programming education

Rád bych poděkoval mému vedoucímu práce Mgr. Tomáši Kührovi, Ph.D. za vedení a odborné konzultace k práci a dále Mgr. Tomáši Miličkovi a Ing. Františkovi Vařachovi z GPOA Znojmo a Mgr. Radku Tomaštíkovi ze ZŠ nám. Republiky Znojmo za umožnění otestování výsledné aplikace přímo v hodinách informatiky na těchto školách.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

1 Úvod

Výuka algoritmizace a programování se vyučuje jak na středních školách, které jsou přímo zaměřené na informatiku, tak i na gymnáziích a probíhá už i na některých základních školách. Právě na základních školách se při prvotních výkladech musí myslet na to, že děti nemají silný matematický základ. Samozřejmě je nejdříve potřeba vůbec vysvětlit základní prvky algoritmizace a programování, jako je například proměnná, řídící konstrukce, cykly, atd. K tomu všemu je potřeba přívětivé a intuitivní uživatelské rozhraní, které dokáže pochopit a využít i student, který s programováním nikdy nepřišel do styku.

V poslední době se objevuje více a více již hotových řešení výukových prostředí, které umožňují vytvoření programu s využitím programovacích bloků na vyšší úrovni abstrakce a výsledkem je například hra nebo animace, která je právě nejzajímavější pro mladší studenty. O těchto řešeních a celkově o výuce algoritmizace a programování budeme mluvit v kapitole 3.

V kapitole 4 si popíšeme myšlenku vizuálního programovacího jazyka, podíváme se na dostupná řešení a představím návrh svého vlastního jazyka, které bude podloženo grafickými návrhy a výčtem možností, které bude nabízet.

Kapitola 5 se již věnuje návrhu systému jako celku po technické stránce. Řekneme si, jaké technologie bude využívat, jak bude systém pracovat, jaký bude mít funkční rozsah a představím návrh uživatelského rozhraní. Konkrétnímu technologickému zpracování jako jsou jednotlivé funkční bloky vizuálního programovacího jazyka, serverová část (backend) systému a implementace klientské části (frontend) systému jsou podrobně popsány v kapitolách 6, 7. V kapitole 8 najdeme uživatelskou příručku projektu, která obsahuje útržky z oficiální nápovědy. Celá 9. kapitola je věnována konečnému testování systému jak formou unit testů, kde šlo hlavně o ověření správného fungování jednotlivých bloků, tak i praktické testování přímo ve vyučování na znojemské střední a základní škole.

Na závěr si celou práci shrneme a představíme si, jak by šel systém případně rozšířit nebo vylepšit na základě praktického testování.

2 Počítač ve výuce

Zavedení počítačů do výuky se uskutečňuje a ve velké míře již uskutečnilo v mnoha zemích. Využití počítače ve výuce se dá rozdělit na dvě částečně se překrývající oblasti a to je výuka o počítačích a výuka s počítači. Tyto dvě oblasti se jen částečně překrývají. Moje práce se zabývá spíše tou první oblastí a to je výuka o počítačích, kde se vlastně student seznámí s programováním počítače a je tedy zapotřebí, aby uměl alespoň v určitém rozsahu s počítačem komunikovat. Předpokládá se ale, že výuka programování se bude učit na školách zaměřených na informatiku, takže studenti by měli být schopni s počítačem komunikovat.

Dále se výuka s počítačem ještě rozděluje na počítačově podporovanou výuku a počítačově řízenou výuku s počítačem. U počítačově podporované výuky počítač přejímá jen některé funkce učitele. Je například potřebný výklad učitele, aby studentům vysvětlil úkol a oni ho pomocí počítače mohli splnit. Bez učitele by studenti nebyli schopni tento úkol vyřešit nebo jen velmi obtížně. U počítačově řízené výuky ale není vůbec potřeba výkladu učitele a počítač přejímá všechny funkce učitele. Učitel se tak stává jen konzultantem studentů.

2.1 Klady a zápory počítačové výuky

Počítačová výuka umožňuje respektovat podmínky a styl duševní práce jednotlivých posluchačů v takovém rozsahu, jaký je pro výuku orientováno na učitele nebo písemné prameny nedosažitelný. Aktivizace studenta je zaručena jeho způsobem komunikace s programem, který bez jeho aktivity nemůže pokračovat, resp. není dokončená úloha. Je zde dále podporována samostatnost a možnost kontrolovat skutečně veškeré práce každého studenta. Navíc má počítač možnost okamžité zpětné vazby, tedy není potřeba čekat na opravu úkolu učitelem.

Samozřejmě má počítačová výuka i některé zápory. Při počítačové výuce se může vyskytnout vzdálení učitele od studenta, tedy že student si vystačí ve výuce sám bez učitele. Řešením může být vytvoření nových rolí pro učitele jako třeba role konzultanta nebo role vedoucího projektů a jiných forem samostatné práce studentů. Může dojít také k omezení samostatností studentů a to kvůli systému řízení samostatné práce, který potlačuje tvořivost posluchačů a neponechává jim podíl na řízení svého vzdělávání. Řešením může být ponechání studentům takovou míru řízení, která odpovídá jejich vyspělosti. Jako poslední zápor počítačové výuky může být značná pracnost přípravy počítačové výuky.

2.2 Funkce počítače v pedagogickém procesu

Počítač ve výuce chápeme jako jeden z audiovizuálních prostředků. Má ale mnohem více možností, než klasické pomůcky. Například přenos a sdílení informací, řízení procesů a činností, sběr, uchování a zpracování dat nebo zajišťování zpětné vazby.

2.2.1 Student a počítač

Při prezentaci poznatků je možné využít grafiky, animací nebo videozáznamů pro zefektivnění a znázornění výuky. Počítač je možno využít jako archiv různých znalostí. S tím je i spojena možnost využití vyhledávání informací na internetu. Právě vyhledáváním informací na internetu má tak student větší svobodu při řešení úkolu a podporuje se tak jeho schopnost samostatně vyhledávat nebo plnit úkoly.

2.2.2 Učitel a počítač

Učiteli slouží počítač jako pracovní nástroj při přípravě a plánování pedagogického procesu (evidence studentů, známkování, poznámky nebo prezentace). V žádném případě však nenahradí osobnost učitele. Počítač nemá možnost „vcítění“(empatie), která je dána každému člověku. Je také nutno zvážit, je-li nasazení počítače pro danou činnost skutečně vhodné.

2.3 Posuzování a hodnocení výukových programů

Technika, která zefektivňuje výuku, musí umožnit, aby učitel učil méně a žák se naučil více. Tato zásada je i základním cílem tvůrců výukových programů. Nasazení technických prostředků musí přinést minimálně jeden z následujících efektů:

1. zlepšení kvality výuky,
2. ušetření času a práce studentů a vyučujících,
3. ušetření materiálních hodnot.

Počítač má tedy ve výuce velice široké využití, zvlášť v dnešní době, kdy je nejsnadnější přístup na internet, na kterém najdeme spoustu užitečných zdrojů pro učení. Využití počítače pro můj projekt bude nezbytný, protože spadá také do kategorie výuky o počítačích, konkrétně do výuky algoritmizace programování. Všechny informace uvedené v této kapitole vycházely z [11].

3 Výuka algoritmizace a programování

Výuka algoritmizace a programování je složitější už z toho důvodu, že musí žáky naučit úplně jinému způsobu přemýšlení nad daným problémem a jsou zde zapotřebí již nějaké základy matematiky a matematických struktur, což může být problém například na základních školách.

Sám jsem si již výuku programování a algoritmizace vyzkoušel na studentech střední školy a to jak na studentech z informatického oboru, tak i z víceletého gymnázia. Při výuce jsem postupoval od nejzákladnějších pojmů, jako je algoritmus, proměnná, datové typy, operátory, řídící struktury nebo cykly. Největší problém byl právě ve výukovém prostředí, protože jsem učil základy jazyka C#. Hlavní nevýhoda byla ta, že bylo kromě algoritmizace potřeba vysvětlit i základní konstrukce jazyka C#, což vyžaduje další zapamatování názvosloví a to už bylo pro studenty matoucí. Podíváme se nyní na teoretické metody učení algoritmizace a programování.

3.1 Metody učení

Metody učení by se daly rozdělit na teoretické a praktické. Algoritmizace a programování se nedá naučit jen s jednou z těchto částí. V teoretické části by měl být vysvětlen pojem algoritmus tak, aby tomu studenti správně porozuměli. Dále je potřeba vysvětlit i teoreticky, jak vůbec počítače takové algoritmy zpracovávají, aby i studenti dokázali tímto způsobem o daném problému přemýšlet. Potom jsou na řadě pojmy jako datové typy, datové struktury, proměnné a v neposlední řadě řídící struktury a funkce.

Co se týče praktické části, bývá také problém v tom, aby studenti samostatně řešili dané příklady. Na začátek je určitě dobré, aby jim byly poskytnuty již vyřešené jednoduché příklady, na kterých se mohou podívat, jak se daný problém řeší a potom vyřešit o něco složitější příklad samostatně. Najít vyvážený poměr v učení teorie a následné vyzkoušení si těchto znalostí v praxi je klíčové, ale asi nejdůležitější je, aby programování obecně vzbudilo ve studentech zájem a byli ochotni si programování vyzkoušet doma.

3.2 Algoritmické myšlení

Algoritmus jako takový je definován jako schematický postup pro řešení určitého druhu problémů, který je prováděn pomocí konečného počtu přesně definovaných kroků. Pojem algoritmus se využívá nejen v informatice, ale také v matematice nebo to může být jakýkoliv druh práce, který splňuje tuto definici (návod na smontování, recept). Přemýšlení tímto způsobem je základem pro naučení programování. V počítači jsou ony přesně definované kroky operace, které vykonává procesor, resp. příkazy v programovacím jazyce. [19]

Znovu je zde otázka, jak studenty tomuto myšlení naučit. Jedině praktickými příklady ve vhodném prostředí. S tímto jde v ruku v ruce i věková skupina studentů. Jak rozhodnout, kdy by schopnosti studenta mohly stačit na to, aby

se celá třída učila programování? Na základní škole najdeme asi jen vyspělejší jednotlivce, kteří by stačili na naučení se programovat a zde zařídit například zájmový kroužek, ale integrace do výuky základní práce s počítačem by asi nebyla na místě. Nejvhodnější věkovou skupinu představují středoškolští studenti, které budou hlavní cílovou skupinou.

3.3 Vhodný programovací jazyk

Pokud již ve výuce algoritmizace a programování mají studenti některé teoretické základy, kde ještě dosud nepotřebovali nic prakticky programovat a chtějí si nabyté znalosti vyzkoušet v praxi, naskýtá se otázka, jaký zvolit nejvhodnější programovací jazyk. Tímto nejvhodnějším jazykem mám na mysli jazyk, který se lze lehce naučit a nemá nijak složitou syntaxi a další otázka je, jestli má cenu, aby se vůbec učili nějaký konkrétní jazyk?

V době psaní této diplomové práce již třetím rokem vedu zájmový kroužek zaměřený na programování. Každý rok jsem učil programovat jiný druh aplikací, ale vždy jsem je učil v jazyce C#. Prvním rokem jsem si vyzkoušel učit úplné základy tohoto jazyka jako je algoritmizace a programování obecně a zjistil jsem, že studentům dělá potom největší problém se zorientovat v kódu, než že by nevěděli, jak daný problém vyřešit. Určitě existují i jazyky, které by se daly naučit rychleji a lépe, než je C#. Jako první mě napadl například jazyk Python, který je klasifikován jako skriptovací jazyk. Je to ale velmi silný jazyk a obsahuje všechno, co lze najít v jiných jazycích včetně objektově orientovaného programování. Je také velmi snadno dostupný a snadno přenositelný. [10]

Jak jsem již psal, je zde otázka, zda pro výuku algoritmizace a programování vůbec učit konkrétní textový programovací jazyk. Hlavní nevýhoda je naučení se syntaxe daného jazyka. Už jsem se setkal s tím, že jsem chtěl studentům vysvětlit jeden pojem, který se zdá být jednoduchý, ale k tomu, aby ho pochopili by potřebovali znát několik dalších pojmů a tak by z toho vzniklo spíše učení syntaxe jazyka než algoritmizace. A pak narazíme právě na to, že studenti nejsou schopni samostatně vyřešit příklad, když nevědí, jak daný problém zapsat. Částečným řešením by mohly být již vytvořené výukové programy na programování.

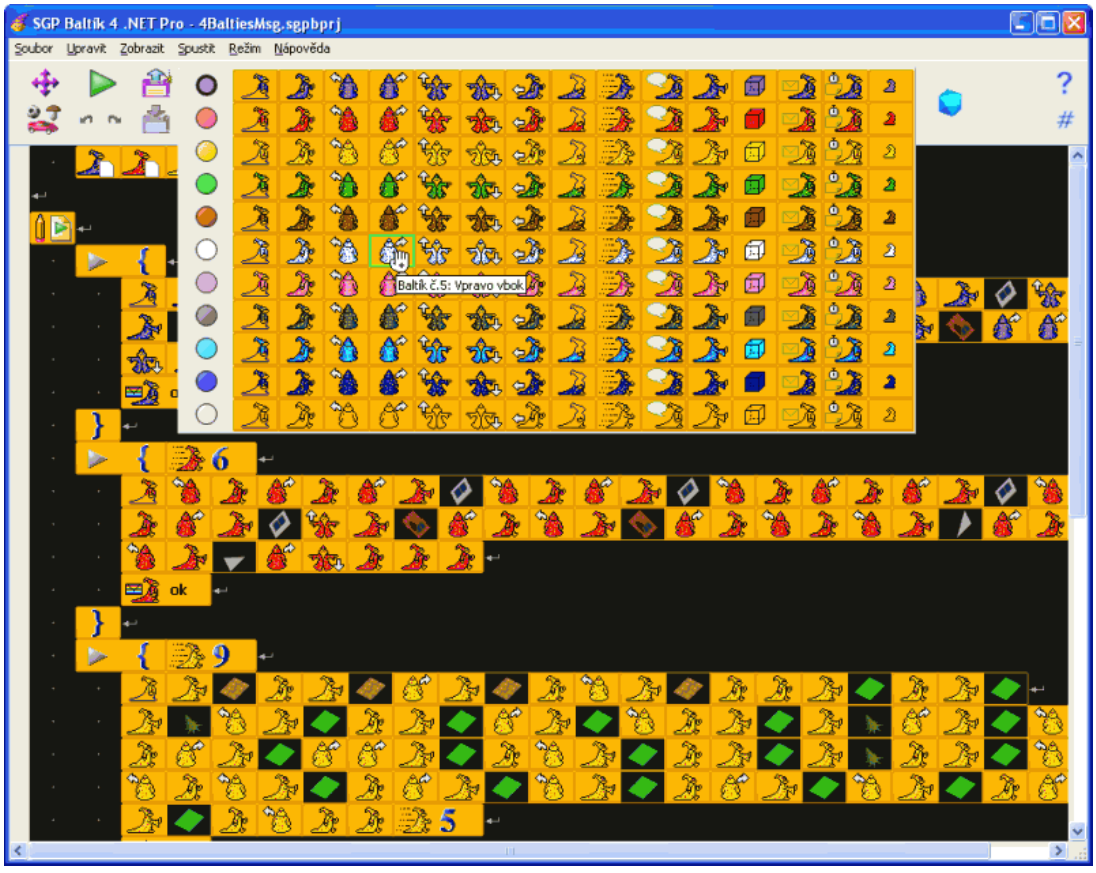
3.4 Výukové programy

V této podkapitole se podíváme na dostupné výukové programy, které jsou ve větší nebo menší míře používány na školách pro výuku programování. Konkrétně v České republice je asi největším problémem fakt, že tyto programy nejsou lokalizovány do češtiny a to může být komplikace hlavně pro mladší studenty. Každoročně také probíhá celosvětová akce s názvem Hour of code (Hodina kódu) jeden týden v prosinci. Tato akce je pořádána neziskovou organizací Code.org. Cílem je, aby se na školách po celém světě udělal úvod do informatiky pro co nejvíce lidem, kteří s ní dosud nepřišli do styku a to nejčastěji na základních nebo i středních školách. Hodinu kódu může zorganizovat prakticky kdokoliv,

když má dost zájemců a prostory. Na oficiálních stránkách je také velké množství materiálů a tutoriálů včetně předpřipravených programů. Do této akce jsou zapojeni i velká skupina partnerů jako je Microsoft, Apple nebo Amazon. Většina těchto výukových programů jsou zaměřeny na vytváření her, protože je to nejsnazší a nejzábavnější způsob, jak zaujmout studenty tak, aby se sami začali učit programování. [20]

3.4.1 Baltík

Jako první si představíme program s názvem Baltík. Je od české společnosti SGP Systems a jde v podstatě o programovací jazyk i s vlastním vývojovým prostředím pro výuku programování na základních nebo středních školách. V programu Baltík se programuje pomocí obrázkových ikon, ale od verze 4 lze programovat i pomocí kódu, konkrétně v jazyce C#. Baltík je zatím jediným plně grafickým (ikonovým) standardním programovacím jazykem na bázi C a Pascal.



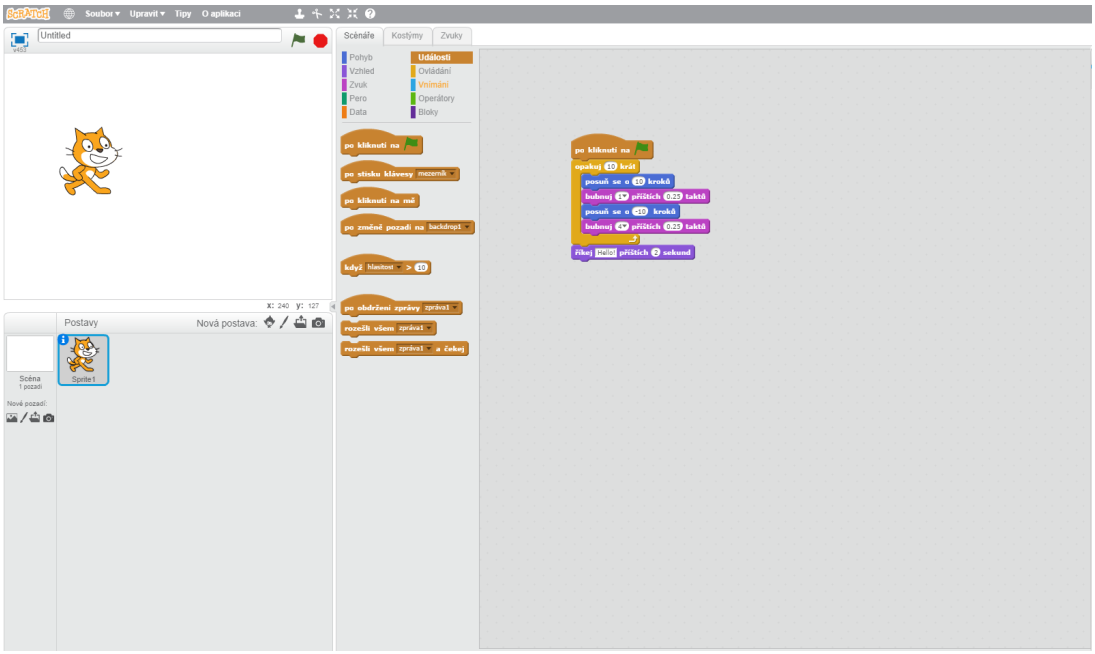
Obrázek 1: Vývojové prostředí Baltie 4 C# [3]

Program Baltík se skládá z několika režimů a to režim skládání scény, programovací a malování. V prvním režimu se skládá scéna z malých obrázků, které se

dají v dalším režimu programovat. V režimu programování se programuje herní logika pomocí ikonek, kde každá představuje kus kódu. Do scény lze vložit i předdefinovanou postavičku čaroděje Baltíka, kterého lze pomocí ikonek programovat. Obsahuje mimo jiné i ikonky pro proměnné, podmínky, cykly, základní matematické funkce, animace a další. Jde o celkem komplexní vizuální programovací jazyk pro ty nejmladší studenty. Pořádají se i mezinárodní soutěže Baltie, kde jsou zapojeny školy z několika zemí. [3]

3.4.2 Scratch

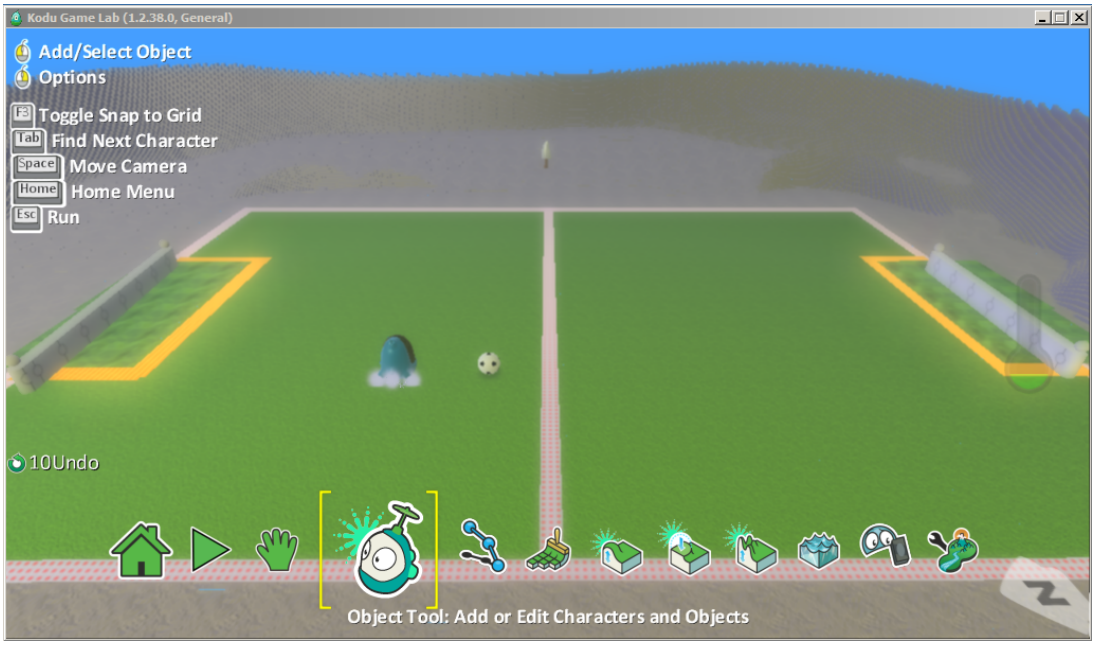
Vizuální programovací jazyk Scratch je další ze zástupců jazyků, které se využívají ve školách k výuce algoritmizace a programování. Autorem jazyka je Mitchel Resnick, který působil ve skupině v rámci MIT Media Lab a do určité míry se inspirovali právě českým Baltíkem. Jazyk Scratch je dostupný online na stránkách www.scratch.mit.edu. Projekty je zde možné vytvářet, upravovat a sdílet. Programovací bloky jsou rozděleny do několika kategorií a to Pohyb, Vzhled, Zvuk, Pero, Data, Události, Vnímání, Operátory a Bloky. Tyto programovací bloky se potom vkládají na tzv. Sprity, což jsou obrázky, ať už bitmapové nebo vektorové, které představují postavičky v programu. Výsledkem může být hra, animace a další. Celý jazyk je také lokalizován do češtiny a je dostupný zdarma bez jakékoliv registrace a výsledek lze sdílet na stránkách a pochlubit se tak ostatním. [28]



Obrázek 2: Ukázka online editoru jazyka Scratch [28]

3.4.3 Kodu

Jako posledního zástupce si představíme vizuální programovací jazyk Kodu od výzkumné skupiny Microsoftu, který je vytvořen speciálně pro tvorbu 3D her. Jde tedy zároveň i o herní engine s jednoduchým ovládáním, kde si každý může vytvořit hru bez znalostí programování. Jazyk Kodu je do jisté míry inspirován jazykem Scratch, který jsme si popsali výše a je dostupný pro platformy Windows a Xbox 360. Tvorba probíhá v trojrozměrném světě, po kterém se jako pozorovatel pohybujete a na určitých místech můžete přidávat objekty a ty programovat znovu pomocí obrázkových příkazů. Celý svět si můžete úplně od začátku vytvořit, tím je myšleno povrch, kde lze tvarovat i nejrůznější nerovnosti. Znovu zde funguje něco jako hlavní hrdina (jako v případě Baltíka) robot Kodu nebo další roboti, kteří se dále programují. Programovací logika je založená ne jednoduchých podmínkách WHEN-DO. Samozřejmostí je možnost si výsledek hned vyzkoušet přímo v programu. Výsledky lze, stejně jako u jazyka Scratch, sdílet na speciálních stránkách a pochlubit se tak svým výsledkem ostatním. [14]



Obrázek 3: Ukázka programu Kodu [14]

Všechny tyto výše zmiňované programy mají jako hlavní cíl naučit jen a pouze algoritmické programování a základy programování ne tím, že by učili syntaxi programovacího jazyka, ale učí jen způsob, jakým se programy vytvářejí. Potom je už na každém, jaký jazyk se naučí, protože každý jazyk má své pro i proti a je dobrý k rozdílným druhům úloh.

4 Vizuální programovací jazyk

Myšlenka vizuálního programovacího jazyka, jak už název napovídá, je v procesu programování pomocí vizuálních prvků, které představují určitý kus kódu. Nejčastěji jde o obrázkové reprezentace v podobě ikonek. Odstraňuje se tak nutnost naučení se syntaxe konkrétního jazyka a stačí jen vědět, jak vhodné stavební bloky poskládat tak, aby program dělal to, co chceme. Má to své výhody i nevýhody. Výhodou je, že nemusíme znát syntaxi daného jazyka, pokud jsou bloky jednoznačně označené. Nevýhoda těchto jazyků je jejich funkční omezenost. Jde v podstatě jen o používání již předpřipravených bloků v konkrétním programovacím prostředí. Další nevýhodou může být přenositelnost. Většina těchto vizuálních programovacích jazyků potřebuje ke svému chodu nainstalované prostředí a program nekompiluje, jen interpretuje. Ve vizuálním programovacím jazyce tedy žádný v praxi použitelný program nevytvoříte, ale na naučení programování je to podle mě ta nejlepší volba.

4.1 Vlastní návrh

Ještě před samotným návrhem programovacího jazyka jsem si položil několik otázek ohledně toho, co daný jazyk vůbec bude umět. Nechtěl jsem se pouštět do herního enginu, aby se pomocí tohoto jazyka naprogramovala hra. Na to existují již řešení jako třeba Scratch nebo Kodu, jak jsme si již popsali. Chtěl jsem jednoduchý jazyk, který má všechny důležité vlastnosti, které obsahuje plnohodnotný programovací jazyk, ale aby to byl pořád výukový program, který má za úkol naučit algoritmizaci a programování úplné začátečníky. V původním návrhu bylo také definování a volání funkcí, které jsem nakonec vypustil hned z několika důvodů. Prvním důvodem je, že jsem chtěl jazyk primárně pro výukové účely a právě rozdělování programu do podprogramů by v případě programování pomocí bloků značně znepřehlednilo výsledný program. Dále to byl i problém návrhu, jak udělat přehledný systém na definování všeho, co funkce potřebuje, jako například proměnný počet parametrů a jak to pak přehledně zakomponovat do uživatelského rozhraní a vytvořit k němu korespondující blok. Výčet možností výsledného návrhu vizuálního programovacího jazyka nazvaného pracovně **VizProg** jsou následující:

- Datové typy:
  - Číslo (float),
  - Text,
  - Boolean;
- Stuktura pole;
- Proměnné;
- Řídící příkazy:

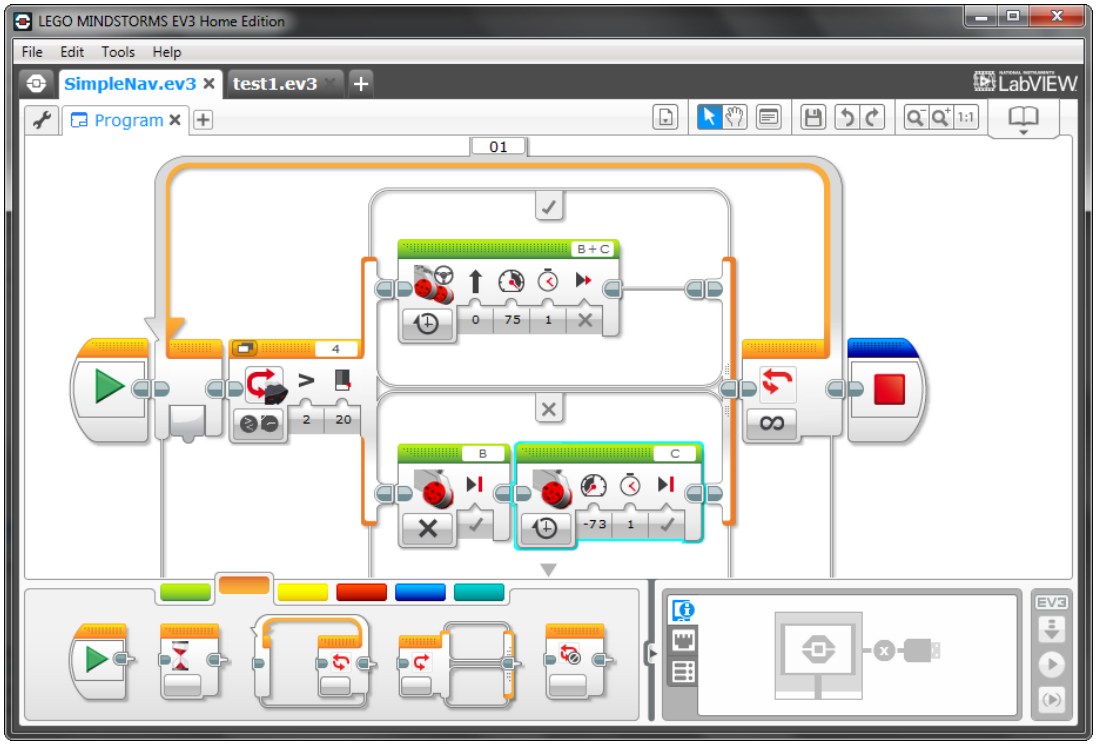
  - Podmínka,
  - Cyklus;

- Operace:
  - Aritmetické,
  - Porovnání,
  - Logické;

4.2 Grafické zpracování

Jak jsme si již na začátku řekli, nejdůležitějším prvkem na vizuálním programovacím jazyku je jeho grafická reprezentace jednotlivých stavebních bloků, které dohromady tvoří program. Pro výše uvedený výčet možností vizuálního programovacího jazyka je potřeba navrhnout grafickou reprezentaci všech jednotlivých položek.

Stanovil jsem si několik požadavků na tento grafický návrh. Hlavní požadavek je barevné sladění jednotlivých bloků tak, aby každý z nich byl na první pohled barevně rozlišitelný, ale aby společně dohromady netvořily nepřehlednou barevnou směs. Dalším požadavkem je, aby i tvary bloků působily příjemně. Zároveň je důležité, aby informace, které tyto bloky obsahují (název nebo hodnota proměnných) byly dobře čitelné. Představím zde grafický návrh na jednotlivé hlavní bloky programu. Částečně jsem se podobou bloků inspiroval v programu Lego Mindstorms [2]. Jde o prostředí pro programování stejnojmenné stavebnice s mikrokontrolérem od firmy Lego, kde je možné naprogramovat mikrokontrolér připojený ke kostičkám Lego. Jde také o vizuální programovací jazyk a jsou zde hezky vyřešeny proměnné a další řídící struktury.



Obrázek 4: Vývojové prostředí Lego Mindstorms [2]

Bloky lze buď za sebe skládat jako puzzle nebo je mít libovolně rozmístěné po obrazovce a spojovat je za pomocí spojujících čar jako diagramy. Celý program začíná startovacím blokem (značený velkým zeleným logem play) a odtud zleva doprava probíhá celý algoritmus. Vstupy i výstupy jsou znovu spojovány pomocí spojujících čar mezi jednotlivými bloky.

5 Návrh systému

V této kapitole si představíme podrobný návrh systému z technického hlediska, tedy výčet toho, jaké technologie budou použity na jednotlivé části systému a jak bude vypadat výsledné uživatelské rozhraní programu.

5.1 Technické zpracování

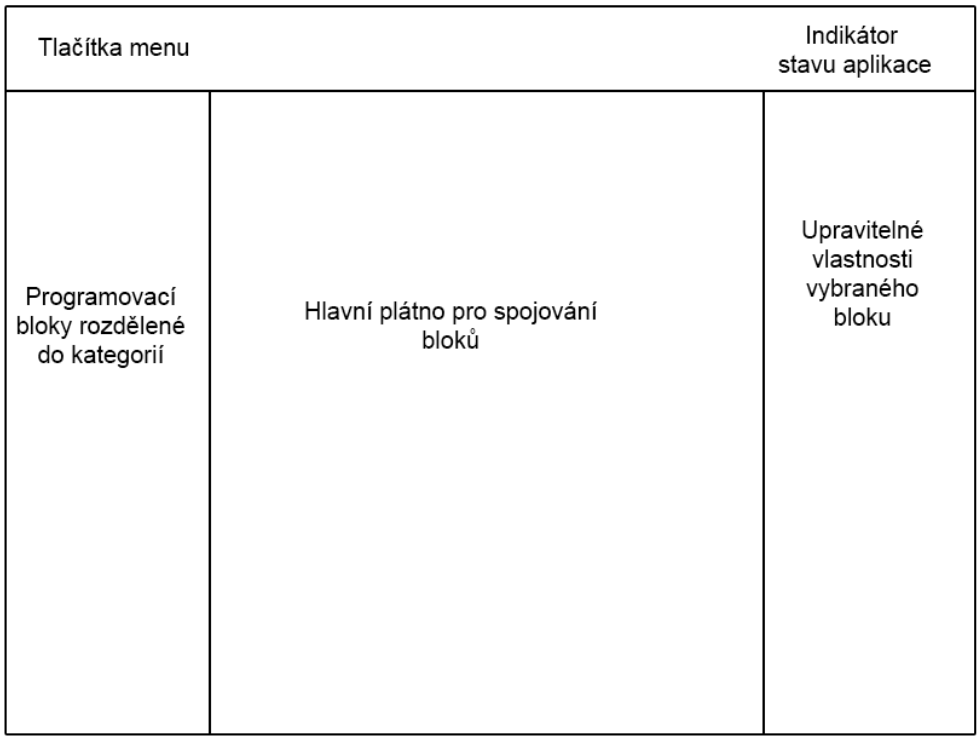
Celý systém bude napsán jako client-server[1] aplikace. Klient bude poskytovat grafické uživatelské rozhraní, ve kterém se budou programové bloky skládat a navzájem spojovat a tak vytvářet program. Veškeré výpočty programu a interpretace programu, se kterou je spojena také syntaktická a sémantická analýza se bude odehrávat na serveru a ten zase pošle příslušné informace buď o chybě nebo o úspěchu (i s případným výstupem) zpátky klientovi.

Protože bude systém řešený jako webová aplikace, může být klientem internetový prohlížeč, který podporuje JavaScript a další knihovny, které budou podrobněji popsány v kapitole 6 o backendu systému. Server poběží jako samostatná aplikace buď na cloudu nebo lokálně na počítači na daném portu. Klient bude napsán v JavaScriptu [23] s využitím knihoven **React** [26] a **Redux** [7]. Server bude napsán v jazyce C# s využitím technologie **ASP.NET Core** a klient se serverem spolu budou komunikovat zprávami ve formátu **JSON** [12]. Systém napsaný tímto přístupem v těchto technologiích bude přenositelný, tedy spustitelný na jakékoliv platformě a díky oddělení klienta od serveru je možné napsat například klienta pro desktop nebo nativní aplikaci a server bude pracovat pořád stejně.

5.2 Návrh uživatelského rozhraní

Při návrhu uživatelského rozhraní jsem, stejně jako u grafické návrhu bloků, vycházel z programu Lego Mindstorms. Chtěl jsem co nejjednodušší a nejpřehlednější grafické rozhraní, aby se uživatel ihned zorientoval a věděl přesně, kam má kliknout. Celá aplikace je přizpůsobena šířce a výšce monitoru a případné přesahu jsou scrollovatelné. Aplikace ale není responzivní a nezobrazuje se správně na mobilních zařízeních, kde ani nejde programovat. Aplikace tedy vyžaduje prohlížeč na počítači.

[1] Klient-server je síťová architekturá, která odděluje klienta (nejčastěji s grafickým rozhraním) od serveru přes počítačovou síť.[13]

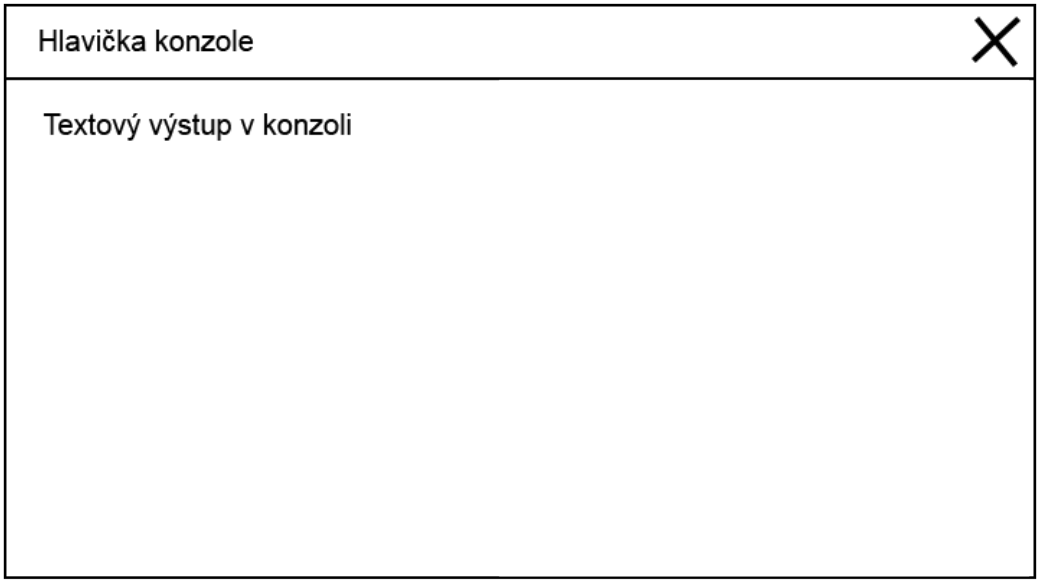


Obrázek 5: Rozložení uživatelského rozhraní

V části „Tlačítka menu“ vlevo nahoře jsou na jednom místě všechny ovládací tlačítka aplikace jako je například vytvoření nového programu, uložení programu, nahrání uloženého programu, spuštění programu, ladění nebo nápověda. Vpravo nahoře se nachází „Indikátor stavu aplikace“. Jde o jednoduché barevné zobrazení, jestli se aplikace nachází ve stavu bez chyb nebo jestli došlo k nějaké chybě při překladu nebo spojení se serverem.

Většina aplikace je rozdělena na tři vertikální části. Jako první je „Programovací bloky rozdělené do kategorií“. V této části jsou pod sebou formou rámečků rozděleny programovací bloky do kategorií, jako je například kategorie „Základní komponenty“, „Pole“, „Řídící příkazy“, „Hodnoty“, atd. Každý blok v dané kategorii zde má podobu obrázku, který graficky reprezentuje vlastnosti programovacího bloku. Například pro proměnnou je to ikona kufru, pro cyklus zacyklené kolečko s šipkami, apod. Podržením levého tlačítka myši a následným přetažením ikony na „Hlavní plátno pro spojování bloků“ se na plátně vytvoří reprezentace bloku podle přetažené ikony. Na tomto plátně se zobrazují bloky v plné podobě i s detaily (u proměnné například její název, mód nebo u čísla jeho hodnota). Na začátku programu musí vždy stát právě jeden startovací blok, na který se dále připojují další řídící bloky. Spojování bloků je realizováno za pomocí spojovacích čar, které tak spojují bloky za sebou jak jde program (od startovního bloku) a také se jimi přivádí vstupy do bloků, resp. výstupy z bloků. Princip je tedy dosti podobný, jako je u Lego Mindstorms. Pokud chceme určité vlastnosti bloku upravit, slouží na to poslední sekce a to jsou „Upravitelné vlast-

nosti vybraného bloku“. Výběr bloku je realizován stisknutím levého tlačítka myši na určitý blok. V této sekci se potom zobrazí upravitelné vlastnosti bloku, které lze upravit a upravují se například pomocí textového vstupu nebo výběrového menu. Tyto změny se ihned projeví na bloku na hlavním plátně.



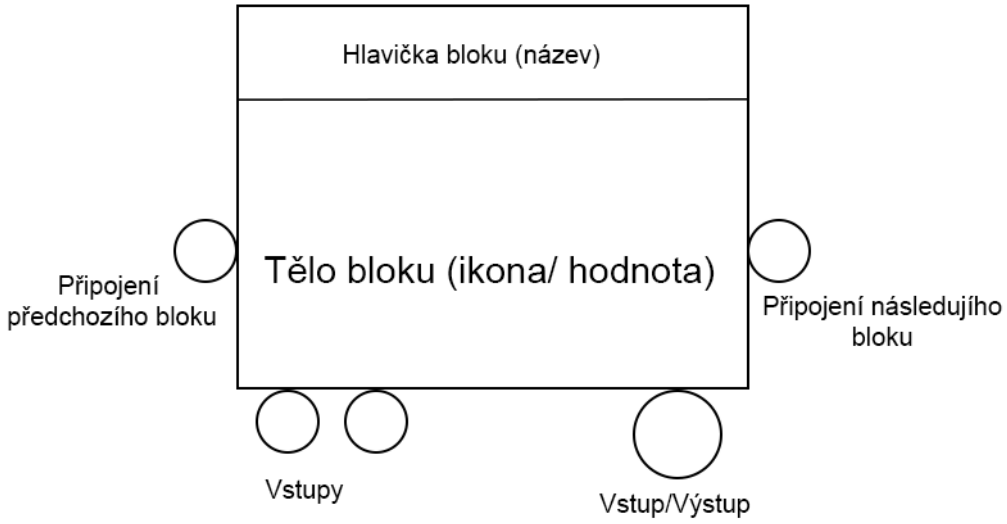
Obrázek 6: Design výstupní konzole

Spuštění programu je pak realizováno zobrazením modálního okna, která má podobnou strukturu jako klasický terminál na operačních systémech distribuce Linux nebo Windows. Výstup je zde jen textový. Toto modální okno se objeví jenom po úspěšném spuštění aplikace, která na serveru projde syntaktickou a sémantickou analýzou. Pokud aplikace nemá žádný výstup, zůstane „konzole“ prázdná.

5.3 Návrh bloku

Nejnáročnější část grafického návrhu byl bezesporu hlavní stavební prvek vizuálního programovacího jazyka a to je samotný blok. Teď je myšlen návrh programovacího bloku obecně, podle kterého budou odvozeny další konkrétní bloky. Nejdříve jsem uvažoval o možnosti skládat bloky do sebe jako to jde například ve Scratchi nebo v Lego Mindstorms. Nakonec jsem ale zvolil čistě diagramový přístup, jaký je možný také v Lego Mindstorms a tedy, že bloky jsou za sebe připojovány spojovací čarou. Každý blok má připojení na předchozí blok (elipsa úplně vlevo) a připojení na následující blok (elipsa úplně vpravo). Jediná výjimka je v blocích vyjadřujících hodnotu (číslo, boolean, text), které mají jen výstup a jsou připojeny k blokům, které mohou nést jejich hodnotu. Takto se-

řazené bloky reprezentují pořadí příkazů programu. Jako první se vychází ze startovního bloku, který má jen připojení na následující blok.



Obrázek 7: Ukázka designu obecného bloku, který může mít vstupní parametry (elipsy vlevo dole) a právě jeden výstup nebo vstup

Každý blok (kromě startovního a speciálních bloků, které budou popsány dále) se skládá z hlavičky, těla, případných vstupů a jednoho výstupu. V hlavičce může být ikonka daného bloku, a název bloku. Pokud jde například o blok reprezentující operaci s proměnnou nebo operaci s polem, je zde napsán název proměnné, resp. pole. Hlavička má také specifickou barvu bloku, aby se uživatel lépe orientoval ve výsledném programu.

Tělo bloku má vždy stejnou barvu a jsou v něm informace o konkrétním bloku. Tyto informace se liší na základě typu bloku. Například u bloku vyjadřující hodnotu textu má v těle zkrácený text, aby bylo na první pohled jasné, jaký text blok vyjadřuje. Celý text si může uživatel přečíst nebo změnit v pravé části uživatelského rozhraní, jak jsme si popsali dříve.

Případné vstupy (například u bloku operace sčítání to mohou být operandy, atd.) jsou vyjádřené jako elipsovité připojení v levé spodní části a jsou sem přiváděny hodnoty jiných bloků. Mohou sem být přivedeny přímo hodnotové bloky nebo bloky, které nesou hodnotu (proměnná, výsledek jiné operace nebo prvek z pole).

Poslední částí je jediný vstup nebo výstup v pravé dolní části bloku a bývá obvykle zobrazen větší elipsou, než jsou ostatní připojení. V případě operace jde o výstup (u sčítání dva sečtené operandy, atd.), v případě proměnné může jít o vstup, když do této proměnné chci zapsat nějakou hodnotu nebo o výstup, pokud z této proměnné chci číst.

Velikost bloků jsem volil tak, aby byly dostatečně veliké na to, aby se tam vešlo dostatečně informací a byly dobře čitelné na první pohled. Ovšem také jsem nevolil bloky zase moc velké, aby nebylo potřeba neustále plátno scrollovat, když by se stavěl větší program. Ovšem nejlépe výsledná aplikace vypadá na dostatečně velkých monitorech.

6 Backend systému

Jak už bylo řečeno, celá aplikace je rozdělena na části klient a server. Na klientovi, který je spuštěn v internetovém prohlížeči probíhá programování přidáváním a spojováním programovacích bloků na hlavní plátno. Samotný překlad a kontrola programu se ale odehrává na serverové části programu.

6.1 Překladač a interpret

Nejdříve ze všeho si popíšeme základní strukturu překladače, interpretu a jak budou jednotlivé části využity na serveru. V této práci je úplně vynechána část Lexikálního analyzátoru, který normálně slouží pro rozkouskování textu na jednotlivé tokeny[2], které jsou pak využity pro syntaktickou analýzu a také samotná syntaktická analýza má zde menší roli, než u klasických překladačů překládajících jazyk ze zdrojového textu. Je to z toho důvodu, že programové bloky, ve kterých se programuje vstupují do tohoto procesu už jako komplexní celky, které mají předem daná pravidla a usnadňují prvotní fáze překladu.

Překladač jako takový je program, který zpracovává vstupní text napsaný v určitém jazyce, kterému se říká zdrojový jazyk. Překladače pak můžeme rozdělit na dvě hlavní skupiny a to na *kompilační* překladače (assemblery a kompilátory) a *interpretační* překladače (interprety).

Kompilační překladač čte zdrojový program napsaný ve zdrojovém jazyce a překládá ho (transformuje) na ekvivalentní cílový program zapsaný v cílovém jazyce. Naproti tomu *Interpretační překladač* čte zdrojový program a analyzuje jej stejně jako u kompilačního programu. Výsledkem není program v cílovém jazyce, ale přímo provádění příkazů, které byly zapsány ve zdrojovém jazyce. Tomuto procesu se říká interpretace a backend programu VizProg je právě interpretační překladač.

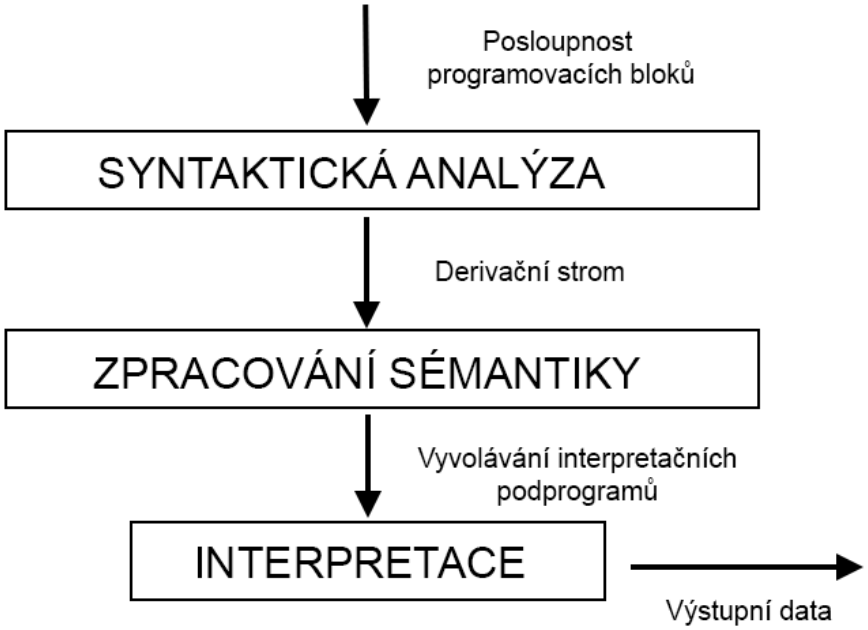
6.1.1 Syntaktická analýza

Jedná se o srdce celého překladače. V klasických překladačích, ať už kompilačních nebo interpretačních do syntaktické analýzy vstupují symboly programu (lexémy). V našem případě nemáme lexémy, ale programovací bloky. Tyto bloky na rozdíl od lexémů jsou už komplexnějším celkem a mají i nějakou syntaxi již z klienta a proto má zde syntaktická analýza mnohem menší roli, než u klasických překladačů. Výstupem syntaktické analýzy je informace o syntaktické struktuře programu nebo signalizace chyby.

[2]Jde o lexikální symboly programu (hodnoty, klíčová slova, omezovače). Všechny lexikální symboly převádí lexikální analyzátor do vnitřní formy, tedy do celočíselné reprezentace, se kterými dále pracuje syntaktický analyzátor [4]

6.1.2 Zpracování sémantiky

Při zpracování sémantiky se vytváří program ve vnitřní formě. Tato vnitřní forma potom slouží jako vstup pro náš interpret. Má za úkol typovou kontrolu nebo jestli jsou všechny použité proměnné nebo pole deklarovány. Tuto část spouští syntaktická analýza a jedná se tedy o syntaxí řízení překlad. [4]



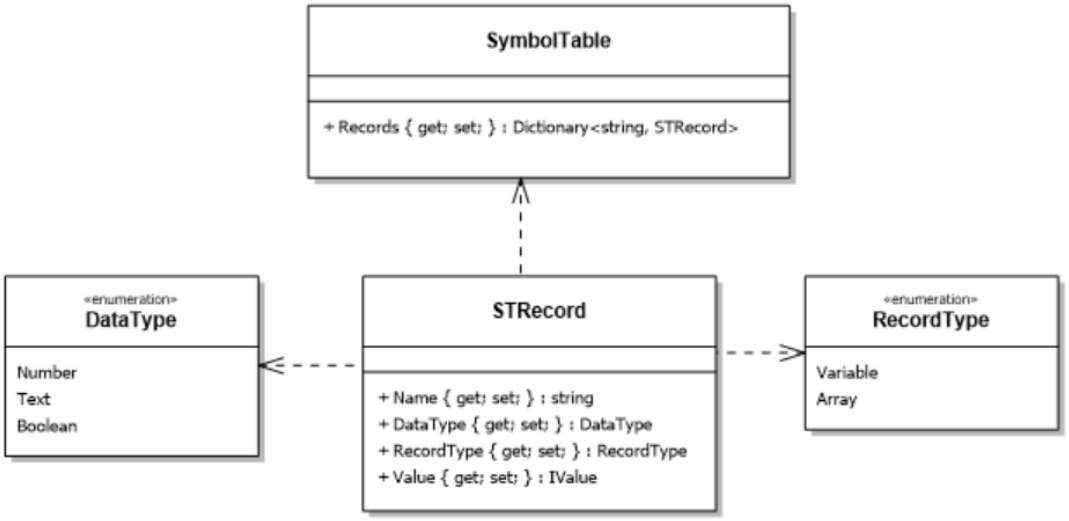
Obrázek 8: Struktura interpretačního překladače [4]

6.1.3 Proces deklarace a tabulka symbolů

Jednou z nejdůležitějších prací překladače je proces deklarace proměnných, polí, funkcí, atd. „Deklarace v programovacích jazycích slouží k pojmenování objektů používaných v programu a k definování jejich vlastností. Překladač, který deklaraci zpracovává, musí uložit informace o vlastnostech objektu do své interní datové struktury nazývané tabulka symbolů. Při pozdějších odkazech na objekty pak využívá údajů z tabulky symbolů a případně ji i dále doplňuje.“ [4]

Tabulka symbolů je důležitou datovou strukturou překladače, která umožňuje provádět kontextové kontroly a uchovávat informace o objektech pro pozdější použití. V našem případě půjde strukturu typu *Dictionary*, která uchovává dvojice typu řetězec (pro název) a samotný objekt záznamu tabulky symbolů. Záznam tabulky symbolu obsahuje název objektu (identifikátor proměnné nebo pole), datový typ (text, číslo, boolean), typ záznamu (proměnná, pole) a hodnotu vyjádřenou třídou implementující rozhraní *IValue*, které bude podrobněji popsáno později. Do tabulky symbolů se dají vkládat nové záznamy, upravovat je nebo

mazat. Slouží pak i sémantickému analyzátoru při výběru hodnoty z tabulky nebo kontrole, jestli už byla proměnná, resp. pole deklarováno.



Obrázek 9: Diagram tříd, které dohromady tvoří tabulku symbolů.

6.1.4 Proces interpretace

V samotném procesu interpretace má již překladač k dispozici program ve vnitřní formě zkontrolovaný syntaktickou a sémantickou analýzou. Překladač poté bere jednotlivé bloky programu a provádí jejich činnost. Pokud se při interpretaci vyskytne chyba, je interpretace zastavena a chyba zobrazena uživateli na klientovi. Pokud interpretace proběhla správně a byl v programu přítomen blok výstupu, odešlou se všechny výsledky výstupů klientovi. Podrobný popis komunikace mezi klientem a serverem je popsán v podkapitole 7.4.

6.2 Technologie ASP.NET Core

Technologicky je backend celé aplikace řešen jako webová aplikace vytvořený ve webovém frameworku ASP.NET Core. ASP.NET aplikace se programuje v jazyce C# a knihovny ASP.NET frameworku obsahuje již řešení nejrůznějších problémů, které se ve webových technologiích vyskytují. Jde například o základní zabezpečení, práce s databází, správa formulářů, atd. Technologie je založena na architektuře klient-server, která tu byla již zmíněna. Jde tedy o program, jehož výstupem je HTML. [21]

6.2.1 Jazyk C#

Jde o vysokoúrovňový objektově orientovaný jazyk od firmy Microsoft. V době psaní této práce je nejnovější verze 7. Jazyk byl primárně vytvořen jen na programování pod platformou .NET, která byla závislá na operačních systémech Windows. Později přišel projekt Mono, což je open source implementace .NET Frameworku. [18] V roce 2016 byla vydána oficiální open source verze .NET platformy nazvaná .NET Core, která je podporována téměř na všech operačních systémech. [9] Jazyk C# patří také mezi jazyky s virtuálním strojem, což znamená, že kód je nejdříve přeložen do tzv. mezikódu nazvaného CIL (Common Intermediate Language). Tento kód je potom interpretován pomocí CLR - Common Language Runtime. [22]

6.2.2 Platforma .NET Core

Jak už jsem psal výše, platforma .NET Core je oficiální open source verze platformy .NET od firmy Microsoft. Verze 1.0 byla vydána 27. 6. 2016 [1] a již v této době se připravuje verze 2.0. Celá platforma .NET Core se skládá z několika částí. Jde o **.NET Core Framework**, který obsahuje knihovny nazvané CoreFX. Ty dále obsahují třídy pro kolekce, souborové systémy, konzole, XML, asynchronní funkce a další. Další částí je **.NET Core Runtime** a ta obsahuje zase garbage collector[3], základní .NET datové typy další low-level třídy. Poslední částí je **.NET Compiler Platform („Roslyn“)**, která obsahuje open-source kompilery s bohatým API pro analýzu kódu pro C# a Visual Basic. [9] Pro komunikaci s klientem ještě využívám v rámci .NET Core aplikace framework Web API, který poskytuje knihovny a další nástroje pro snadné vytvoření RESTful[4] aplikací na platformě .NET Core.

6.3 Rozdělení programu

Celá aplikace, která se stará o překlad a interpretaci je rozdělená na čtyři logické celky (projekty), které spolu navzájem komunikují a zajišťují správné fungování VizProgu. První částí je **Core**, která obsahuje předpisy jednotlivých programovacích bloků a vymezuje tak i hodnoty, kterých můžou nabývat. Neaplikuje žádnou aplikační logiku. Druhou částí je **Analysis**, která obsahuje aplikační logiku pro syntaktickou a sémantickou analýzu. Používá třídy z Core části. Předposlední důležitou částí je **Interpret**, která obsahuje funkce na samotnou interpretaci celého programu a stará se o možný výstup. Ke své funkci využívá funkce z části Analysis. A konečně poslední část je **Web API**, což je ASP.NET Core aplikace, která běží na serveru a poskytuje API na přijímání a odesílání zpráv mezi

[3] Je to způsob automatické správy paměti, kdy je uvolňována paměť, kterou zabírají objekty, které již nejsou potřeba. Využívají ho i další jazyky, jako například Java, Python, C++, atd. [29]

[4] „Representational State Transfer - je architektura, která umožňuje přistupovat k datům na určitém místě pomocí standardních metod HTTP. [16]“

serverem a klientem. Stará se také o převod reprezentace programu z klienta ve formátu JSON na bloky, se kterými dokáže pracovat knihovna Interpret. Ke své funkci využívá všechny ostatní části programu. Každou část teď krátce popíši a zaměřím se jen na ty nejzajímavější a nejdůležitější funkce, které daná část poskytuje.

6.3.1 Core

Knihovna Core obsahuje předpisy tříd a rozhraní, se kterými dále pracuje analyzátor a interpret. Třídy jsou děleny podle kategorií, podobně jako u klienta. Na všechny bloky, které se vyskytují na klientovi zde najdeme její předpis v jazyce C# na vyšší úrovni abstrakce. Všechny bloky ale mají společného předka a to je třída *Block*, která implementuje rozhraní *IBlock*. Tato třída nese jediné dvě informace a to je identifikátor Id datového typu string a příznak o tom, jestli byl tento blok již interpretován.

Mezi nejpoužívanější předpisy tříd patří právě hodnotové třídy, které mohou nést hodnotu (text, číslo, boolean) a implementují důležité rozhraní ***IValueBlock***. Každý blok, který implementuje toto rozhraní může nést nějakou hodnotu a to je podstatná většina bloků, které se ve VizProgu objevují. Toto rozhraní má dva povinné parametry a to je datový typ, který je dán enumerátorem. Ten obsahuje hodnoty Text, Number, Boolean a NotSet (pro nové proměnné, kterým ještě nebyla přiřazena hodnota analyzátorem) a druhá je funkce *GetValueInstance()*, která slouží pro rekurzivní získání surové hodnoty, kterou daný blok nese. Toto rozhraní implementují i bloky operací, pole nebo proměnné.

Jeden z nejstěžejnějších a nejpoužívanějších bloků je třída pro předpis proměnné *VariableBlock*, který implementuje také rozhraní *IValueBlock* a může tak nést hodnotu. Třída obsahuje atribut název proměnné (pole) pro jednoznačnou identifikaci. Nemohou existovat dvě proměnné nebo pole se stejným názvem. Dále obsahuje určení, o jaký typ proměnné jde. V našem případě to může být buď klasická proměnná nebo pole. Pole se zde ukládá stejně jako proměnná do tabulky symbolů. Dále obsahuje určení datového typu kvůli implementaci rozhraní *IValueBlock*, mód proměnné (může být buď mód zápis nebo čtení), hodnota proměnné jako vlastnost implementující rozhraní *IValueBlock* a záznam tabulky symbolů *STRecord*. Vlastnost hodnota proměnné VariableValue může být jiná, než je hodnota vlastnosti STRecord, protože hodnota v STRecord je vždy surová hodnota, buď text, číslo nebo boolean, a nic jiného. Hodnota proměnné může být výsledek aritmetické operace apod. Pokud je proměnná v režimu zápisu, tak je potom uložena přímo surová hodnota z vlastnosti hodnota proměnné VariableValue se uloží do záznamu STRecord a aktualizuje se záznam v tabulce symbolů.

Dalšími zajímavými předpisy tříd je cyklus (*LoopBlock* a podmínka *ConditionBlock*. Obě tyto třídy mají vlastnost Condition, což je objekt implementující rozhraní *IValueBlock*. Dále mají tělo, které je reprezentováno jako seznam objektů implementujících rozhraní *IBlock*. Podmínka má jedno tělo pro případ, že byla podmínka splněna, které musí obsahovat alespoň jeden blok v seznamu (IfBody) a druhé tělo je pro případ, že podmínka nebyla splněna, které je nepovinné. Právě

výsledek podmínky, která musí být datového typu boolean, rozhodne potom v interpretu o tom, které tělo se nakonec vykoná. Blok cyklu má jen jedno tělo, které se opakuje pořád dokola, pokud podmínka tohoto bloku je stále platná. VizProg pro cyklus také obsahuje bloky *BreakBlock* a *ContinueBlock*, které zastupují stejnou funkci jako v kterémkoliv jiném programovacím jazyce, tedy přerušení cyklu a skok znovu na začátek cyklu a ověření podmínky.

Knihovna **Core** obsahuje kromě předpisů tříd a rozhraní bloků také třídy pro výjimky, které mohou při překladu nastat nebo předpis pro tabulku symbolů a její záznamy, jak bylo ukázáno na obrázku 9.

6.3.2 Analysis

Knihovna **Analysis** zajišťuje správnou syntaktickou a sémantickou analýzu programu vytvořeného ze tříd, které poskytuje knihovna **Core**. Funkce zajišťující sémantickou kontrolu pracují pouze s tabulkou symbolů. Jde o funkce statické třídy *SymbolTableHelper* a mají za úkol například deklarovat proměnnou nebo pole, pokud ještě neexistuje v tabulce symbolů nebo tento záznam v tabulce symbolů upravovat. Proměnná se deklaruje tak, se vloží na plátno proměnná v režimu zápisu a definuje se její název, který ještě nebyl v programu použit. Potom je její vlastnost *VariableValue* převzata jako čistá hodnota buď jako text, číslo nebo boolean a uložena do nově vzniklého záznamu *STRecord*, která ukládá název proměnné, datový typ, typ záznamu (proměnná nebo pole) a samotnou hodnotu. Tento záznam je potom uložen do globální tabulky symbolů, kterou využívá celý program. Třída dále obsahuje funkce pro nalezení záznamu v tabulce symbolů nebo aktualizace hodnoty v záznamu tabulky symbolů. Všechny tyto funkce jsou volány z funkcí syntaktické analýzy, které na jejich výsledky patřičně reagují jako například na neexistující tabulku symbolů atd.

Funkce pro syntaktickou analýzu zajišťují kontrolu podmínek na základě překládaného bloku, u operací je to správný datový typ (například sčítat se mohou jen dvě čísla a u logického násobení musí být oba operandy typu boolean, atd.) Základní funkce je *CheckSyntax()*, jejíž jediným argumentem je objekt implementující rozhraní *IBlock*. Uvnitř této metody se na základě druhu bloku spustí příslušná funkce pro kontrolu syntaxe. Pokud jde o operaci, je spuštěna funkce pro konkrétní operaci a zároveň je tato operace vykonána v tom smyslu, že výsledek operace se uloží do *Output* vlastnosti bloku operace. Takto má potom interpret k dispozici přímo výsledek již od syntaktické analýzy. Při kontrole syntaxe proměnné jsou využity funkce ze sémantické analýzy pracující s tabulkou symbolů. Důležitá je také funkce *GetValue()*, která má parametr objekt implementující rozhraní *IValueBlock*. Tato funkce se rekurzivně volá podle druhu bloku až narazí na surovou hodnotu buď datového typu text, číslo nebo boolean. U proměnné ještě záleží na tom, jestli je v režimu zápisu nebo čtení. Pokud je v režimu zápisu, bere jako hodnotu vlastnost VariableValue a pokud je v režimu čtení, bere záznam z tabulky symbolů, který ovšem musí existovat, jinak dojde k chybě. U bloku podmínky a cyklu ještě syntaktická analýza zjišťuje, jestli je definována podmínka a jestli je typu boolean nebo jestli je přítomné tělo podmínky a tělo

cyklu.

6.3.3 Interpret

Spojením výše popsaných částí je knihovna **Interpret**. Ta obsahuje funkce pro interpretaci *RunProgram()* a debugging programu *DebugProgram()* a obsahují vstupní argument seznam bloků, které tvoří program. Interpratace jednoho bloku probíhá tak, že je zavolána příslušná funkce pro interpretaci konkrétního typu bloku a po úspěšné interpretaci se nastaví příznak Interpreted na hodnotu true. Například funkce pro interpretaci cyklu v sobě obsahuje cyklus while, který pořád dokola interpretuje bloky v jeho těle, dokud podmínka bloku cyklu nabývá pravdivé hodnoty. Tato podmínka je vždycky na konci cyklu znovu přepočítána. Stejně tak to funguje i u podmínky, kde se na základě podmínky provede buď tělo pro splnění podmínky nebo tělo pro nesplněnou podmínku. Pro interpretaci proměnné používá interpret funkce syntaktické a sémantické analýzy, které pracují s tabulkou symbolů. Stejně tak pracuje i s polem a s operacemi s polem a pro interpretaci operace volá funkce syntaktické analýzy, aby získal výsledek operace. Interpret obsahuje seznam řetězců pro ukládání výstupů, pokud se objeví v programu textový výstup a ukládá také, jaký blok je zrovna zpracováván a jestli je už zpracovaný. Tyto informace pak používá pro odesílání informací zpět ke klientovi.

Funkce *DebugProgram()* pro spuštění ladění programu má ještě jeden vstupní argument navíc a to je číselná hodnota *finalStepCount*. Toto číslo určuje po jaký krok se má program interpretovat. Po každém interpretovaném bloku od začátku programu se inkrementuje čítač interpretovaných bloků *debugStep*. Vždycky před interpretací nového bloku (platí i v tělech podmínek nebo cyklů) se zkontroluje, jestli se už počet interpretovaných bloků nerovná konečnému počtu kroků, po který se má program interpretovat. Pokud se rovná, skončí interpret svoji činnost a odešle stav programu takový, jaký byl do tohoto kroku zpět klientovi. Právě klient určuje, jak velká je hodnota *finalStepCount* a může ji buď zvětšovat nebo zmenšovat, aby viděl, jaký je stav programu v určitém kroku. O tom, jak může klient řídit ladění programu najdete v uživatelské příručce pod kapitolou 8.4 Ladění programu.

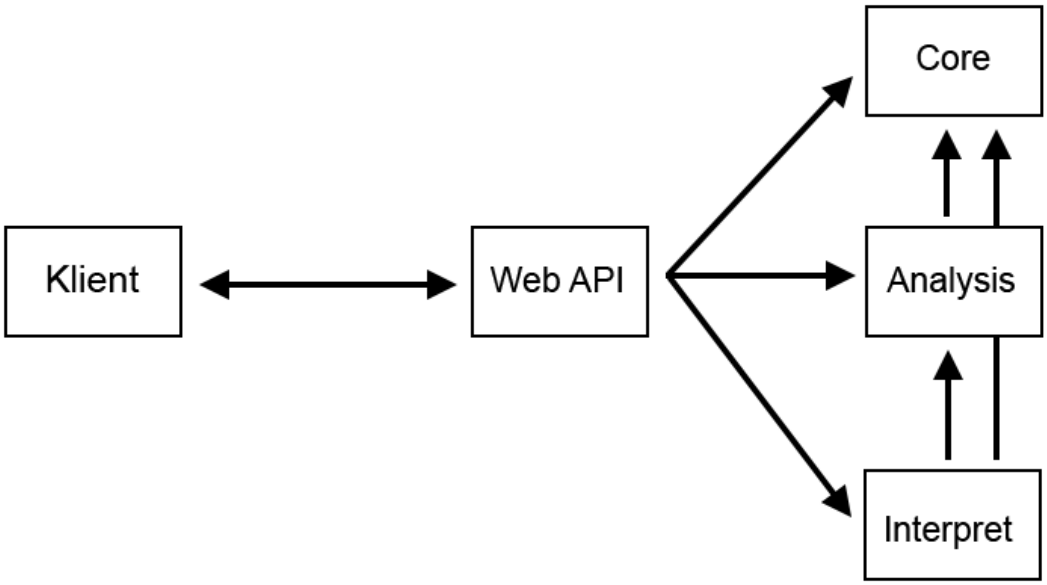
6.3.4 Web API

Poslední částí překladače jazyka VizProg je ASP.NET Core aplikace **Web API** s využitím knihoven Web API .NET Frameworku. Je to hlavní komunikační prostředník mezi klientem a interpretem. Klient posílá zprávy Web API aplikaci a ta na ně reaguje také zasíláním zpráv.

Aplikace obsahuje jediný controller s názvem *InterpretController* a obsahuje metody na spuštění nebo ladění programu. Funkce *Run()*, která reaguje na příchozí data HTTP metodou POST má jako vstupní argument seznam JSON objektů *JObject*. Každý blok je již od klienta serializován ve formátu JSON kterému klient rozumí. Web API aplikace obsahuje třídu *BlocksFactory*, jejíž funkce

zajistí převod bloků ve formátu JSON na objekty bloků z knihovny Core, kterým rozumí interpret. Bloky jsou převáděny podle hodnoty *blockType*, kterou obsahuje každý blok ve formátu JSON.

Třída *BlocksFactory* vytvoří jen prázdné reprezentace všech bloků, které byly poslány od klienta ve formátu JSON. Je ještě potřeba je naplnit a správně pospojovat. Společně s popisy bloků ještě klient odešle i informaci o tom, jaké bloky mají jako vstupy nebo výstupy. Tato informace je definována jen identifikátorem daného objektu. Například když má operace k sobě připojeny další dva bloky jako operandy, obsahuje blok operace ve formátu JSON hodnoty „operandA“, „operandB“. Tyto hodnoty obsahují identifikátory bloků, které představují operandy. K tomu slouží metody třídy *BuildFactory*, která v tomto případě najde blok operace, potom najde oba bloky, které představují operandy a na místo operandů vloží tyto bloky v reprezentaci, kterým rozumí interpret. Takto zpracovává například i tělo cyklu nebo podmínek, kdy je u těchto bloků ve formátu JSON určený identifikátor jen začátečního bloku a všechny ostatní, až po konečný (nebo v případě cyklu je ještě určený, který blok je konečný, aby se uzavřel cyklus) vkládá do daného těla. Může také upozornit na chybu v překladu, když některé povinné vlastnosti chybí (například operand operace, název proměnné nebo tělo při splnění podmínky). Výsledné bloky se potom ukládají do seznamu, který tvoří připravený program pro interpretaci. V této reprezentaci je tady předán interpretu a začne samotná interpretace.



Obrázek 10: Struktura backendu aplikace VizProg a komunikace s klientem

7 Frontend systému

Klient aplikace je úplně oddělená aplikace, která umožňuje uživateli programovat program v jazyce VizProg pomocí programovacích bloků v programovacím prostředí. Jde znovu o webovou aplikaci napsanou v jazyce JavaScript. Program se poté odešle serveru ve formátu JSON a dojde k jeho interpretaci, jak jsme si popsali v kapitole 6. Jazyk JavaScript, ve kterém jsem napsal frontend je implementací ECMAScript[5] verze 6 s využitím knihoven React a Redux. Funkčnost ECMAScript 6 vlastností u starších prohlížečů je zajištěno pomocí kompilátoru Babel, který převede nové funkce jazyka JavaScript do staršího formátu.[24]

7.1 Knihovna React

Knihovna React je open-source JavaScriptový framework pro vytváření uživatelských rozhraní. Je udržována organizacemi jako je Facebook, Instagram a čím dál větší komunitou individuálních vývojářů a dalšími korporacemi. Používají ho veliké organizace jako je již zmíněný Facebook, Netflix, Imgur nebo třeba Walmart. React dovoluje vytvořit velké webové aplikace pomocí tzv. komponent, která si každá sama udržuje svůj stav, který může měnit, aniž by musela být znovu načtena celá stránka. Jde o úplně jiný způsob vytváření uživatelského rozhraní webových aplikací. V návrhovém vzoru MVC (Model-View-Controller) koresponduje React jen s View. Může být použit v kombinaci s jinými JavaScriptovými knihovnami nebo frameworky s MVC jako je například ASP.NET MVC nebo AngularJS. [26]

7.1.1 Komponenta

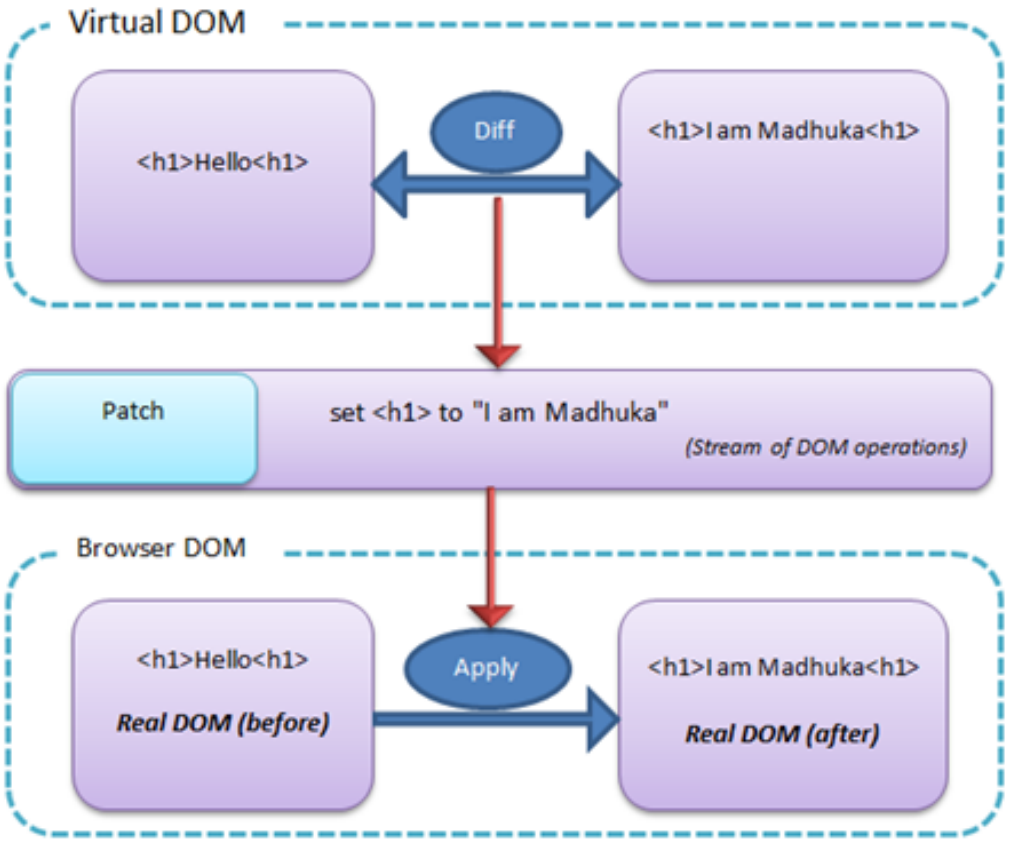
Stěžejní částí v knihovně React je komponenta, ze kterých se skládá celá aplikace. Komponenta jako taková je z programátorského hlediska jen třída, která dědí od třídy *Component* z knihovny React.

Celé uživatelské rozhraní se tvoří hierarchicky z komponent. V souboru *index.html* je v těle jeden ** ** kontejner s určitým id (například „root“), na který je poté navázána React aplikace a tedy i základní komponenta, která obsahuje ostatní komponenty a tak se tvoří uživatelské rozhraní. V souboru *index.js* je zavolána funkce *render* na Virtuální DOM se dvě vstupními argumenty. První argument je komponenta, která představuje aplikaci a druhý je objekt kontejneru, na který se má React aplikace navázat.

Virtuální DOM je abstrakce klasického DOMu (Document Object Model). Byl vytvořen proto, že spousta JavaScriptových frameworků updatuje DOM vícekrát, než by vůbec měla, čímž dochází k velkému zpomalování. V Reactu existuje pro každý DOM object korespondující „Virtual DOM object“. Jde pouze o tzv.

[5] Jde o scriptovací jazyk normovaný firmou ECMA International. Tento jazyk je používán především na webových stránkách a jeho nejčastěji používaná implementace je právě JavaScript. [8]

lehkou kopii tohoto objektu. Manipulace s Virtuálním DOMem je o mnoho rychlejší, než s klasickým DOMem. Vždycky po updatu celého Virtuálního DOMu dochází k porovnání s klasickým DOMem a updatují se jen ty objekty, které se liší. Až update klasického DOMu zapříčiní změny na obrazovce. [27]



Obrázek 11: Změna ve Virtuálním DOMu a klasickém DOMu.[27]

Komponenty nám dovoluje oddělit UI[6] na nezávislé, znovupoužitelné části a možnost přemýšlet o každé komponentě jako o izolované části aplikace.

Komponenta má tzv. properties (česky vlasnosti, ale budu se držet anglického názvu) a state (stav). Properties jsou jakákoliv pojmenovaná data (čísla, proměnné, objekty), které jsou vloženy do komponenty „zvenčí“, tedy z komponenty, která má s touto komponentou vztah rodič a potomek. Stav si komponenta udržuje sama a na rozdíl od properties ho může měnit jen samotná komponenta. Stav se mění pomocí funkce *setState()*, kam se jako vstupní argument vloží objekt definující dvojice „název“: „hodnota“ stavu komponenty. Properties mohou sloužit například na inicializaci stavu a uvnitř komponenty se pak pracuje už jen

[6]User Interface - uživatelské rozhraní

se stavem. Komponenta také musí implementovat jednu povinnou metodu a to je metoda *render()*. Tato metoda vrací HTML kód, který se má vykreslit tam, kde je tato komponenta použita. Tento HTML kód bývá kombinován i s JavaScript kódem, kde bývá použit právě stav, properties nebo další definované funkce. [6]

Komponenta má také životní cyklus. Průběh životního cyklu je definován v metodách, které se v angličtině nazývají „lifecycle hooks“. Metody mají prefix **will**, které se volají předtím, než se stane nějaká událost nebo **did**, když se tato událost již stala. Metody jsou ještě rozděleny podle toho, jaký typ události obsluhují. Je to **Mounting**, kde jsou tyto metody volány, když je instance komponenty vytvářena a vkládána do Domu. Patří sem:

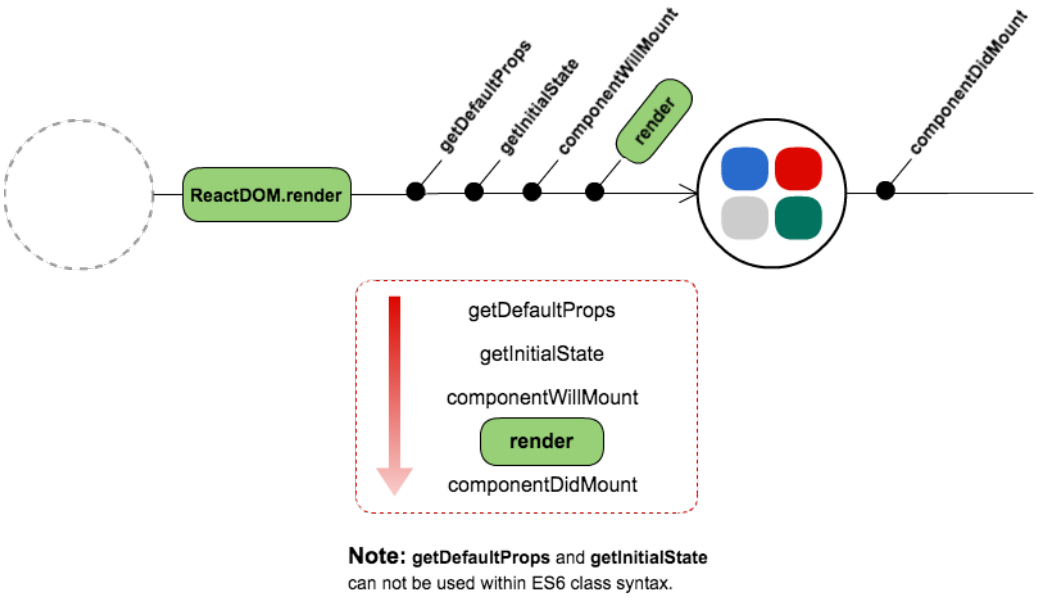
- constructor(),
- componentWillMount(),
- render(),
- componentDidMount().

constructor() je volán ještě než je komponenta mountnutá (přidána do DOMu). Když je implementována, musí být ještě volána metoda *super(props)*, jinak budou properties této komponenty nedefinované. V těle konstruktoru je dobré definovat výchozí stav komponenty, kde mohou být využity i properties.

compnentWillMount() metoda je volána těsně předtím, než je namountována. Volá se ještě před metodou *render()*.

render() je jediná povinná metoda. Definuje podobu komponenty pomocí HTML elementů v kombinaci s JavaScriptem.

componentDidMount() metoda je volána ihned po namountování komponenty. Je dobrá například na načítání dat z jiných knihoven nebo ze vzdálených úložišť. Změna stavu má za následek překreslení komponenty.

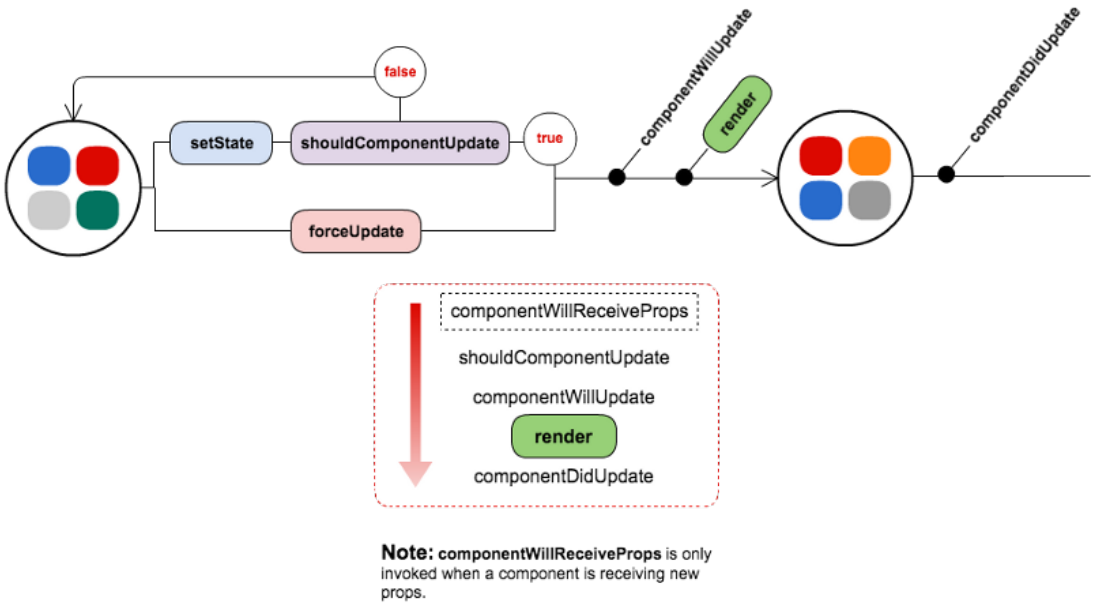


Obrázek 12: Diagram volání metod skupiny Mounting a dopad na životní cyklus komponenty. [5]

Další skupinou metod pro řízení životního cyklu komponenty je **Updating**, kam patří metody:

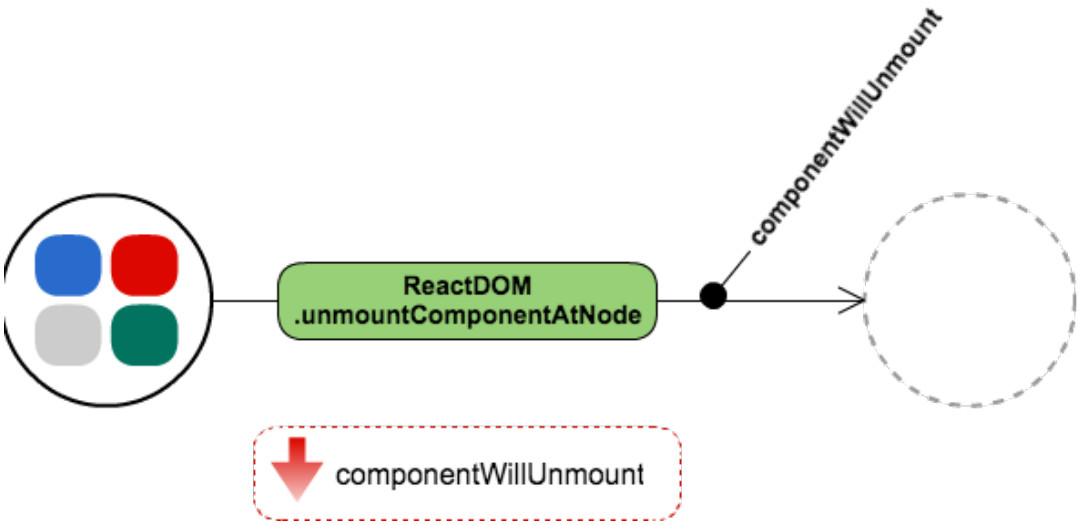
- componentWillReceiveProps(),
- shouldComponentUpdate(),
- componentWillUpdate(),
- render(),
- componentDidUpdate().

Tyto metody už názvem napovídají, kdy se jejich těla vykonají. Z metod v této skupině jsem ve své práci nevyužil žádnou (kromě metody *render()*. Všechno potřebné se dá naimplementovat v metodách skupiny **Mounting**.



Obrázek 13: Diagram volání metod skupiny Updating a dopad na životní cyklus komponenty. [5]

Poslední skupinou je **Unmounting**, kam patří jediná metoda a tou je ***componentWillUnmount()***, která se vykoná těsně předtím, než je komponenta unmountována a zničena. Lze v ní naimplementovat nezbytné uvolnění paměti nebo zavolání metod, které jsou potřebné. [30]



Obrázek 14: Diagram volání metod skupiny Unmounting a dopad na životní cyklus komponenty. [5]

7.2 Knihovna Redux

Redux je stavový kontejner pro JavaScriptové aplikace a jde hezky využít právě v kombinaci s React aplikací, jako je v případě VizProgu. Redux je koncepčně velice jednoduchý a proto je tak lehce a efektivně využitelný. Může být popsán ve třech základních principech.

První princip je **Single source of truth**, neboli „jediný zdroj pravdy“. To znamená, že celý stav aplikace je uložen v objektovém stromu jen na jednom místě a to je tzv. store, o kterém si napíšeme dále.

Druhý princip je **State is read-only**, „stav je jen ke čtení“. Jediná možnost, jak změnit stav je vykonat akci (actions), což je objekt, který popisuje, co se má stát.

Třetí a poslední princip **Changes are made with pure functions**, „změny jsou dány jen funkcemi“, což znamená, že změny stavu probíhají čistě ve funcích, tzv. reducerech, které berou v potaz typ akce a případná data a z těch vytvoří nový stav. Stav není nikdy upravován, ale vždy vzejde nový stav, který je jediný, který zrovna platí.

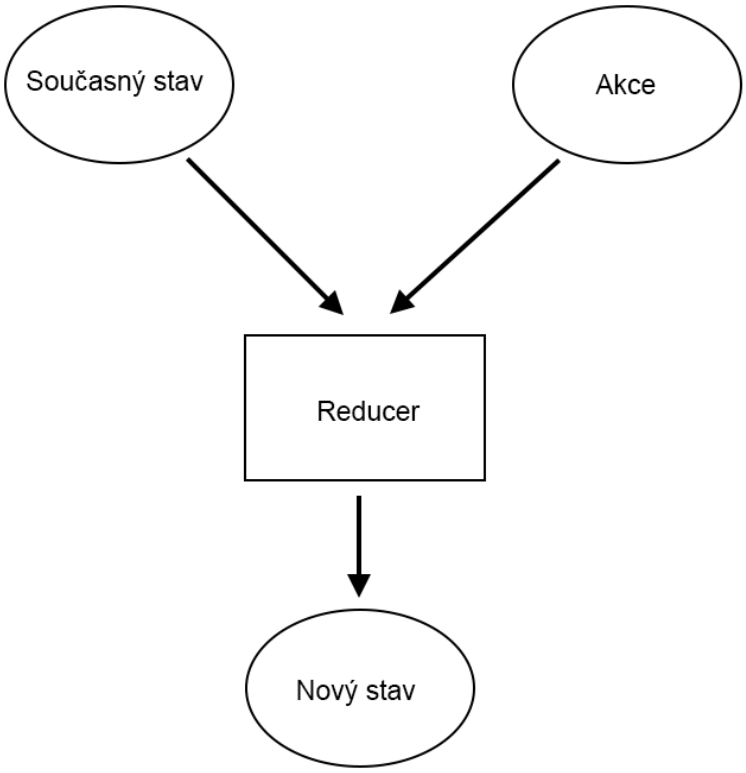
Na těchto třech jednoduchých principech funguje celý Redux a na těch mohou fungovat jak malé, tak i velké aplikace a pořád jsou zachovány jen a pouze tyto tři principy. [33]

7.2.1 Actions

Akce jsou stěžejní prvek Reduxu. Říkají nám, co se má dít a jaká data se popřípadě mají změnit. Jde o klasický JavaScriptový objekt, který má jednu povinnou vlastnost „type“, která musí jednoznačně určit, o jaký typ akce jde. Ostatní vlastnosti mohou specifikovat konkrétní data, která mají určit stav aplikace nebo podle kterých se má blíže specifikovat akce, například id záznamu, co se má smazat, atd.

7.2.2 Reducers

Akce, jak jsme si popsali, definuje fakt, že se „něco stane“, ale nespecifikuje, jak se stav aplikace změní a jak na tuto akci odpoví. K tomu slouží tzv. reducery. Reducer je funkce, která přijímá dva vstupní argumenty **současný stav** a **akci** a vrací **nový stav**. Reducer by měl zůstat čistou funkcí bez „side efektů“, takže by v těle reduceru nemělo být volání jiných API nebo změna routování, modifikace argumentů nebo volání non-pure funkcí jako například *Date.now()* nebo *Math.random()*. Používá jenom čisté výpočty. Výsledek je vždy nový stav, nikdy se nemodifikuje starý stav. Pokud se mají změnit jen některé vlastnosti současného stavu, jsou jeho hodnoty, které mají být zachovány zkopírovány do nového objektu a to stejné platí i když se ve stavu vyskytne seznam. Nejdříve je vytvořena hluboká kopie a potřebné změny (přidání, modifikace nebo mazání) jsou prováděny až v této kopii.



Obrázek 15: Struktura reduceru

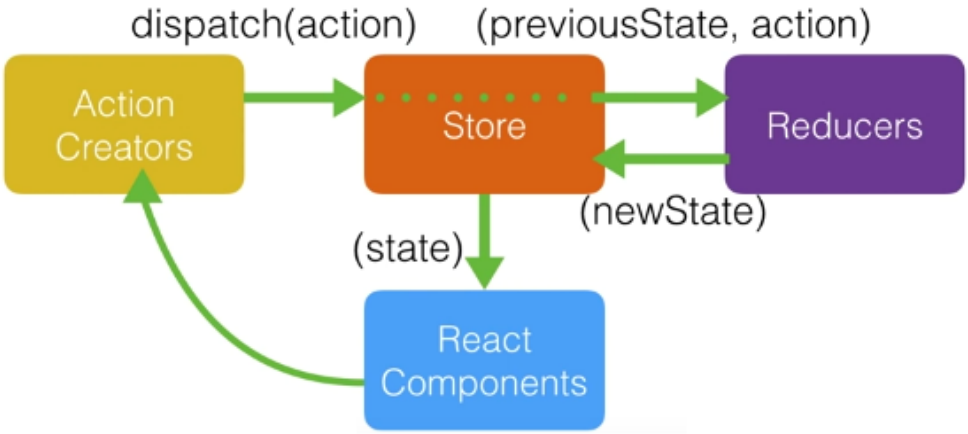
7.2.3 Store

Store je, jak jsem již nastínil v prvním principu funkce Reduxu právě ten „jediný zdroj pravdy“. V celé aplikaci může být jen jeden a přes něj se stav jak čte, nebo pomocí funkcí (2. a 3. princip) modifikuje. V předchozích podkapitolách jsme si definovali **akce**, které reprezentují fakt, že se „něco stane“ a **reducery**, které modifikují stav na základě těchto akcí. Store je objekt, které tyto dvě věci spojuje dohromady a má následující vlastnosti:

- Udržuje stav aplikace,
- Přistupuje ke stavu přes funkci ***getState()***,
- Umožňuje, aby byl stav modifikován pomocí funkce ***dispatch(action)***,
- Registruje listenery pomocí funkce ***subscribe(listener)***,
- Odregistrovává listenery pomocí funkce, kterou vrátí funkce, ***subscribe(listener)***.

Tohle jsou jediné funkce Storu v Redux aplikaci. Je důležité, aby byla vytvořena jen jedna instance Storu v aplikaci. Je možné rozdělit data do více celků pomocí kompozice reducerů, kdy je vytvořeno více reducerů na jednu aplikaci a každý se může starat o část stavu aplikace. Při vytváření Storu se jako vstupní

argument funkci *createStore()* dává reducer (nebo kompozice více reducerů pomocí funkce *combineReducers()*), který, resp. které budou modifikovat stav aplikace. Potom již na Store voláme funkci *dispatch()* se vstupním argumentem akce, kterou chceme vykonat a provede se reakce, která je definována v reduceru, který byl předán při vytváření Storu.



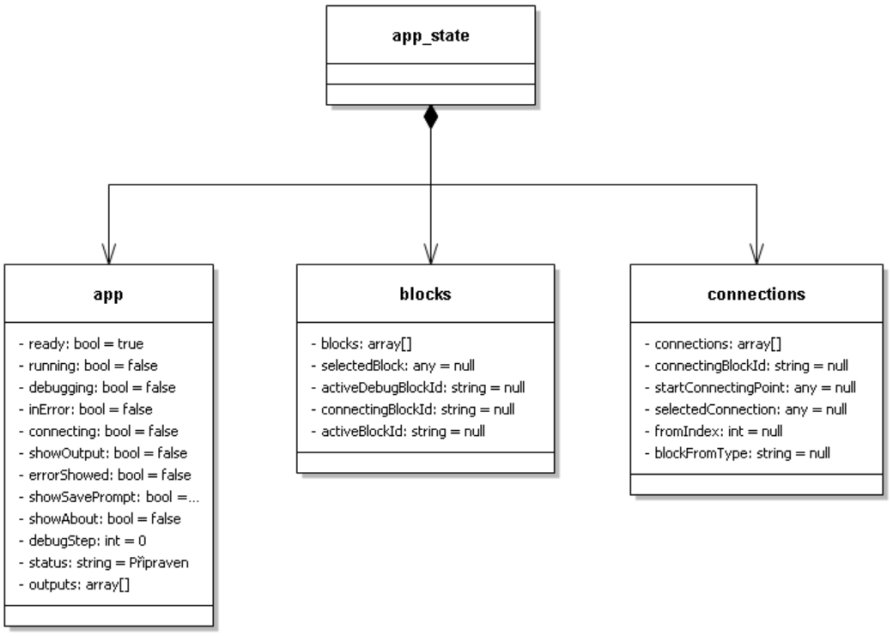
Obrázek 16: Ukázka toku dat v Reduxu s napojením stavu na komponentu v React aplikaci [34]

7.2.4 Propojení Reactu a Reduxu ve VizProg

Propojení Reactu s Reduxem je velice jednoduché. Víme, že může v celé aplikaci existovat jen jedna instance Storu. Store lze mezi komponenty sdílet buď za pomocí dependency injection nebo celý Store předávat jako property do komponenty, což je způsob, který jsem zvolil i já. K objektu Storu se pak dostanu z každé komponenty pomocí *this.props.store* a můžu na něj volat metody, které jsou popsány výše v podkapitole 7.2.3 o Storu. Bloky mohou být také subscribovány na změnu Storu pomocí funkce *subscribe()*, kde se v těle může naimplementovat změna stavu komponenty na základě změny stavu ve Storu. Tento subscribe je dobré volat například v metodě *componentDidMount()*.

Stav celé aplikace VizProg je rozdělen do třech částí a to jsou **app**, **blocks** a **connections**. Reducer celé aplikace je tedy rozdělen do třech stejnojmenných reducerů, které jsou propojeny pomocí funkce *combineReducers()*. Část **app** definuje stav celého uživatelského rozhraní ve smyslu, zda je zrovna ve stavu připojování k serveru, jestli je program v běhu nebo jestli se vyskytla během překladu chyba, atd. Na tyto stavy potom reagují jednotlivé komponenty, které tvoří toto uživatelské rozhraní. Část **blocks** zase definuje stav okolo programovacích bloků. Jeho hlavní částí je právě pole bloků, které jsou na „hlavním plátně“, ale také určuje, jestli je nějaký blok označený nebo jestli se dva bloky spojují, atd. Poslední částí **connections** obsahuje pole spojení, které obsahuje informace o tom,

jaké bloky jsou mezi sebou spojené a z jakého vstupu (má-li jich více) do jakého výstupu a naopak. Výčet všech vlastností stavu aplikace je na obrázku 17.



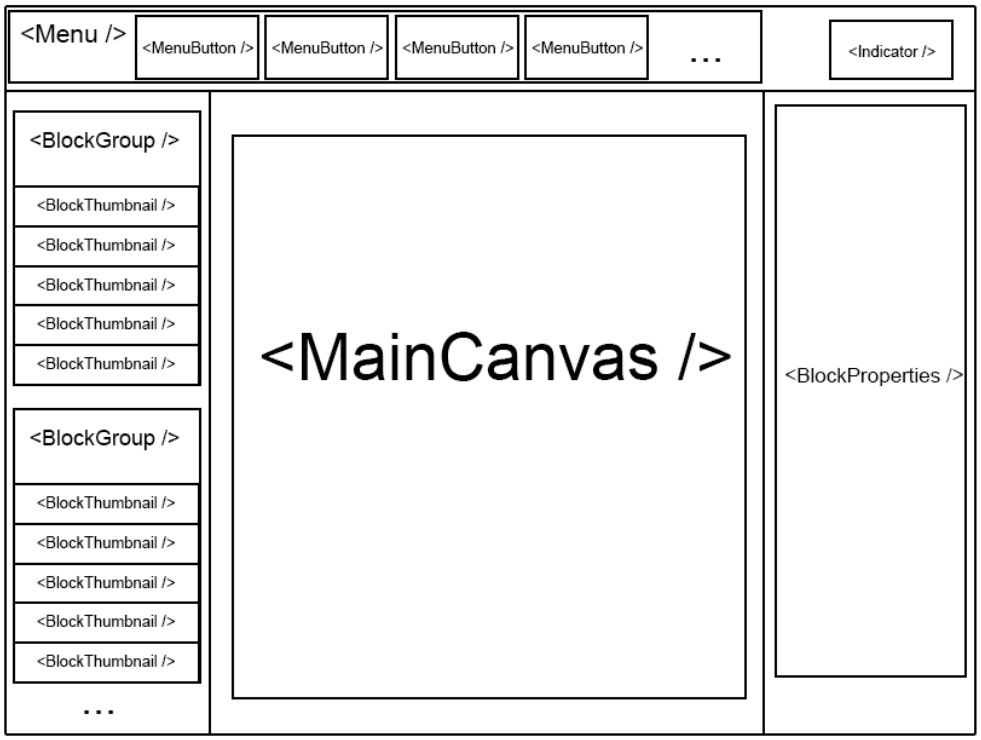
Obrázek 17: Stav celé aplikace rozdělený do třech částí

7.3 Uživatelské rozhraní

Uživatelské rozhraní je rozděleno na několik komponent. Celý stav komponent je určen stavem ze Storu. Stav rozhraní je dán částí stavu **app**. Hlavní komponenty jsou **Menu**, které obsahuje komponenty tlačítek menu **MenuButton** a ještě je nahoře zvlášť komponenta **Indicator**, která určuje barevně stav programu (stala se chyba, resp. nestala se chyba).

Na levé straně každá skupina bloků představuje také jednu komponentu **BlockGroup**, která obsahuje pak komponenty náhledů bloků **BlockThumbnail**, které se dají předávat na plátno. Plátno je také jedna samostatná komponenta **MainCanvas**, kterou si podrobněji popíšeme v podkapitole 7.3.2 o blocích. Na pravé straně se ještě nachází komponenta **BlockProperties** na zobrazování vlastností právě vybraného bloku. Každá z těchto komponent má přístup ke Storu a například stisk některého tlačítka menu zavolá metodu *dispatch()* s určitou akcí na Store, při které se změní stav ve Store, ale i stav komponent, když má změnu stavu subscribnutou na Store pomocí funkce *subscribe()*. Například, když uživatel stiskne tlačítko „Spustit program“, provede se akce typu „RUN_PROGRAM“, na který reducer reaguje změnou vlastností „ready“ na false a „running“ na true v části **app** stavu aplikace a například tlačítko „Spustit program“ má vlastnost spuštění programu jen, když je stav „running“ na false, jinak má vlastnost „Zastavit program“ a má tak změněnou ikonu i funkci, která se spustí při kliknutí.

Na obrázku 18 je popsáno rozložení aplikace pomocí komponent, které odpovídá původnímu rozložení z obrázku 5.



Obrázek 18: Rozložení uživatelského rozhraní seskládaného z komponent.

7.3.1 SVG

V této krátké podkapitole popíši značkovací jazyk SVG a k čemu jsem ho v programu VizProg využil. Zkratka SVG znamená Scalable Vector Graphics a tento jazyk je využíván k definování vektorové grafiky pro web. Definování grafiky je v XML formátu a protože je vektorová, neztrácí svoji kvalitu při zoomování nebo změně velikosti. Každý element a atribut je v SVG zanimovatelný a lze zintegrovat z dalšími W3C standardy jako je DOM nebo XSL. Pomocí SVG lze definovat jednoduché tvary jako je obdélník, elipsa, čára nebo i složitější obrazce pomocí path (cesty). Všechny tyto elementy lze také nastylovat pomocí CSS kaskádových stylů. [32]

Protože je VizProg vizuální programovací jazyk, bylo potřeba vymyslet, jakým způsobem bloky vykreslit, aby nezabíraly moc paměti a vypadaly na jakémkoliv rozlišení pořád stejně. K tomu se nejlépe hodí právě SVG, pomocí něhož React komponenty představující jednotlivé bloky vykreslují svoji podobu a stav. Například u návrhu bloku na obrázku 7 jsou hlavička a tělo bloku prosté obdélníky nastylované pomocí CSS a spojovací elipsy jsou SVG elipsy, které mohou měnit barvu na základě stavu React komponenty. V SVG lze definovat i texty (například pro názvy proměnné nebo název bloku) i vkládat obrázky (ikonky bloků).

7.3.2 Bloky

Hlavní částí vizuálního programovacího jazyka VizProg jsou právě programovací bloky, pomocí nichž se aplikace programuje. Ve webové aplikaci to není nic jiného, než React komponenty. Všechny dědí od React komponenty **BaseBlock** a implementuje základní vlastnosti, které mají všechny bloky společné a to je například pozice, identifikátor, typ bloku a jestli je označen. Dále implementuje chování jako je změna pozice pomocí táhnutí myši, označení bloku nebo spojování bloku. Od této komponenty dědí konkrétní implementace bloků, pro každý programovací blok jedna komponenta. Každá tato komponenta má ještě doplnění stavu ke stavu z **BaseBlock**, například komponenta VariableBlock má ještě stav *variableName* nebo *writeMode*. Dále tyto konkrétní implementace mají definovanou metodu *render()*, která vrací HTML kód s kombinací SVG a JavaScriptu a tak se vykreslí blok.

Všechny bloky jsou uloženy ve Storu v části **blocks** v seznamu blocks. V tomto seznamu jsou objekty se stejným identifikátorem a typem, jaké mají komponenty, které tyto bloky vykreslují. Komponenta **MainCanvas** má za úkol vykreslovat všechny bloky, které jsou v seznamu blocks stavu ze Store. Podle typu a dalších dat vykreslí správnou podobu bloku. Každý blok tedy v aplikaci existuje dvakrát. Jednou jako záznam v seznamu stavu aplikace a jednou jako vykreslená komponenta na hlavním plátně. Když ale měníme stav bloku, ať už změnou pozice nebo měníme název proměnné, atd., volají se akce, které předávají Storu pomocí metody *dispatch()* a tím se změní vlastnosti objektu v seznamu blocks stavu aplikace ve Store a protože mají všechny bloky subscribnutou změnu Store pomocí metody *subscribe()*, kde na základě id komponenty mění svůj stav podle objektu, který s ní koresponduje, je vždy hned tato komponenta překreslena. Díky Virtuálnímu DOMU je toto všechno pěkně rychlé a vždy se překreslí jen ty bloky, které se opravdu změnily.

Spojování bloků je implementováno tak, že komponenta reaguje na stisknutí a držení levého tlačítka myši na spojovací elipse jednoho bloku, táhnutím nad jinou spojovací elipsu bloku a následného puštění tlačítka. Znovu pomocí akcí a aktualizací stavu aplikace části **connections**, která si udržuje seznam spojení mezi bloky jsou tyto spojení pomocí čar v jazyce SVG vykresleny na hlavní plátno pomocí komponenty **MainCanvas**, která prochází všechny spojení v seznamu stavu aplikace. Každé spojení má definováno, z jakého bloku (jeho identifikátor) vychází a do jakého vchází a jaký index spojení (0 pro předchozí blok, 1 pro následující blok, 2 pro vstup, atd.) každého bloku toto spojení má. Takto lze jednoznačně definovat, jakým způsobem jsou bloky spojené.

7.4 Propojení Backendu a Frontendu

Konečně se tedy dostáváme k tomu, jak je Frontend spojen s Backendem aplikace. Program bývá programován na klientovi přidáváním, spojováním a upravováním programovacích bloků do logických celků a poté je celý program odeslán na server, kde dojde k překladu. Pokud překlad proběhne v pořádku, vrátí zpět

ke klientovi na vypsání buď nějaký textový výstup nebo se vrátí jen potvrzení o tom, že interpretace dopadla v pořádku. Pokud ale dojde k chybě, je odeslána chybová hláška a informace o vzniklé chybě včetně identifikátoru bloku, který chybu zapříčinil, zpět ke klientovi a ten na zareaguje změnou stavu aplikace a zobrazení chybové hlášky.

Při spuštění programu na klientovi ještě probíhá vnitřní setřídění programu, než se odešle na server. Jde o funkci **buildProgram()**, který na vstupu bere seznam bloků a seznam připojení ze stavu aplikace ze Storu. Funkce projde všechny bloky, najde k nim všechna příslušná připojení podle toho, o jaký typ bloku jde. Například pro operaci sčítání bude hledat připojení na oba operandy a kam se má uložit výsledek. Takto nově sestavené objekty i s připojeními vloží do nového seznamu, který je poté metodou POST odeslán na server na překlad.

7.4.1 Formát JSON

Všechna data, které si klient a server vyměňují jsou ve formátu JSON (JavaScript Object Notation). Je to odlehčený formát pro výměnu dat. Je jednoduše čitelný jak člověkem, tak i strojově. Existují knihovny, které objekty jednoduše serializují do JSON formátu nebo je naopak mohou deserializovat zpět na objekt. JSON je textový a na jazyce zcela nezávislý formát. Využívá však konvence dobře známé programátorům jazyků rodiny C.

JSON je založen na dvou strukturách. Buď je to kolekce párů název/hodnota, která bývá realizována jako objekt, záznam nebo asociativní pole. Nebo to může být seřazený seznam hodnot, který bývá realizován jako pole, vektor, seznam nebo posloupnost. [12]

V případě VizProgu je program poslaný seřazený seznam objektů, kde každý objekt představuje blok, tedy kolekci párů název/hodnota. Všechny bloky mají vlastnosti blockId a blockType a další hodnoty již záleží na konkrétních blocích. Pokud je spuštěna funkce ladění, posílá se s programem ještě hodnota *debugStep*, aby server věděl, po jaký krok má program interpretovat.

Programu v tomto formátu teď může rozumět i server, který podle něj sestaví program ve formátu pro interpret, jak bylo popsáno v podkapitole 6.3.4 o Web API.

Server může zasílat zprávy zpět klientovi také ve formátu JSON. Buď to je zpráva o chybě, kde objekt v JSON obsahuje popis chyby a identifikátor bloku, který chybu způsobil, nebo je to potvrzovací zpráva o tom, že překlad proběhl v pořádku a pošle se případná kolekce textových výstupů. Server může v případě ladění poslat zpět klientovi tzv. *DebugValues*, což je kolekce objektů, které obsahují identifikátor bloku a jeho hodnotu, kterou má v určitém kroku. Takto může klient zobrazit hodnotu bloku v určitém kroku.

8 Uživatelská příručka

V této kapitole popíši program VizProg pro uživatele, který nemá žádné zkušenosti s programováním a už vůbec ne s programem VizProg. Bude popsáno rozložení uživatelského rozhraní, struktura a funkce jednotlivých bloků, jak se program spouští, ladí, ukládá nebo načítá. Jsou zde uvedeny části z oficiální nápovědy, která je k dispozici společně s vývojovým prostředním. V celé nápovědě jsou také do podrobna popsány všechny programovací bloky, příklady kusů programů, operace s polem, atd. [35].

8.1 Co je VizProg

VizProg je vizuální programovací jazyk. Vizuální znamená, že zdrojový kód není psán textově (například jazyky C#, Java, C, Python, atd.), ale je tvořen vizuálními prvky, které dohromady tvoří logiku programu. V případě VizProgu to jsou programové bloky, z nichž každý má svoji funkcionalitu. Tyto programové bloky korespondují s konstrukcemi většiny programovacích jazyků. V každé instanci VizProgu lze naprogramovat právě jeden program a nelze vytvářet podprogramy.

Program vytvořený ve VizProgu má podobu grafického diagramu, kde jsou všechny funkční bloky spojeny za sebou a mají vlastní vstupy, resp. výstupy. Takto vytvořený program je po spuštění odeslán na server, kde se vyhodnotí a pokud je syntakticky a sémanticky v pořádku, spustí se. V opačném případě pošle zpět chybu a označí blok, který chybu způsobil a který je potřeba opravit, aby program správně fungoval.

8.2 Vývojové prostředí

Vývojové prostředí má podobu webové aplikace. Správně aplikace běží jen na prohlížeči Google Chrome nebo Opera, protože některé části aplikace využívají vlastnosti, které zatím podporují jen tyto dva prohlížeče. Pokud tedy používáte nějaký jiný prohlížeč, vývojové prostředí VizProg Vám v něm nebude fungovat správně. Ve vývojovém prostředí lze naprogramovat libovolný program a následně ho spustit nebo ladit (krokovat).

V horní části programu je hlavní menu, kde se ovládá vývojové prostředí, popřípadě program. Zleva jsou to tlačítka:

Obrázek 19: Tlačítka horního menu vývojového prostředí VizProg. Zleva Nový program, Otevřít program, Uložit program, Spustit program, Ladit program, Nápověda a O programu

Tlačítko „Nový program“ slouží na vymazání stávajícího projektu bez uložení a můžete tak ihned začít s novým programem. Program může být také uložen pomocí tlačítka „Uložit program“, kdy Vás aplikace vyzve k pojmenování souboru, do kterého chcete program uložit. Soubor má příponu „.vizprog“ a je z aplikace stažen jako soubor. K nahrání tohoto souboru slouží tlačítko „Otevřít program“, kdy Vás aplikace vyzve k nalezení cesty souboru s příponou „.vizprog“.

Další tlačítko slouží na spuštění programu „Spustit program“ a dojde při něm k pokusu o připojení se k serveru a následné spuštění programu. Pokud je program bez chyb, spustí se, jinak se program vypne a zobrazí se chybová hláška. Tlačítko pro ladění programu „Ladit program“ program spustí, ale jen po krocích. Můžete tak program krokovat jak dopředu, tak dozadu a sledovat tak stav aplikace v průběhu spuštění.

Tlačítko „Nápověda“ spustí v novém okně nápovědu a poslední tlačítko „O programu“ zobrazí modální okno se základními informacemi o programu.

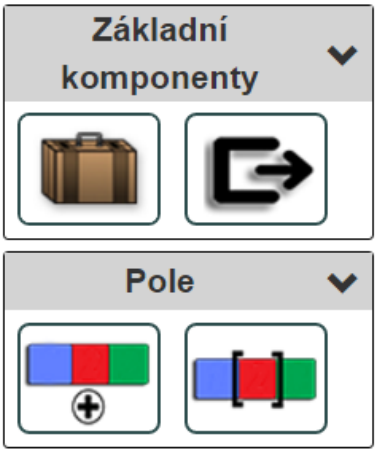
Obrázek 20: Barevný indikátor stavu programu. Zelená je pro status, že je vše v pořádku a červená signalizuje chybu

V pravé horní části prostředí se ještě nachází indikátor programu, který svou barvou vyjadřuje, jestli je vše v pořádku (zelená barva) nebo jestli došlo k chybě (červená barva). Po najetí myši se jako titulek zobrazí případná chybová hláška.

8.2.1 Levé menu

V levé části prostředí jsou programovací bloky. Každý je definován nějakou ikonkou, která vyjadřuje jeho funkcionalitu. Přidání bloku do programu se provede jednoduchým přetažením bloku (podržením levého tlačítka myši a táhnutím) na prostřední část prostředí (programovou část).

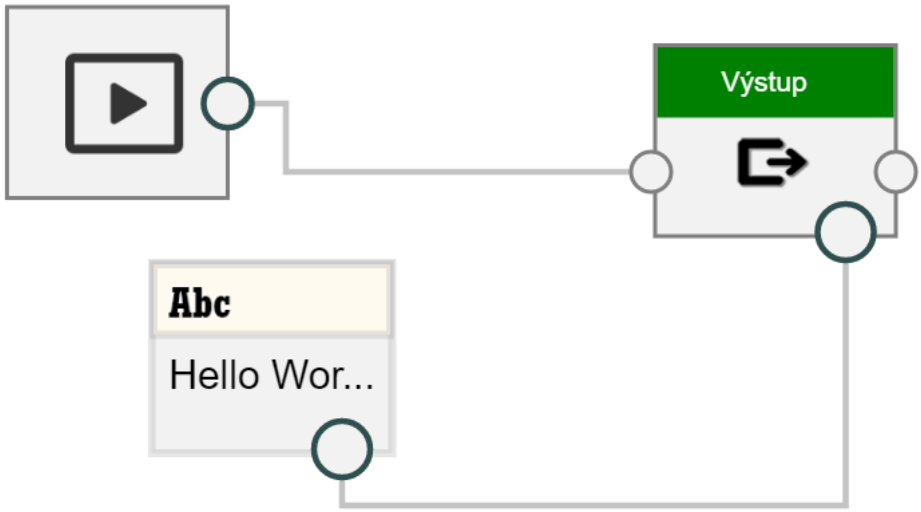
Bloky jsou rozdělené na kategorie podle jejich funkcionality pro lepší orientaci. Kategorie se dají skrývat nebo rozevřít kliknutím na hlavičku dané kategorie.



Obrázek 21: Ukázka dvou skupin programovacích bloků v levém menu vývojového prostředí VizProg

8.2.2 Programová část

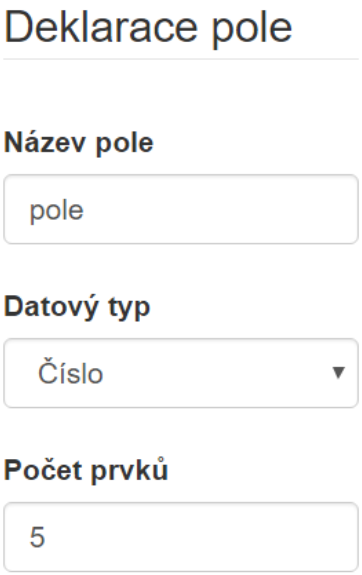
Programová část je hlavní částí vývojového prostředí a právě do této části se z levého menu přidávají bloky, kterým se popřípadě zadají určité parametry a pak se zde spojují do programu. Bloky zde lze, pomocí přidržení levého tlačítka myši a táhnutím, přesouvat, označovat nebo mazat. Smazání bloku se po označení levým tlačítkem myši provede stisknutím klávesy „delete“ nebo „backspace“. Zde se také provádí spojování bloků do logických celků. Celkový průběh spojování je vysvětlen v sekci 8.5 o blocích.



Obrázek 22: Spojení bloků do programu, který na výstupní konzoli vypíše větu „Hello World!“

8.2.3 Vlasnosti bloku

Poslední částí vývojového prostředí je pravé menu, kde se zobrazují vlastnosti označeného bloku. V případě bloků, které mohou nést nějakou hodnotu se zde zobrazí příslušná vstupní pole, kterými můžete vlastnosti bloku upravovat. Například na obrázku 23 je zobrazená vlastnost bloku z kategorie Pole Deklarace pole, kde můžete definovat název pole, datový typ a velikost pole.



Obrázek 23: Vlasnosti bloku „Deklarace pole“, které lze v pravém menu po označení upravit.

8.3 Spouštění programu

Program se dá spustit pomocí tlačítka tlačítka „Spustit“ v horním menu. Spuštěním se bloky odešlou na server, kde se provede sestavení programu do vnitřní formy a následná syntaktická a sémantická analýza. Pokud vše proběhlo správně, zobrazí se případný výstup programu v podobě konzole, jakou můžete znát z prostředí Windows nebo terminálu z distribucí Linux. Okno konzole zavřete pomocí stisknutí libovolné klávesy nebo pomocí křížku v pravé horní části konzole. Při spuštění programu má tlačítko „Spustit program“ roli „Zastavit program“ a změní se i vizuálně na tlačítko stop. V případě chyby se vpravo nahoře u indikátoru stavu programu objeví chybová hláška. Dokud se tato chyba neopraví, program se správně nespustí.

8.4 Ladění programu

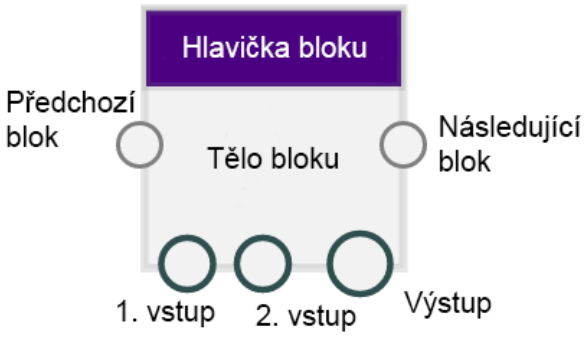
Ladění programu na rozdíl od spuštění programu nespustí program najednou. Místo toho se dá program tzv. krokovat. Po stisknutí tlačítka „Ladit program“ se stejně jako u spuštění programu odešlou bloky na server, kde se provede sestavení programu a analýzy, ale jen k prvnímu bloku. Na tomto bloku se program zastaví a čeká, jestli uživatel posune program o krok dál, resp. zpět nebo celé ladění vypne.

Při spuštěném ladění se vedle tlačítka „Ladit program“ (které má teď funkci zastavit ladění) objeví další 2 tlačítka a to je „Krok vpřed“ a pokud není program úplně na začátku, tak i „Krok vzad“. Při stisknutí na jedno z těchto tlačítek se provede buď další krok programu nebo krok předchozí. Blok, který je zrovna prováděn, je označen červeným čárkovaným ohraničením. Bloky, které již byly zpracovány ukazují svoji aktuální hodnotu v daném kroku. Takto si můžete projít celý program krok po kroku a podívat se, jak se stav programu mění. Pokud dojde během ladění k chybě, znovu se objeví v pravém horním rohu chybová hláška a ladění je ukončeno.

8.5 Bloky

Bloky jsou základním stavebním kamenem jazyka VizProg. Spojováním bloků se vytváří logika programu. Blok se skládá z těla, ve kterém bývá obvykle ikona daného bloku nebo jeho hodnota. Dále jsou zde spojovací články „Předchozí blok“ a „Následující blok“. Každý blok (kromě hodnotových bloků) musí mít svůj předchozí blok a (kromě posledního) následující blok, jinak není do programu přidán. Dále má každý blok vstup (vstupy) nebo výstup.

Na obrázku 24 je ukázána základní podoba bloku, jakou má většina bloků ve VizProg. Tento blok má zleva spojení k „Předchozímu bloku“, zprava zase spojení k „Následujícímu bloku“. Pokud chcete mít daný blok ve svém programu, musí být napojený na nějaký předchozí blok. Tak je jasné, jak jdou bloky za sebou a tvoří tak program. Blok na obrázku 24 má také dva vstupy a jeden výstup. Taková kombinace je typická pro matematickou operaci.

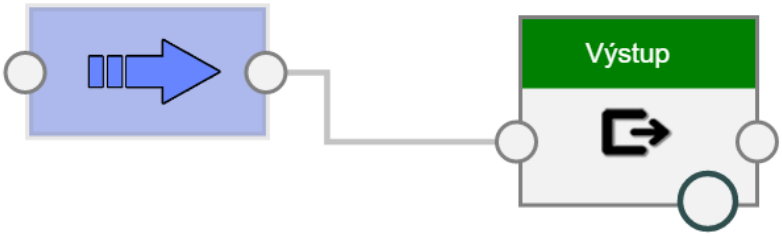


Obrázek 24: Základní rozložení bloku, který je základem pro všechny ostatní bloky kromě bloku podmínky a cyklu

8.5.1 Spojování bloků

Jak bylo řečeno, bloky se dají spolu spojovat. Ne jen spojování následujících a předchozích bloků, ale také spojovat vstupy a výstupy nebo v případě řídících příkazů napojovat bloky do těla těchto bloků. Spojení se vytváří tak, že u bloku, který chcete spojovat podržíte levé tlačítko myši na spojnici, kterou chcete spojit a natáhnete spojovací čáru ke spojnici u dalšího bloku. Pokud jste blok spojili správně, objeví se mezi nimi spojovací čára. Tuto čáru lze pomocí levého tlačítka myši případně označit (svítí červeně) a pomocí klávesy „delete“ vymazat.

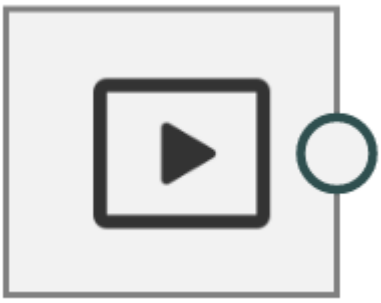
Na obrázku 25 je ukázka napojení z bloku „Pokračovat“ na blok „Výstup“. Blok „Výstup“ je následující blok bloku „Pokračovat“ a blok „Pokračovat“ je předchozí blok bloku „Výstup“. Takto se za sebe bloky napojují.



Obrázek 25: Spojení bloku „Pokračovat“ s blokem „Výstup“

8.5.2 Startovní blok

Nyní se dostáváme ke konkrétnímu popisu jednotlivých bloků. Začneme s tím úplně prvním, který je vždy přítomný v programu právě jednou a nejde smazat. Je to tzv. „Startovací blok“. Tento blok se nachází na úplném začátku programu a nemá žádný předchozí blok. Napojuje se za něj první blok, kterým celý program začíná. Platí tedy, že blok, pro kterého je předchozí „Startovní blok“, je prvním blokem v programu. A od něj se pokračuje dále až do konce programu.



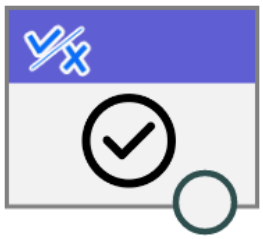
Obrázek 26: Startovní blok, kterým musí začínat každý program ve VizProgu

8.5.3 Hodnoty

Jako první skupinu bloků si popíšeme „Hodnoty“ i když je v levém menu v pořadí až čtvrtá. Obsahuje tři základní datové typy, se kterými VizProg pracuje. Datový typ obecně určuje rozsah hodnot, jakých může proměnná nebo konstanta nabývat. V jiných programovacích jazycích existuje více datových typů pro číslo, mají speciální datový typ pro znak a řetězec, atd. Ve VizProgu hodnotové bloky jako jediné nemají napojení na předchozí a následující blok. Mají jenom výstup, který se dá připojit do vstupů jiných bloků a pořád budou zapojeny do programu. Ovšem bloky, které je berou jako vstup musejí být napojeny v programu.

Boolean

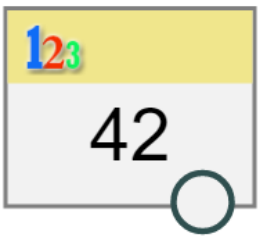
Hodnota typu boolean může nabývat pouze dvou hodnot a to je pravda (TRUE) nebo nepravda (FALSE). Tyto hodnoty se dají libovolně měnit, když je blok označený v pravém menu. Tento blok může být napojen na proměnnou, kdy proměnná nabude jeho hodnotu. Používá se i na vyhodnocení podmínky nebo cyklu.



Obrázek 27: Hodnotový blok datového typu boolean

Číslo

Blok „Číslo“ může nabývat hodnoty od -99999 do 99999 a to včetně desetinných čísel. Hodnotu tohoto bloku lze měnit znovu po označení v pravém menu.



Obrázek 28: Hodnotový blok datového typu číslo

Text

Blok „Text“ může mít text libovolně dlouhý. Znovu jde tuto hodnotu měnit po označení v pravém menu. Hodnota textu na bloku je zkrácená, aby se tam vešla a je doplněna třemi tečkami. Blok jako takový nese plnou hodnotu, která je napsána vpravo, ať je jakkoliv dlouhá.

Obrázek 29: Hodnotový blok datového typu text

8.5.4 Základní komponenty

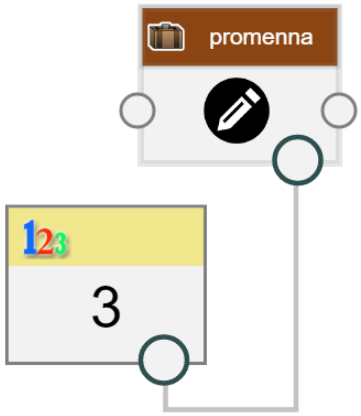
Proměnná

Jeden z nejdůležitějších bloků je „Proměnná“. Bez proměnných by se žádný programovací jazyk neobešel a jsou tak základním kamenem programování. Proměnné bývají určitého datového typu nebo jsou dynamicky typované (což znamená, že v jednu chvíli můžou nabývat hodnoty jednoho datového typu a potom dalšího). Stejně tak jsou udělané proměnné ve VizProgu. Neurčuje se jim datový typ při vytvoření, ale její datový typ se nastaví podle hodnoty, kterou nabývají.

Obrázek 30: Blok proměnné

Proměnná ve VizProgu má dva módy. Mód zápisu, kdy je do proměnné zapisována hodnota a mód čtení, kdy je z proměnné čteno. Do proměnné tedy nejde zapisovat a číst z ní zároveň jako u jiných programovacích jazyků. Pokud byste chtěli do proměnné zapsat hodnotu a pak ji hned přečíst, potřebujete na to dva bloky. Mód proměnné se mění v pravém menu včetně názvu proměnné, který je také důležitý.

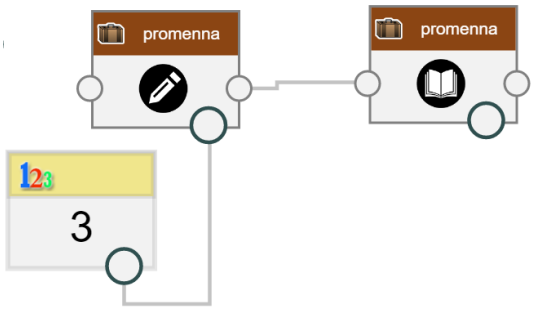
Mód **zápisu** tedy do proměnné zapisuje hodnotu. Takto definujeme novou proměnnou nebo přepisujeme hodnotu již definované proměnné a rozlišují se právě pomocí jmen. Jméno proměnné může být dlouhé až 12 znaků a doporučuji nepoužívat diakritiku a mezery. Jakmile je proměnná v režimu zápisu, stává se z dolního kolečka vstup, kam může být přiveden výstup nějakého jiného bloku, ať už nějaké hodnoty, proměnné v režimu čtení nebo výsledek operace.



Obrázek 31: Kus programu, který reprezentuje zápis hodnoty 3 do proměnné „promenna“

Na obrázku 31 je příklad zápisu hodnoty 3 do proměnné s názvem „promenna“. Pokud se zápis do proměnné „promenna“ použije někde dále v programu novu, hodnota se přepíše a nemusí to být stejný datový typ. Proměnná je dynamicky typovaná.

Druhý mód **čtení** slouží ke čtení hodnoty již vytvořené proměnné. Nelze tedy číst z proměnné, která ještě nebyla vytvořena. V pravém menu můžete vybrat jenom proměnné, které již byly v programu deklarovány režimem zápisu. V módu čtení se dolní kolečko změní na výstup, který lze napojit na vstupy jiných bloků.

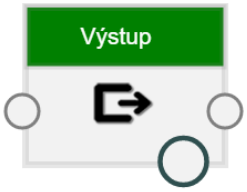


Obrázek 32: Kus programu, který reprezentuje čtení z proměnné s názvem „promenna“, do které byla zapsána hodnota 3

Na obrázku 32 je ukázka zápisu do proměnné „promenna“ čísla 3 a následné čtení. Jakýkoliv blok, který by byl teď napojen na výstup proměnné v režimu čtení by měl jako vstup hodnotu 3 a může to být zase proměnná.

Výstup

Blok „Výstup“ slouží na textový výstup programu. Tento vstup přijímá jakoukoliv hodnotu, kterou poté ve formě textu zobrazí na výstupní konzoli. Pokud je v programu použit vícekrát, vypíšou se všechny výstupy (každý na jeden řádek) na konci do konzole.



Obrázek 33: Výstupní blok

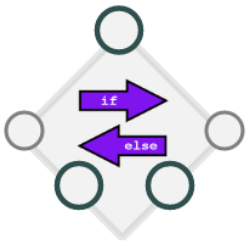
8.5.5 Řídící příkazy

Řídící příkazy jsou to, co dělají programování nejzajímavější a určují nám, jak se program může chovat za určitých podmínek nebo opakovat určitý kus programu dokola za určitých podmínek. Mezi základní řídící příkazy patří podmínka, která provede část programu, pokud je podmínka splněna nebo pokud není splněna. Dalším je cyklus, který na základě podmínky (pokud je splněna) provádí svoje tělo pořád dokola, dokud není podmínka zneplatněna nebo pokud nebyl cyklus přerušen blokem „Přerušit“.

Podmínka

Blok „Podmínka“ obsahuje více vstupů. Kromě předchozího a následujícího bloku obsahuje ještě vstup pro podmínku, která musí být datového typu boolean. Dále

jsou to vstupy s názvy „Podmínka splněna“ (vlevo dole) a „Podmínka nesplněna“ (vpravo dole). Tyto vstupy nepřijímají hodnoty, ale vstupy pro předchozí bloky bloků, které tvoří tělo programu, když je podmínka splněna, resp. podmínka není splněna. Podmínka tedy určuje pomyslný nový startovní blok, ke kterému se připojují další logické celky, které mohou být zase řídící příkazy.

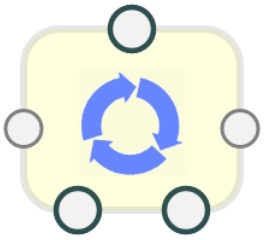


Obrázek 34: Blok podmínky

Cyklus

Blok „Cyklus“ má stejně jako blok „Podmínka“ stejný počet vstupů. Má znovu vstup pro podmínku, která určuje, jestli se tělo cyklu má znovu provádět nebo ne. Vstup vlevo dole znovu slouží pro blok, který bude stát na začátku cyklu. Rozdíl je ale ve vstupu vpravo dole. Tam se napojuje vstup následujícího bloku, který je v těle cyklu poslední. Takto vznikne uzavřené tělo cyklu, které se opakuje. Pokaždé, když proběhne poslední blok, znovu se prověří podmínka a pokud je pravdivá (má hodnotu pravda), znovu se provede tělo cyklu až do doby, než je podmínka zneplatněná nebo pokud nedošlo k přerušení cyklu pomocí bloku „Přerušit“.

VizPro může detekovat i tzv. nekonečný cyklus a dojde k němu v případě, že počet iterací překročil povolený limit, což je hodnota 99999 jako maximální povolená hodnota čísla ve VizProgu. Pokud by totiž byl nekonečný cyklus povolen, prováděl by se program pořád dokola na straně serveru a ke klientovi by se nikdy nedostal výsledek.

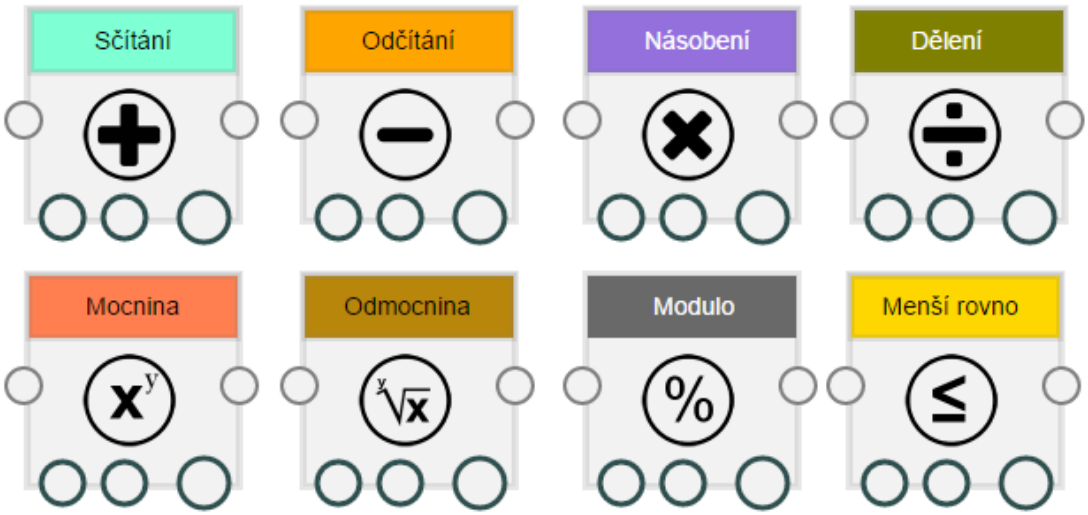


Obrázek 35: Blok cyklu

8.5.6 Matematické operace

VizProg také obsahuje bloky pro matematické operace, které operují s čísly. Jde spíše o základní matematické operace, ze kterých se dají sestavit i složitější kombinace. Jedna operace může naráz pracovat maximálně se dvěma operandy, takže například sčítání třech a více čísel se už musí za sebe skládat do více bloků. Výsledek operace vrátí hodnotu datového typu „Číslo“.

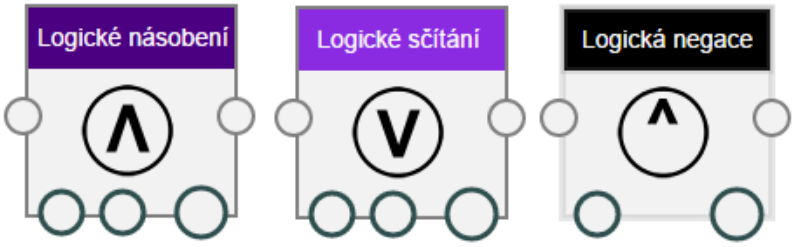
Všechny operace jsou již z názvu zřejmé. Poslední blok „Operace porovnání“ porovná dvě čísla na vstupu (dolní kolečka na levé straně) a výsledek je k dispozici na výstupu (dole vpravo). Jako operandy bere jenom vstup datového typu číslo. Výsledek je ale na rozdíl od ostatních matematických operací datového typu boolean. V pravém menu po označení můžete vybrat druh porovnání. Druhy porovnání jsou: Rovná se, Nerovná se, Větší, Větší rovno, Menší, Menší rovno. Vždy se porovnává první operand s druhým, záleží tedy na pořadí, aby byl výsledek správný.



Obrázek 36: Bloky matematických operací, zleva seshora je to „Sčítání“, „Odčítání“, „Násobení“, „Dělení“, „Mocnina“, „Odmocnina“, „Modulo“ a „Porovnání (Menší rovno)“

8.5.7 Logické operace

Poslední skupinou bloků jsou logické operace. Ty na rozdíl od matematických operací pracují jenom s operandy datového typu boolean a také výsledek je datového typu boolean. Stejně jako u matematických operací přijímají maximálně dva vstupy a mají jeden výstup. V logických operacích se pracuje s tzv. výroky a výsledek takové operace je opět výrok.



Obrázek 37: Bloky logických operací, zleva je to „Logické násobení (konjunkce)“, „Logické sčítání (disjunkce)“, „Logická negace“

9 Testování

Testování aplikace byl věnován velký důraz, hlavně co se týče praktického testování na školách. Praktickému testování předcházelo i programové testování pomocí unit testů jednotlivých funkcí nebo naprogramování vzorových programů, které jsou také k dispozici na přiloženém CD.

9.1 Programové testování

Programové testování se skládá z unit testů a vzorových programů. Unit testy jsou součástí serverové části aplikace a jsou přiloženy na CD společně se serverem. Bylo vytvořeno celkem 41 unit testů pro všechny stěžejní operace. Pokrývá všechny logické i aritmetické operace, všechny operace porovnání a operace s polem, včetně jeho deklarace. Několik testů je vytvořeno přímo na operace s proměnnými, včetně několik aritmetických a logických. Byly vytvořeny i testy na otestování podmínky a cyklu.

Další testování bylo realizováno pomocí vzorových programů, které jsou také přiložené na CD. Jedná se o několik programů ve formátu „.vizprog“, které se dají nahrát do klienta a přímo spustit. Jde o klasické algoritmické úlohy, které se učí na školách a ze kterých se lze naučit algoritmickému myšlení. Na těchto úlohách jsem také vyzkoušel, jestli VizProg pracuje správně. Jde o úlohy typu bubblesort, fibonacciho posloupnost, rozeznávání sudých a lichých čísel, operace s polem, kvadratická rovnice, podmínka, atd.

9.2 Praktické testování

Praktické testování jsem uskutečňoval celkem na dvou školách (GPOA Znojmo a ZŠ nám. Republiky Znojmo) v celkem 4 třídách, každou po dobu jedné vyučovací hodiny (45 minut), kde byla hodina rozdělena na 3 části. První část byla úvodní přednáška, ve které jsem představil tento projekt a vysvětlil základní ovládání. Ve druhé části studenti řešili praktické úlohy (od nejlehčích po nejtěžší) a ve třetí části vyplňovali krátký dotazník viz. příloha A. Zpracované odpovědi z dotazníků na jednotlivé otázky jsou k dispozici v příloze B.

9.2.1 Základní škola

Jako první jsem testoval VizProg na žácích 9. ročníku ZŠ náměstí Republiky ve Znojmě v předmětu zabývajícím se seznámení s hardwarem a softwarem počítače. Žáci neměli žádné předchozí znalosti o programování jako takovém, ale podařilo se jim s mojí pomocí vyřešit některé příklady a zorientovat se v prostředí. Protože nevěděli, co od programování čekat, byly i jejich odpovědi v dotazníku někde spíše nejisté. Většina jich ale byla toho názoru, že je využití vizuálních programovacích jazyků vhodné pro výuku základů programování a algoritmizace. Co by ale zlepšili je přehlednost prostředí a také trochu upravený grafický design, jinak by neměnili vůbec nic. Většinu žáků také zaujalo programování díky této

přednášce, ale nikdo z nich se neplánuje dále věnovat programování nebo vůbec informatice.

9.2.2 Obor informatika

V oboru informatika na GPOA Znojmo byly testovány celkem 2 třídy. Jako první jsem projekt testoval na 3. ročníku oboru Informační technologie. Studenti mají již zkušenosti jak s vizuálním programovacím jazykem, tak i s některým strukturovaným programovacím jazykem. Těmto studentům nebylo potřeba dopodrobna vysvětlovat základní programovací konstrukce a ve VizProgu se rychle zorientovali. Zároveň i většinu úkolů, které jsem pro ně připravil zvládli vyřešit ve velice krátkém čase. Co se týče připomínek nebo nedostatků k programu, měli připomínky spíše ke grafickému rozlišení vstupů a výstupů nebo k nemožnosti správného spuštění v jiných prohlížečích, než je Google Chrome nebo Opera.

Ve druhé třídě byli studenti 2. ročníku oboru Informační technologie. Šlo u nich znát, že se s programováním a algoritmizací teprve seznamují, ale některé základní znalosti již mají. Bohužel kvůli časovému limitu nebyl dostatek času pro podrobnější vysvětlení všech bloků a tvoření rozsáhlejšího programu, takže někteří nebyli vůbec schopni samostatné práce i když měli k dispozici podrobnou nápovědu. Do testování se ale zapojili všichni a snažili se vyřešit co nejvíce úloh. Co se týče připomínek a nedostatků z dotazníku, byly téměř totožné s předchozí třídou, nejvíce kritiky sklidil právě grafický design. Ovšem nejčastěji zazněla odpověď, že na programu by neměnili vůbec nic. Další návrhy byly spíše na vylepšení, jako možnost označování více bloků najednou, atd.

9.2.3 Obor Předškolní a mimoškolní pedagogika

Můj poslední test aplikace byl ve třídě se studentkami oboru Předškolní a mimoškolní pedagogika na GPOA Znojmo. Žádná z těchto studentek nikdy nic neprogramovala a pojem programovací jazyk byl pro ně zcela cizí. Všechny studentky ovšem spolupracovaly a snažily se naučit základy programování pomocí jazyka VizProg. Dívaly se spíše na vizuální stránku programu, který se jim až na některé drobnosti líbil a podařilo se všem naprogramovat připravené úlohy. Z dotazníku, který vyplňovaly na konci hodiny je zřejmé, že studentky programování díky této ukázce zaujalo, ale žádná z nich neplánuje se dále programování věnovat. Většina si také myslí, že je využití vizuálního programovacího jazyka vhodné pro výuku programování. Co se týče názoru na technické vylepšení programu, nebyly zde žádné připomínky.

9.3 Výsledky praktického testování

Celkem byl program VizProg otestován ve 4 třídách, z nichž jedna byla na základní škole a tři na střední škole. Výsledky jsou ze všech tříd skoro totožné, i když dvě třídy byly z oboru Informační technologie. Otázky z dotazníku byly zaměřeny hlavně na myšlenku výuky pomocí vizuálního programovacího jazyka a na samotný VizProg. Většině dotázaným se VizProg jako takový líbil i když neměli s vizuálními programovacími jazyky zkušenosti. Bylo zde pár připomínek k designu rozhraní a někteří studenti přišli i se zajímavými nápady, jak případně design vylepšit (například graficky odlišit vstup a výstup). Z mého pohledu jako lektora hodnotím práci studentů a dosažené výsledky více než kladně. Studenti se ochotně zapojovali do práce a někteří se snažili i s programem experimentovat, díky čemuž jsem mohl opravit dva drobné bugy v klientské části aplikace. Moje přednášky se lišily i podle tříd. Ve třídách z oboru Informační technologie jsem mohl zajít více do detailů, popsat více programovacích bloků a porovnat programovací bloky s konstrukcemi jazyka C#, který se na této škole vyučuje. U žáků ze základní školy a z pedagogického oboru jsem tyto konstrukce vysvětloval více povrchově a zaměřil se spíše na praktické ukázky, jak jednotlivé bloky fungují a co se při jejich vykonávání děje. Bohužel 45 minut je velice krátká doba na naučení základů algoritmizace a programování. Největší problém ve vyučování algoritmizace a programování vidím spíše v zaujetí studentů pro tuto problematiku a aby je daná práce bavila. Myslím si, že vizuální jazyk VizProg by mohl některým dětem nebo studentům otevřít dveře do světa programování a motivovat je k dalšímu učení s jiným programovacím jazykem. Hlavní ale je, aby měli dobrého učitele, který problematice rozumí a umí to také dobře vysvětlit a naučit. Všechno ostatní je už na studentech.

Závěr

Cílem této práce bylo navrhnout nový vizuální programovací jazyk pro výuku algoritmizace a programování u mladších studentů. Zároveň bylo potřeba navrhnout a vytvořit přívětivé vývojové prostředí s možností základního „ladění“ programů. Prostudoval jsem si materiály o tvorbě výukových programů a také již existující řešení, které se na některých školách používají. Nakonec se mi podařilo vytvořit vizuální programovací jazyk nazvaný VizProg, který je jednoduchý na pochopení a obsahuje všechny potřebné konstrukce, jaké má mít plnohodnotný programovací jazyk (konstanty, proměnné, řídící příkazy, matematické operace, logické operace a textový výstup).

Spolu s jazykem jsem vytvořil i přívětivé vývojové prostřední dostupné na webu pomocí prohlížeče Google Chrome nebo Opera na adrese www.vizprog.com. Toto prostředí bylo napsáno za pomocí jazyka JavaScript s využitím knihoven React a Redux. Pomocí něj se dají vytvářet libovolně velké programy v jazyce VizProg za pomocí spojování programovacích bloků a ty pak následně spouštět nebo ladit pomocí krokování. Spuštění a ladění programu je realizováno pomocí serveru, který interpretuje program vytvořený ve vývojovém prostředí. Tento program je serveru z klienta zaslán pomocí Web API a stejným způsobem je vrácena odpověď. Server je napsán v jazyce C# s využitím ASP.NET Core Frameworku.

Výsledný program byl nakonec otestován v celkem 4 třídách, z nichž tři byly na střední škole a jedna na základní škole. Studenti byli seznámeni s problematikou programovacích jazyků a algoritmizace. Následně jim byly předvedeny vlastnosti jazyka VizProg a vývojového prostřední. Nakonec si zkusili sami naprogramovat několik jednoduchých programů a vyplnili dotazník týkající se jazyka VizProg.

Co se týče možných vylepšení, je jich hned několik. Hlavně se týkají designu vývojového prostředí, na který poukazovalo i několik studentů při testování. Bylo by dobré graficky oddělit vstupy a výstupy nebo přidat možnost označovat a mazat více bloků najednou. Další vylepšení by se mohly týkat lepší administrace a více možností učitele spravovat jednotlivé programy. Například by mohla existovat autentizace a autorizace na školním serveru, kde by byli žáci přihlášeni a mohli by tyto programy přímo odevzdávat do školního systému. Učitel by mohl mít přes tento systém možnost přímo kontrolovat a ladit programy konkrétních žáků.

Conclusions

Goal of this thesis was to design new visual programming language for teaching algorithms and programming for younger students. Simultaneously it was necessary to design and create user friendy development environment with basic "debugging" functions. I studied materials about creating teaching programs and also some solutions which has been made and which are used in schools. Finally I developed visual programming language called VizProg, which is simple to understand and contains all required constructions which programming language has to have like constants, variables, cycles, conditions, mathematic operations and text output.

I also created user friendly development environment accessible in Google Chrome or Opera internet browsers available on www.vizprog.com. This environment has been written in JavaScript language using React a Redux libraries. In this environment you can create programs in VizProg language with connecting programming blocks and then you can run or debug this program with steps. Running and debuging program is implemented in server which interpreting this program created in the environment. This program is received by server from client by Web API and send answer also with this method. Server is written in C# language using ASP.NET Core Framework.

My final program was tested in 4 classes where 3 classes were on high school and one on elementary school. I explained problematics about programming languages and algorithmization. Then I showed them abilities of VizProg language and development environment. They tried program some simple programs on their own and then filled questionnaire about this language.

There are many possible improvements for this system. Mainly its about design of the development environment, which said some students during the testing. It would be good to separate inputs and outputs with graphic or tag multiple blocks at the same time. Next improvement should be better administration for lector who should have more posibilities to administrate programs in VizProg. For example there should be exist some authentification and authorization on school server, where students and teachers should login submit their programs into this system. Lector then should haeve possibility to check and debug programs of his students.

A Dotazník

1. **Už jste někdy programoval(a) v jakémkoliv programovacím jazyce?**

   - ◯ Ano
   - ◯ Ne

2. **Zaujalo Vás programování díky této ukázce?**

   - ◯ Ano, určitě si někdy něco zkusím naprogramovat
   - ◯ Ano, ale programovat nebudu
   - ◯ Ne

3. **Jste pro využití vizuálních programovacích jazyků na výuku algoritmizace a programování?**

   - ◯ Ano
   - ◯ Ne

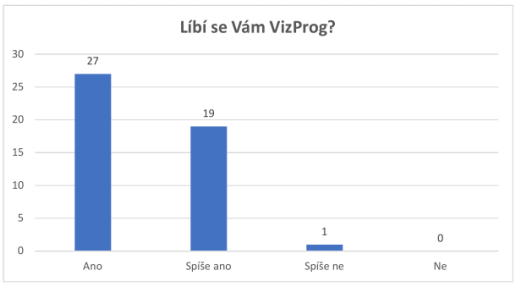
4. **Líbí se Vám VizProg?**

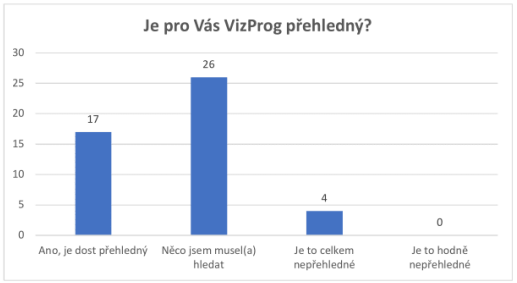
   - ◯ Ano
   - ◯ Spíše ano
   - ◯ Spíše ne
   - ◯ Ne

5. **Je pro Vás VizProg přehledný?**

   - ◯ Ano, je dost přehledný
   - ◯ Něco jsem musel(a) chvíli hledat
   - ◯ Je to celkem nepřehledné.
   - ◯ Je to hodně nepřehledné

B Výsledky z dotazníků





C Obsah přiloženého CD/DVD

`client/`
: Zdrojové soubory klienta VizProg v Reactu včetně nápovědy. K jeho spuštění a nainstalování všech potřebných balíčků je potřeba balíkový systém *npm*, který je volně ke stažení na https://www.npmjs.com/. Funkční klient s připojením na server je také dostupný online na adrese www.vizprog.com.

`server/`
: Zdrojové soubory serveru VizProg v ASP.NET Core frameworku. K jeho spuštění a nainstalování všech potřebných balíčků je potřeba mít nainstalován .NET Core, který je volně ke stažení na https://www.microsoft.com/net/download/core.

`doc/`
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce včetně zdrojových souborů ve formátu *.ZIP*.

`examples/`
: Všechny přiložené vzorové programy pro klienta VizProg. K jejich spuštění je potřeba mít spuštěný server i klienta VizProg na webovém serveru. Poté už stačí jen v klientovi některý z těchto programů otevřít a spustit.

`readme.txt`
: Instrukce pro nasazení a spuštění serverové a klientské části programu VizProg a všech požadavků na jejich bezproblémový provoz.

D Použitá literatura

Literatura

[1] *Announcing .NET Core 1.0*. URL: https://blogs.msdn.microsoft.com/dotnet/2016/06/27/announcing-net-core-1-0/.

[2] *Aplikace EV3 Programmer - Lego Mindstorms*. URL: https://www.lego.com/cs-cz/mindstorms/apps/ev3-programmer-app.

[3] *Baltík*. 2015. URL: https://cs.wikipedia.org/wiki/Balt%C3%ADk.

[4] Milan Češka aj. *Konstrukce překladačů*. Czech. Praha, CZ: Czech Technical University, 1999, s. 636. ISBN: 80-01-02028-2. URL: http://www.fit.vutbr.cz/research/view_pub.php?id=5594.

[5] Eddy Chang. *React Lifecycle methods diagram – Eddy Chang – Medium*. 2016. URL: https://medium.com/@eddychang_86557/react-lifecycle-methods-diagram-38ac92bb6ff1.

[6] *Components and Props - React*. 2016. URL: https://facebook.github.io/react/docs/components-and-props.html.

[7] *Core Concepts of Redux*. URL: http://redux.js.org/docs/introduction/CoreConcepts.html.

[8] *ECMAScript*. URL: https://en.wikipedia.org/wiki/ECMAScript.

[9] *Home repository for .NET Core*. 2016. URL: https://github.com/dotnet/core.

[10] Cody Jackson. *Learning to Progam Usin Python*. 2013. URL: https://docs.google.com/file/d/0B8IUCMSuNpl7MnpaQ3hhN2R0Z1k/edit.

[11] Lada Jandová. *Počítačová výuka : zásady tvorby výukových programů*. Plzeň : Západočeská univerzita, 1995, s. 62. ISBN: 80-7043-147-4.

[12] *JSON*. URL: http://www.json.org/json-cz.html.

[13] *Klient-server*. 2015. URL: https://cs.wikipedia.org/wiki/Klient-server.

[14] *Kodu*. 2012. URL: https://www.kodugamelab.com/.

[15] Dalibor Krčmář. *Programujeme .NET aplikace*. COMPUTER PRESS, 2001, s. 336. ISBN: 80-7226-569-5.

[16] Martin Malý. *REST: architektura pro webové API*. 2009. URL: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/.

[17] E.F. Meyer aj. *Guide to Teaching Puzzle-based Learning*. Springer-Verlag Londo, 2014, s. 345. ISBN: 978-1-4471-6475-3.

[18] *Mono*. URL: http://www.mono-project.com/.

[19] Jan Neckář. *Algoritmus*. Ed. by Jan Neckář. 2015. URL: https://www.algoritmy.net/article/1240/Algoritmus.

[20] Hadi Partovi. *Hour of code*. 2013. URL: https://code.org/.

[21] David Čápka. *Úvod do ASP.NET*. 2012. URL: http://www.itnetwork.cz/csharp/asp-net/tutorial-uvod-do-asp-dot-net.

[22] David Čápka. *Úvod do C# a .NET frameworku*. 2012. URL: http://www.itnetwork.cz/csharp/zaklady/c-sharp-tutorial-uvod-do-jazyka-a-dot-net-framework.

[23] David Čápka. *Úvod do JavaScriptu*. 2013. URL: https://www.itnetwork.cz/javascript/zaklady/javascript-tutorial-uvod-do-javascriptu-nepochopeny-jazyk.

[24] Tomáš Randus. *Redux + React – Prostředí*. 2016. URL: https://www.zdrojak.cz/clanky/redux-react-13-prostredi/.

[25] Tomáš Randus. *Redux + React – Prostředí*. 2016. URL: https://www.zdrojak.cz/clanky/redux-react-13-prostredi/.

[26] *React (JavaScript library)*. 2017. URL: https://en.wikipedia.org/wiki/React_(JavaScript_library).

[27] *React.JS and Virtual DOM*. 2015. URL: http://madhukaudantha.blogspot.cz/2015/04/reactjs-and-virtual-dom.html.

[28] Mitchel Resnick. *Scratch*. 2005. URL: https://scratch.mit.edu/.

[29] Margaret Rouse. *What is garbage collection?* URL: http://searchstorage.techtarget.com/definition/garbage-collection.

[30] *State and Lifecycle - React*. 2016. URL: https://facebook.github.io/react/docs/react-component.html.

[31] *Store · Redux*. URL: http://redux.js.org/docs/basics/Store.html.

[32] *SVG Tutorial*. URL: https://www.w3schools.com/graphics/svg_intro.asp.

[33] *Three Principles · Redux*. URL: http://redux.js.org/docs/introduction/ThreePrinciples.html.

[34] *Using Redux in React*. URL: https://ghost.mybluemix.net/using-redux-in-react/.

[35] Zdeněk Vídeňský. *VizProg - Nápověda*. 2017. URL: http://www.vizprog.com/help/.