Středoškolská technika 2018

Setkání a prezentace prací středoškolských studentů na ČVUT

INTERAKTIVNÍ APLIKACE PRO VÝUKU ALGORITMŮ

David Martínek

Vyšší odborná škola, Střední škola, Centrum odborné přípravy
Budějovická 421, Sezimovo Ústí

Prohlášení

Prohlašuji tímto, že jsem maturitní projekt vypracoval samostatně pod vedením pana učitele Jiřího Roubala a uvedl jsem veškerou použitou literaturu a další informační zdroje včetně internetu.

V Sezimově Ústí dne 10. dubna 2018

podpis autora

Poděkování

Rád bych poděkoval vedoucímu této práce Ing. Jiřímu Roubalovi, Ph.D. za nápad využití vývojových diagramů v aplikaci Robot Karel, za pomoc s tvorbou tohoto textu a za testování aplikace. Dále bych rád poděkoval Mgr. Taťáně Horehleďové za jazykovou korekturu této práce a Mgr. Haně Filipové za korekturu anglické anotace. Speciální poděkování patří Mgr. Jaroslavu Pejšovi, dnes profesionálnímu programátorovi, za zpětnou vazbu ke kapitole 4. Také děkuji Jakubovi Davidovi za vytvoření obrázků robota, zdi a diamantu. Poděkování též patří škole za podporu a možnost vytvoření tohoto projektu. V neposlední řadě bych rád poděkoval svým rodičům a blízkým za podporu při studiu.

Anotace

V této práci byla navržena a naprogramována nová interaktivní aplikace pro výuku algoritmického myšlení. Tato aplikace vychází z vývojového prostředí Robot Karel, ale algoritmus se zde nezapisuje pomocí programového kódu nýbrž pomocí vývojového diagramu, který se následně vykoná v její grafické části. Aplikace byla naprogramována v jazyce C# pomocí frameworku WPF a splňuje požadavky kladené na dnešní moderní výuku. V současné době umožňuje vytvářet základní algoritmické struktury (posloupnosti, větvení, cykly). Dále lze tyto algoritmy ukládat a testovat je v grafické části aplikace.

Annotation

A new interactive application for teaching algorithm thinking was designed in this project. The application is based on development environment Robot Karel but the algorithm is not written using a program code but using flow chart which is then executed in the graphic part of the application. It was programmed with C# using WPF framework and it meets today's educational requirements. Right now it allows to create basic algorithmic structures (sequences, branching, cycles) to save these algorithms and test them in the graphic part.

Klíčová slova: Algoritmus; výuková aplikace; Robot Karel; C#; framework WPF.

Obsah

Kapitola 1

Úvod

Kdo by se dnes nesetkal s počítačem? Prakticky neexistuje odvětví, ve kterém by se dnes počítače nepoužívaly. Řídí veřejnou dopravu, dodávky energie a vody do domácností, lze je nalézt v osobních automobilech, v průmyslové výrobě, bankovnictví, zdravotnictví, ve sportu. Mnoho lidí využívá počítače také k zábavě, ke sledování filmů, hraní her, komunikaci s okolím a tak dále. Tyto možnosti ale nevzešly z přírody. Vytvořil je člověk, který musel počítač nejprve postavit a poté naprogramovat. K tomu musel bezpodmínečně zvládnout algoritmické myšlení.

Protože se počítače staly nedílnou součástí lidského života, měli by se lidé alespoň částečně naučit algoritmicky myslet, aby jim mohly počítače sloužit a ne naopak, aby se lidé stali jejich otroky. Součástí informatických oborů a kurzů je výuka tvorby algoritmů, to je přesných postupů, které vedou k danému cíli.

Jednou z možností, jak názorně zapsat nějaký algoritmus, je použití vývojového diagramu. Vývojové diagramy používají různé geometrické tvary propojené pomocí orientovaných čar. Tyto tvary reprezentují dílčí části algoritmů jako jsou posloupnosti, větvení a cykly (Pečínková, J., 2009). Dříve probíhala výuka algoritmického myšlení právě pomocí vývojových diagramů, které se ručně kreslily na tabuli. Žáci je následně přepisovali do kódu v daném programovacím jazyce. Souvislost mezi vývojovým diagramem a následným programem byla ale pro některé žáky obtížně viditelná.

Proto v minulosti vznikly aplikace pro výuku algoritmického myšlení, které spojily tvorbu algoritmu s jeho následným provedení. Dnes existuje takovýchto aplikací, které lze nalézt na internetu, celá řada, například Robot Karel (Pattis, R. E., 1981; Lecián, T., 2017), Baltík, Scratch (Krejsa, J., 2014). Některé tyto aplikace jsou ovšem poněkud zastaralé, některé působí spíše dojmem hry určené pro mladší věkové skupiny, a nehodí se tak pro výuku na středních nebo vyšších odborných školách. Srovnáním existujících apli-

kací se ukazuje, že chybí aplikace, která by měla moderní (grafický) vzhled a jednoduché interaktivní ovládání a zároveň spojovala „teorii s praxí“ tak, jak to ve své době dělala aplikace Robot Karel (Pattis, R. E., 1981).

Cílem této práce tedy je porovnat výhody a nevýhody existujících aplikací pro výuku algoritmického myšlení. Dalším, a to hlavním cílem, je vytvořit aplikaci novou, která bude splňovat požadavky kladené na dnešní moderní školu. Tato aplikace spojí vývojové diagramy, kterými se obvykle algoritmy znázorňují, s grafickým rozhraním, ve kterém se názorně provede algoritmus, který uživatel vytvořil.

Struktura této práce, která je napsána v LaTeX 2_ε[1] (Schenk, C., 2009), je následující. Kapitola 2 porovnává některé existující aplikace pro výuku algoritmizace. Kapitola 3 ukazuje, jak se v nově navržené aplikaci vytvářejí základní algoritmické struktury a vysvětluje odlišnosti zápisu od klasických vývojových diagramů. V kapitole 4 je popsán vývoj nově navržené aplikace. V příloze této práce čtenář nalezne samotnou aplikaci Karel VD včetně zdrojových kódů, tento text a některé zdroje, na které se tato práce odkazuje.

Autor této práce se o programování zajímá již několik let. Přibližně v patnácti letech objevil jazyk C# a začal se v něm učit a zlepšovat. Od té doby již vytvořil několik menších vlastních projektů, například aplikaci na správu TeamSpeak 3 serveru (TeamSpeak je komunikační aplikace, ze které vychází dnes velice populární aplikace Discord), nebo aplikace na úpravu binárních souborů s nastavením. V poslední době autor sbírá zkušenosti v ASP.NET, kde programuje dynamické webové stránky pomocí jazyka C# pro firmu SpravaIT.

[1] LaTeX 2_ε je rozšíření systému LaTeX, což je kolekce maker pro TeX. TeX je ochranná známka American Mathematical Society.

Kapitola 2

Výuka algoritmizace pomocí softwarových aplikací

Tato kapitola popisuje několik softwarových aplikací, které byly v minulosti vyvinuty pro výuku *algoritmického myšlení*, které je nezbytné pro *programování*, to je k vytváření počítačových aplikací, bez kterých si dnes asi běžný život nelze představit. Ať už se jedná o aplikace typu počítačových her, kancelářských prostředí (MS Word, Excel, PowerPoint), aplikace pro vytváření elektronických schémat, plošných spojů či simulace elektronických obvodů (Eagle, MultiSim), technických výkresů (AutoCAD, Solid Edge), aplikace pro řízení a vizualizaci technologických procesů, pro analýzu či syntézu regulačních dějů (Matlab/Simulink), aplikace pro internetové bankovnictví, e-mail, mobilní aplikace, aplikace pro práci s digitální fotografií atd., všechny tyto aplikace by nemohly být vytvořeny, kdyby jejich tvůrci dokonale nezvládli algoritmické myšlení.

V této kapitole je tedy nejprve shrnut pojem algoritmus jako stěžejní pojem celé této maturitní práce. Dále jsou zde stručně popsány některé aplikace používané při výuce algoritmického myšlení. Na závěr této kapitoly jsou shrnuty jejich výhody a nevýhody, na jejichž základě bude vytvořena aplikace zcela nová, která bude splňovat požadavky na moderní výuku.

2.1 Algoritmus a jeho vlastnosti

Většina lidí ráno vstane a provádí určitou rutinu (toaleta, umytí obličeje, snídaní, oblékání, cesta do práce...). Tuto mají tak zautomatizovanou, že už nad ní nepřemý-

šlejí. Podobné je to například při řešení nějaké matematické úlohy. Ta může být pro někoho složitější, ale někdo ji má už tak zautomatizovanou, že ji vyřeší velmi rychle. Oba tyto příklady mají jedno společné a to, že se skládají z určitých jednoduchých na sebe navazujících kroků, které dohromady dávají nějaký postup. Přesný postup, který řeší určitou úlohu pomocí konečného počtu jednoznačně definovaných kroků, se nazývá **algoritmus** (Pečínková, J., 2009). Podle odborné literatury musí každý algoritmus splňovat následující podmínky.

- *Determinovanost* (jednoznačnost) – v každém kroku musí být naprosto zřejmé, co a jak se má provést.
- *Elementárnost* – algoritmus se skládá z konečného počtu jednoduchých (elementárních) kroků.
- *Finitnost* (konečnost) – algoritmus skončí v konečném počtu kroků (v konečném čase).
- *Opakovatelnost* – algoritmus musí pro stejné vstupy dospět ke stejnému výsledku.
- *Rezultativnost* – algoritmus musí vydat aspoň jeden výstup, který je správný.
- *Univerzálnost* (obecnost) – algoritmus musí být použitelný pro všechny úlohy stejného typu (neřeší pouze jeden konkrétní problém).

Stojí za zmínku, že slovo algoritmus se používá na počest perského matematika z první poloviny 9. století, kterým byl Abu Ja'far Mohammed ibn Mûsa al-Khowârizm, a který se zabýval algebrou (Velebil, J., 2017). Jméno bylo do latiny převedeno jako „algoritmus“ a původně znamenalo „provádění aritmetiky pomocí arabských číslic“.

2.2 Aplikace používané při výuce algoritmizace

V této podkapitole budou velice stručně představeny dvě aplikace, konkrétně „Robot Karel“ a „Scratch“, které jsou při výuce algoritmizace patrně nejrozšířenější. Zároveň zde budou uvedeny důvody pro vytvoření aplikace nové, která bude slučovat výhody popsaných aplikací.

2.2.1 Robot Karel

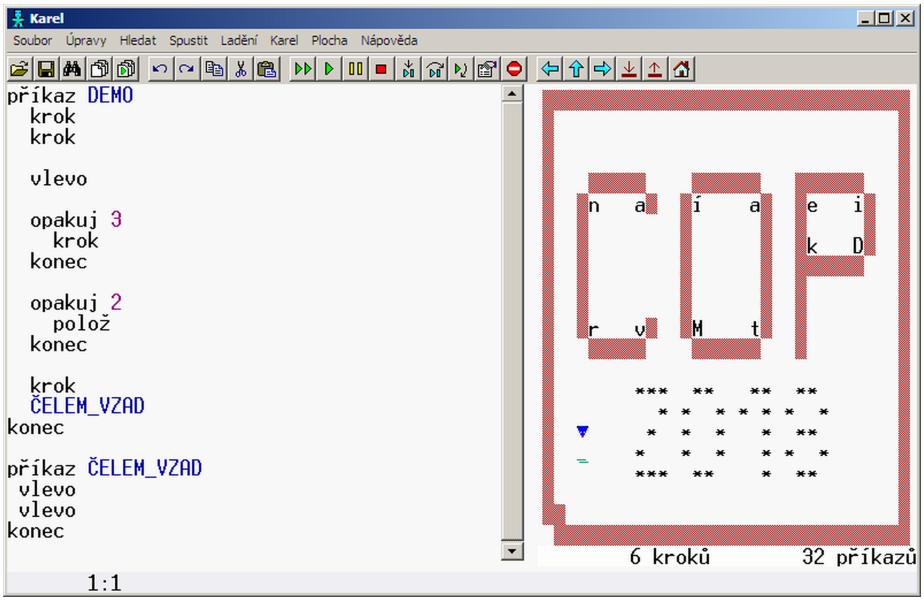
Jedním z nejznámějších prostředí pro výuku algoritmického myšlení je programovací jazyk **Karel**. Vymyslel ho profesor Richard E. Pattis, když učil své studenty programátorskému umění na Stanfordově univerzitě ve Spojených státech amerických. Původní koncepci tohoto jazyka popsal v knize (Pattis, R. E., 1981). Za zmínku stojí také to, že Pattis zvolil jméno Karel (ne Charles) na počest českému spisovateli Karlu Čapkovi, autoru divadelní hry R. U. R, při jejíž premiéře se lidé v roce 1921 v Hradci Králové se slovem „robot“ prvně setkali.

V Pattisově verzi se robot mohl pohybovat ve světě (městě), které bylo tvořeno vodorovnými ulicemi (v originále streets) a svislými bulváry (avenues). V tomto světě se mohly nacházet dva objekty. Jedním objektem byla zeď (wall), přes kterou nemohl robot projít a druhým objektem byl bzučák (beeper), který mohl robot na daném místě položit nebo zvednout. Robot se mohl nacházet pouze na průsečících ulic a bulvárů a rozuměl pěti základním příkazům

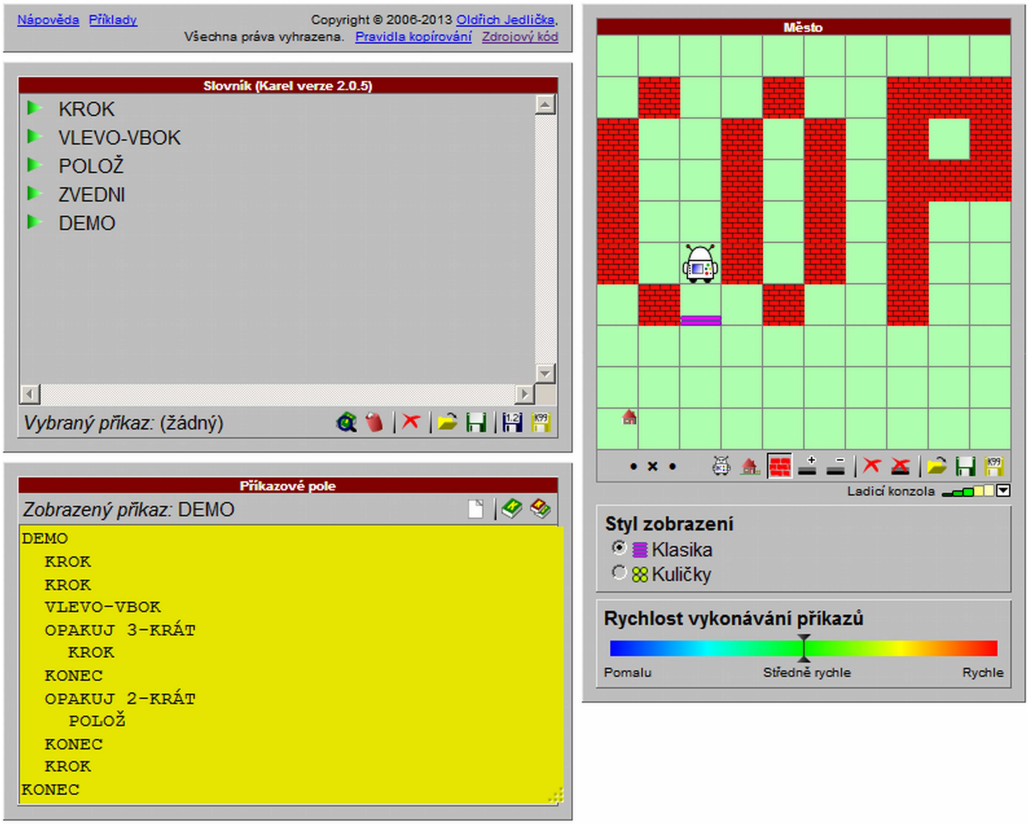
- `move` – přesuň se na další křižovatku,
- `turnleft` – otoč se vlevo,
- `putbeeper` – polož bzučák,
- `pickbeeper` – zvedni bzučák,
- `turnoff` – vypni se.

Robot mohl dále vyhodnocovat, co se kolem něho nachází za objekty. To umožňovalo studentům vytvořit téměř libovolné vlastní funkce, díky nimž se mohl robot bezpečně pohybovat po celém městě. Tvorba těchto vlastních funkcí probíhala v textové podobě. Jak tato „aplikace“ původně vypadala lze nalézt například v (Lecián, T., 2017, obr. 1), kde je také popsán další vývoj robota Karla.

Co se týče českého prostředí, určitě stojí za zmínku velice pěkná knížka (Synovcová, M., 1989), která má přiblížit algoritmické myšlení (programování počítačů) malým dětem právě na bázi robota Karla prostřednictvím stolní hry (bez používání počítače). Dále lze zmínit prostředí, které vytvořil Petr Laštovička v jazyce `C++` pro platformy MS DOS i Windows, jehož náhled je na obr. 2.1. Další pěknou obdobou robota Karla je webová aplikace vytvořená v jazyce JavaScript Oldřichem Jedličkou (Jedlička, O., 2006), jejíž náhled je na obr. 2.2.



Obrázek 2.1: Robot Karel od Petra Laštovičky



Obrázek 2.2: Robot Karel od Oldřicha Jedličky

2.2.2 Aplikace Scratch

Scratch je vizuálním programovacím jazykem, kde se algoritmy tvoří posouváním a skládáním grafických elementů. Začal vznikat v roce 2003, je zdarma a nyní lze využívat jeho webovou nebo lokální verzi. Tato aplikace je velmi intuitivní a lze v ní snadno vyvíjet různé hry a učit se tak algoritmickému myšlení. Může se též využít na přípravu animovaných prezentací, tvorbě interaktivního umění, hudby a animovaných příběhů (Scratch, 2003). Náhled tohoto prostředí je vidět na následujícím obrázku.

Obrázek 2.3: Náhled aplikace Scratch

2.2.3 Výhody a nevýhody stávajících aplikací

Jedním z cílů této práce je porovnat výhody a nevýhody aplikací, které se používají pro výuku algoritmického myšlení. Druhým a zároveň hlavním cílem této práce je, na základě tohoto porovnání, vytvořit vlastní aplikaci pro výuku algoritmizace, která bude vhodná pro střední či vyšší odborné vzdělávání a zároveň bude splňovat požadavky na moderní výuku.

V aplikaci Robot Karel je možné vidět již poněkud zastaralý vzhled. Také psaní programu pomocí příkazů může v dnešním světě interaktivních aplikací žáky a studenty spíše odradit. V aplikaci Scratch je naopak použit moderní vzhled. Programy se tvoří intuitivně pomocí skládání obrázkových příkazů, které se přesouvají po interaktivní části aplikace. Aplikace ale působí spíše dojmem, že je určena pro mladší věkové skupiny, než jsou právě středoškolští žáci nebo studenti vyšší odborné školy.

Nově tvořená aplikace, která je vyvíjená v rámci této práce, bude podobná robotovi Karlovi v tom, že se robot bude také pohybovat po šachovnici. Program se ovšem nebude psát pomocí textových příkazů, ale bude se tvořit sestavováním vývojových diagramů, které jsou grafickým a zároveň velmi přehledným způsobem zápisu nějakého algoritmu (Pečínková, J., 2009). Po sestavení daného algoritmu se bude robot pohybovat po šachovnici právě podle vytvořeného vývojového diagramu.

Kapitola 3

Značky vývojového diagramu

V této kapitole budou zopakovány základní značky vývojových diagramů pro zápis posloupností, větvení a cyklů. Nebude zde ale opisován výklad, který lze najít v mnoha publikacích, například (Pečínková, J., 2009), a samozřejmě i na mnoha internetových stránkách. Kapitola pouze ukáže jejich použití na několika konkrétních příkladech a zaměří se na rozdílnost zápisu pomocí klasického vývojového diagramu a vývojového diagramu, který je použit v navržené aplikaci Karel VD, kde bylo potřeba tyto změny udělat kvůli lepší manipulaci uvnitř aplikace.

3.1 Rozdíly ve značkách vývojových diagramů

V navržené aplikaci představuje nakreslený vývojový diagram v podstatě poskládané jednotlivé metody aplikace v určeném pořadí. Proto bylo potřeba některé značky vývojových diagramů přizpůsobit tak, aby mohl být základ aplikace co nejvíce univerzální. V aplikaci je značka symbolizována klasicky určitým geometrickým obrazcem, u kterého lze z určitého bodu táhnout spojnici k další značce. Podle těchto bodů se právě řídí chod navržené aplikace.

Například u standardně kreslených vývojových diagramů má cyklus s podmínkou na začátku po provedení tak zvaného těla cyklu, tedy toho, co by se mělo opakovat, dokud počáteční podmínka platí, vedenou cestu znovu před počáteční značku cyklu. To je do stejného místa, kam ale také vedla spojnice z předchozí části algoritmu. To znamená, že konec těla cyklu je pro aplikaci vlastně stejný jako začátek cyklu. Tato situace by tedy znamenala, že by aplikace při každém průchodu cyklem hledala znovu podmínky a začátek

cyklu místo toho, aby ihned vyhodnocovala, jestli podmínka stále platí, a pokud ano, tak pokračovala do těla cyklu.

Jak již bylo zmíněno výše, v navržené aplikaci byla snaha mít základ, kam patří posloupnost, větvení a základní cykly, co nejvíce univerzální. Proto byla některá značení ve vývojových diagramech pro potřeby této aplikace upravena. Jednotlivé úpravy jsou podrobně popsány v následujícím textu.

3.1.1 Posloupnost (sekvence)

Posloupnost je nejjednodušším typem algoritmu. Jedná se o řadu jednotlivých „příkazů“, které na sebe bezprostředně navazují. V tomto případě není žádný rozdíl mezi klasickým vývojovým diagramem a navrhovanou aplikací. Následující obrázek ukazuje příklad posloupnosti, ve které robot provede jeden krok dopředu, položí diamant a poté udělá další krok dopředu. Na obrázku je znázorněn algoritmus jak pomocí klasického vývojového diagramu, tak jeho interpretace v navrhované aplikaci.

(a) vývojový diagram (b) aplikace

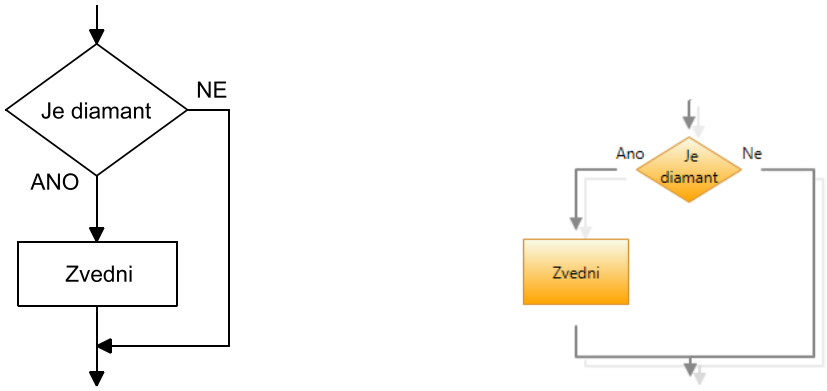
Obrázek 3.1: Zápis posloupnosti pomocí vývojového diagramu a pomocí aplikace Karel VD

3.1.2 Větvení (rozhodování)

Pro větvení se v algoritmu často používá slovo podmínka. Podmínka je ovšem pouhou částí algoritmu větvení – je jejím začátkem. Pro podmínku se ve vývojových diagramech používá kosočtverec, do kterého se zapisuje tzv. výroková formule. Horní vrchol kosočtverce slouží jako vstup do větvení. Pravý vrchol kosočtverce je negativní větev, tj.

provádí se, pokud daná formule neplatí. Dolní vrchol kosočtverce je kladná větev, která se provádí, pokud výroková formule platí.

Následující obrázek ukazuje příklad větvení, ve kterém robot testuje, zda se na místě, na kterém stojí, nachází diamant. Pokud ano, robot diamant zvedne; pokud ne, robot neprovede nic. Na obr. 3.2(a) je tento postup vyjádřen klasickým vývojovým diagramem. Z důvodů výše uvedených je větvení v navrhované aplikaci zapisováno odlišným způsobem. V tomto případě se jedná pouze o nepatrnou úpravu, která je spíše stylistická. V aplikaci je vše téměř stejné jako u klasického vývojového diagramu, kromě kladné větve, která je z dolního vrcholu přesunuta do levého rohu kosočtverce, viz obr. 3.2(b).



(a) vývojový diagram (b) aplikace

Obrázek 3.2: Zápis větvení pomocí vývojového diagramu a pomocí aplikace Karel VD

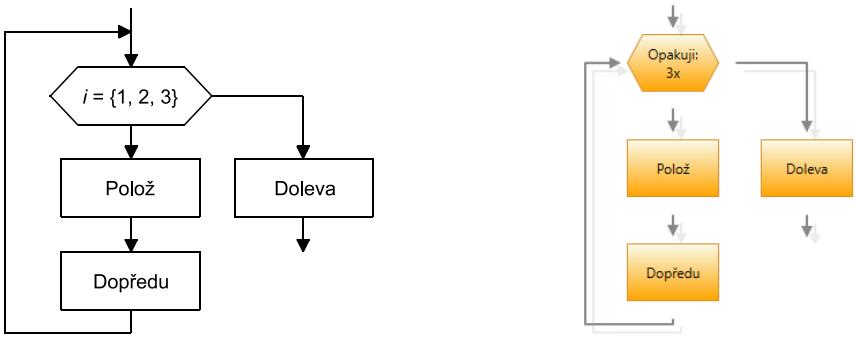
K algoritmu větvení je třeba ještě zmínit jednu velmi častou chybu. Pokud dojde někde v algoritmu k rozvětvení, pak se posléze musí ony dvě větve opět spojit v jednu cestu, na což začátečníci často zapomínají.

3.1.3 Cyklus s pevným počtem opakování

Cyklus s pevným počtem opakování začíná ve vývojovém diagramu šestiúhelníkem. V něm je obvykle uvedena takzvaná řídicí proměnná a výpis hodnot, kterých bude nabývat. Do těla cyklu míří šipka, která vychází z prostřed spodní hrany šestiúhelníka. Touto cestou pokračuje algoritmus v tom případě, že řídicí proměnná ještě nenabyla všech definovaných hodnot. Po provedení těla cyklu se šipka vrací zpět (vlevo) před značku šestiúhleníka, řídicí proměnná nabude další definované hodnoty a cyklus se opakuje. Po-

kud již není další hodnota pro řídicí proměnnou, cyklus končí a pokračuje se šipkou, která vychází z pravého vrcholu šestiúhelníka.

Následující obrázek ukazuje příklad cyklu s pevným počtem opakování, ve kterém robot položí na tři políčka po jednom diamantu a po skončení cyklu se otočí vlevo vbok. Na obr. 3.3(a) je tento postup vyjádřen klasickým vývojovým diagramem. Jak již bylo zmíněno, aplikace by po provedení těla cyklu hledala vše znovu. Proto se v navržené aplikaci konec těla cyklu vede do levého vrcholu šestiúhelníka, to je do jiného místa, než je vstupní cesta do cyklu, viz obr. 3.3(b).



(a) vývojový diagram (b) aplikace

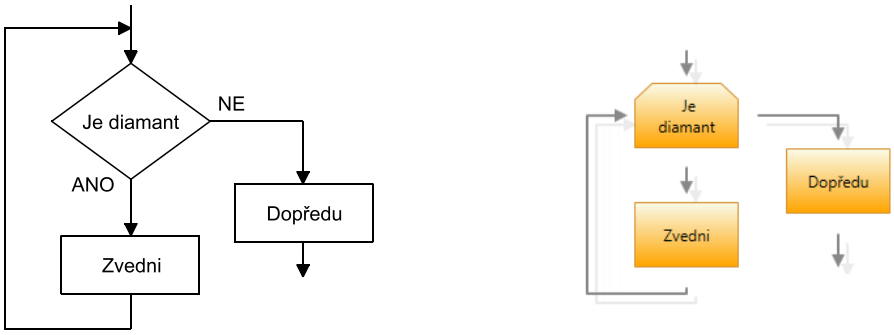
Obrázek 3.3: Zápis cyklu s pevným počtem opakování pomocí vývojového diagramu a pomocí aplikace Karel VD

3.1.4 Cyklus s podmínkou na začátku

Cyklus s podmínkou na začátku je dalším typem cyklu, který se používá v programování. Ve vývojových diagramech se značí kosočtvercem, podobně jako větvení, do kterého se zapisuje výroková formule. Horní vrchol kosočtverce slouží jako vstup do cyklu. Pravý vrchol kosočtverce je negativní větev, tj. provede se, pokud daná výroková formule neplatí a cyklus tím končí. Dolní vrchol kosočtverce je kladná větev, která se provádí, dokud výroková formule platí. Tato větev směřuje do těla cyklu, odkud se vrací zpět před kosočtverec podobně jako u cyklu s pevným počtem opakování.

Následující příklad ukazuje použití cyklu s podmínkou na začátku. V tomto algoritmu kontroluje robot, zda je na políčku, na kterém stojí, diamantu. Pokud ano, pak ho robot zvedne a opakuje tuto kontrolu. Pokud robot posbíral diamanty všechny, pak cyklus skončí a robot přejde na další políčko. Na obr. 3.4(a) je tento postup vyjádřen kla-

sickým vývojovým diagramem. Aplikace používá místo kosočtverce šestistěn. Tato změna oproti klasickému vývojovému diagramu byla provedena ze dvou důvodů. První důvod je stejný jako u cyklu s pevným počtem opakování. Druhým důvodem bylo komplikované centrování textu, který se v aplikaci vypisuje ve značkách vývojového diagramu. Text přesahoval okraje kosočtverce a provedení správného zalamování textu by bylo příliš komplikované. Na obr. 3.4(b) je uvedena interpretace cyklu s podmínkou na začátku, která byla naprogramována v aplikaci.



(a) vývojový diagram (b) aplikace

Obrázek 3.4: Zápis cyklu s podmínkou na začátku pomocí vývojového diagramu a pomocí aplikace Karel VD

Kapitola 4

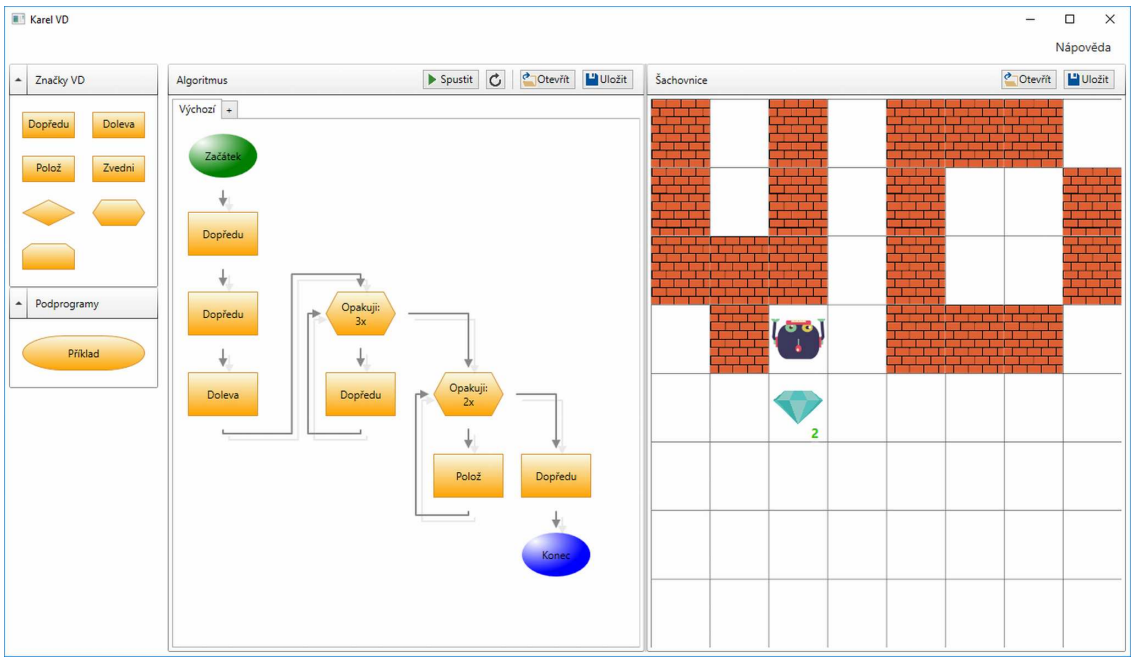
Vývoj aplikace Karel VD

Tato kapitola se zabývá vysvětlením postupů řešení, které byly použity při programování této aplikace. Je zde popsáno, jak je provedeno propojení jednotlivých částí aplikace a řízení referencí objektů pomocí návrhového vzoru *Inversion of Control* (IoC). Dále jsou v této kapitole ukázky a vysvětlení ovládání některých základní prvků aplikace. V neposlední řadě je zde naznačeno možné rozšíření aplikace, které přinesou další verze. Tento text je psaný pro verzi aplikace 1.1, proto se mohou některé věci v dalších verzích aplikace lišit.

4.1 Části aplikace

Navržená aplikace, jejíž celkový náhled je na následujícím obrázku, byla z důvodu přehlednosti, možnosti vylepšování a rozšiřování stávajících funkcí, rozdělena na čtyři části, které byly vloženy do hlavního okna. Tyto čtyři části jsou:

- `Algoritmus` – panel pro skládání vývojových diagramů (algoritmů),
- `Šachovnice` – mapa, kde se robot může pohybovat a kde vykonává příkazy dle vytvořeného vývojového diagramu,
- `Značky VD` – okno s geometrickými obrazci, ze kterých může uživatel skládat vývojový diagram,
- `Podprogramy` – okno s již vytvořenými algoritmy, které uživatel uložil, a které je možné použít v dalších úlohách jako „podprogramy“.



Obrázek 4.1: Celkový náhled na aplikaci Karel VD

Výhodou tohoto rozdělení je to, že se do aplikace může kdykoliv přidat rychle jakákoliv nová část, která nezasáhne do již existujících částí. Velikost jednotlivých částí se nastaví v hlavním okně aplikace (`MainWindow.xaml`), kde se zároveň přiřadí pozice dané části (okna) v aplikaci.

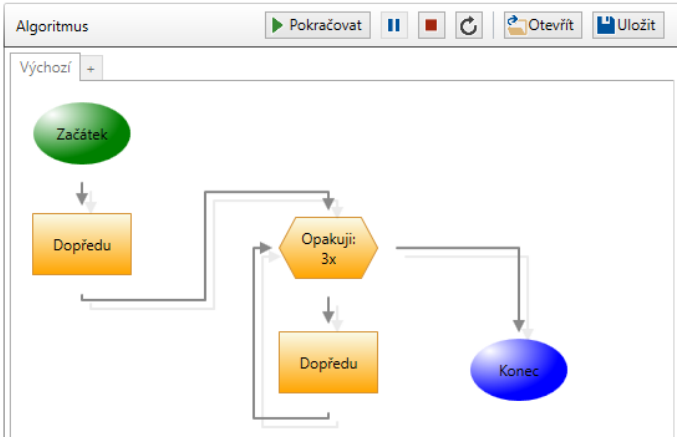
4.1.1 Panel Algoritmus

V tomto panelu může uživatel skládat svůj vlastní algoritmus, podle kterého se bude následně řídit Robot. Tvorba vývojového diagramu je velmi intuitivní. Jednotlivé značky se do této části přetahují pomocí myši z okna *Značky VD* nebo z okna *Podprogramy*. Spojování značek se provádí tak, že se čára táhne při stisknutém levém tlačítku myši od jedné značky k další značce. Jakmile se u druhé značky objeví propojovací čtvereček, tlačítko myši je možné uvolnit a tím dojde ke spojení dvou značek. Uživatel zde také může vytvořený algoritmus uložit nebo naopak otevřít již dříve uložený. Nakonec je tu možnost spuštění algoritmu, jeho pozastavení a ukončení.

Tato část navržené aplikace vznikla z aplikace *Diagram Designer – Part 4*, která byla stažena z portálu ⟨www.codeproject.com⟩. V této aplikaci bylo již vyřešeno kreslení vývojových diagramů (značky a jejich spojování). Tento kód byl tedy převzat a následně modifikován pro použití v navržené aplikaci. Tyto stažené zdrojové kódy jsou pod licencí

The Code Project Open Licence (CPOL), to je, mohou být staženy a modifikovány pro vlastní potřeby.

Základem části Algoritmus je panel `DesignerCanvas`, který dědí ze třídy `Canvas`. Jednotlivé panely jsou poté zobrazovány pomocí záložek tzv. `TabControl`. Tyto záložky řeší pomocí bindingu vlastnost `Designers`, která odkazuje na generickou kolekci typu `ObservableCollection`. Tato kolekce uchovává objekty typu `DesignerCanvasItem`. Tento objekt obsahuje 3 vlastnosti: řetězec `Header`, logickou hodnotu `Selected` a objekt typu DesignerCanvas `Designer`. `Header` se pomocí bindingu načte do hlavičky `TabItem`. To samé platí pro `Designer`, který má ovšem svůj odkaz v `ContentTemplate`. Vzhled této části je vidět na obr. 4.2.



Obrázek 4.2: Aplikace: náhled části Algoritmus

Přidání jednotlivých záložek, viz obr. 4.2, se řeší v objektu `TabControl` pomocí události `SelectionChanged`. Pokud je kliknuto na značku plus, otevře se vlastní dialogové okno `TextDialog.xaml` pro zvolení názvu, které je ve Visual Studiu ve složce `Dialogs`. Tento dialog hlídá délku textu, tedy minimálně 5 a maximálně 25 znaků. Odpověď se uloží do veřejné vlastnosti a poté je možné nastavit název nové záložky. Po vytvoření je záložka automaticky vložena do kolekce pomocí metody `Insert` na předposlední místo v panelu Algoritmy.

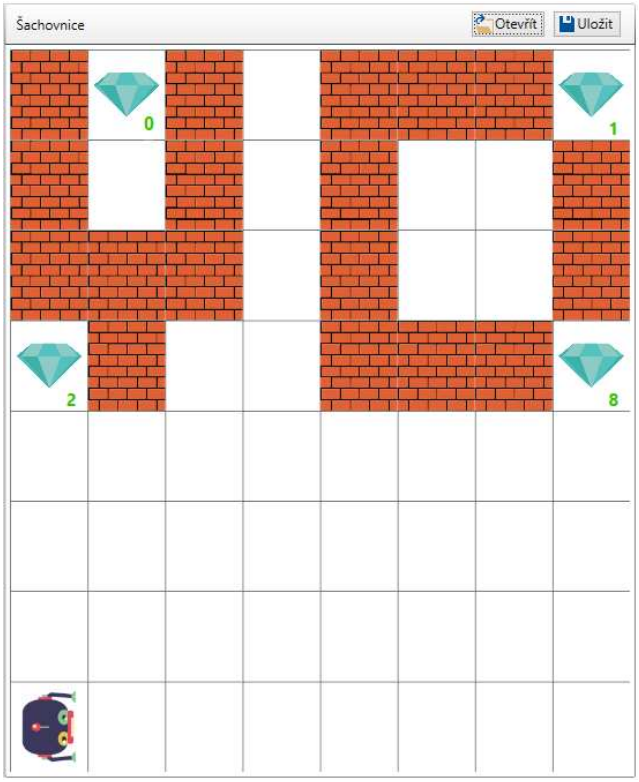
V panelu Algoritmy jsou také použity tlačítka (`Buttons`), které využívají příkazy (`Command`). V `UserControl` jsou definovány veřejné statické proměnné, které se inicializují v konstruktoru a jsou typu `RoutedCommand`. Tlačítka v sobě dále mají animace pomocí `XAML`u. Jedná se o tzv. `Storyboard`, který se poté volá na příslušné akce.

Ukládání a nahrávání uloženého vývojového algoritmu se řeší pomocí `XML`. Při ukládání se serializují jednotlivé objekty umístěné v panelu, tedy značky vývojového diagramu

a jejich propojení. Uloží se pouze základní vlastnosti objektů, které jsou nezbytné pro jejich znovu vytvoření (ID, název, pozice, připojení...). Při otevření tedy stačí deserializovat XML soubor, vytvořit jednotlivé objekty a přidat je do nově vytvořeného panelu.

4.1.2 Šachovnice

Šachovnice je část aplikace, ve které se pohybuje Robot. V šachovnici mohou být předem umístěny zdi, to je objekty, přes které Robot nemůže projít, nebo diamanty, to je objekty, které má Robot sbírat (případně pokládat). Zeď se do šachovnice umísťuje kliknutím na levé tlačítko myši, diamant kliknutím na pravé tlačítko myši. Pro odstranění těchto objektů ze šachovnice se použije opačné tlačítko. Vytvořenou mapu zde lze uložit nebo naopak načíst již dříve uloženou. Náhled této části aplikace je na následujícím obrázku.



Obrázek 4.3: Aplikace: náhled části Šachovnice

Šachovnice je řešená pomocí kontrolky `UniformGrid` zatím pevně nastavené na 8 řádků a 8 sloupců. V budoucnu toto půjde dynamicky měnit v nastavení aplikace. Kliknutí, které přidává, respektive odebírá zeď nebo diamant ze šachovnice je řešeno pomocí třídy

`ItemsControl`, která je nad objektem `UniformGrid`. Překrytí těchto dvou objektů není nejvhodnější řešení a bude upraveno v další verzi aplikace.

Šachovnice je ovládána pomocí třídy `ChessBoard`. Tato třída obsluhuje právě třídy `UniformGrid` a `ItemsControl`. Obsahuje privátní dvourozměrné pole typu `ChessBoard-Item`. Pole se musí, po inicializaci a následném naplnění prvky, seřadit kvůli systému souřadnic. `UniformGrid` skládá prvky zleva do prava a odshora dolů, tj. souřadnice y je nahoře 0 a dole 7 (pro šachovnici 8×8). Protože v navržené aplikaci má dolní levý roh souřadnice $[0, 0]$, je třeba upravit souřadnici y. Seřazení se provádí pomocí rozšiřující metody. Touto metodou je rozšířeno dvourozměrné pole. Tato metoda je pojmenována `Reverse2D` a nachází se v souboru `ExtensionMethods.cs`. Po seřazení funguje správně souřadnicový systém a díky vytvořenému indexu se pracuje s třídou `ChessBoard` jako s polem. Práce s touto třídou je popsaná později při vysvětlování třídy `Robot`.

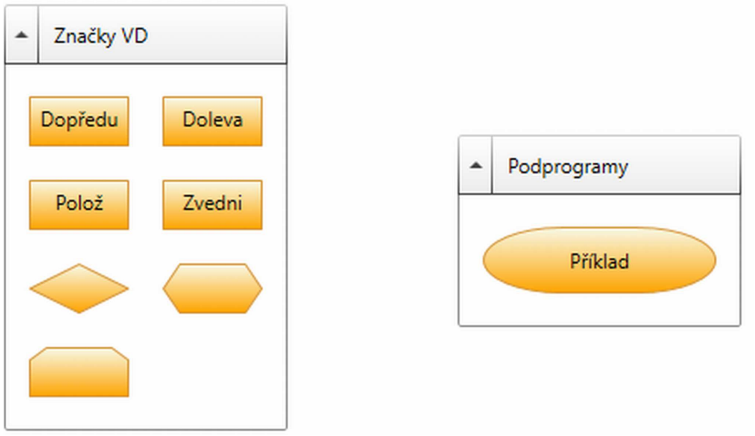
Tlačítka fungují na stejném principu jako u panelu pro kreslení vývojových diagramů. Tato část obsahuje pouze dvš tlačítka a to pro uložení a načtení mapy. Ukládání je zde vyřešeno pomocí binárního zápisu. Při ukládání se uloží velikost šachovnice a následně se zapíšou postupně všechny prvky. U jednotlivých prvků se zapisuje pouze typ (zeď, diamant, robot), úhel natočení prvku (rotace robota) a pokud je pole diamant, pak počet diamantů. Při načtení šachovnice tedy stačí vytvořit novou herní plochu a tu projít cyklem a nastavit vlastnosti podle nahraných dat.

4.1.3 Značky VD a podprogramy

V okně Značky VD jsou uživateli k dispozici základní značky vývojových diagramů (příkazy, větvení, cykly) a v okně Podprogramy se uživateli objeví algoritmy, které jako podprogramy uložil. Náhled této části aplikace je na obr. 4.4.

Značky vývojových diagramů jsou řešeny pomocí aplikace *Diagram Designer: Part 4* z portálu ⟨www.codeproject.com⟩. Ve složce `Resources` je vytvořená třída `ToolBox`, která dědí ze třídy `ItemsControl`. Toto je ze zdrojů načítáno jako obsah do kontrolky `Expander`, která zajišťuje, že tyto položky mohou být skryty. Jednotlivé položky jsou tvořeny z `Gridu`, které právě dávají dohromady obrazce vykreslené pomocí vektorové grafiky a pomocného textu.

Podprogramy jsou řešené také pomocí `ToolBoxu`, ale už nejsou vytvářeny ve zdrojích ale pomocí jazyka `XAML` hlavního okna. Do tohoto `ToolBoxu` se pomocí bindingu nahraje kolekce s jednotlivými podprogramy. Aby správně fungoval vzhled položek, musí se vytvořit v `Resources` `DataTemplate` a nastavit za cíl typ objektu na třídu `SubProgram`. To



Obrázek 4.4: Aplikace: náhled části Značky VD

zajistí, že objekty typu `SubProgram` budou dědit nastavený vzhled v `DataTemplate`. Do podprogramů se položky přidávají dynamicky a to pokud uživatel při ukládání zvolí, že chce daný vývojový diagram uložit také jako podprogram.

4.2 Řízení referencí obejktů v aplikaci

Při objektovém programování je někdy potřeba zavolat funkci, která nepatří danému objektu. Každý objekt se má starat o svoji část, ale někdy je třeba zavolat nějakou funkci jiného objektu, který je za danou oblast aplikace odpovědný. Z tohoto důvodu musí mít objekt referenci na jiný objekt, se kterým chce pracovat. Tomu se říká předávání závislostí a programátoři vymysleli mnoho způsobů, jak reference předávat. Samozřejmě existují způsoby špatné, pracné a správné. Mezi špatné způsoby patří například statika, singleton, service locator. Pracným způsobem je předávání referencí manuálně, což znamená psát například konstruktor, který bude mít 10 referencí při každé inicializaci. To je ale velmi pracné a neefektivní. Jako správný postup při předávání závislostí se uvádí Inversion of Control (IoC) – Dependecy Injection (DI), který je použit v této aplikace (Microsoft, a. s., 2018).

Inversion Of Control je návrhový vzor, ve kterém již objekty neřídí své závislosti samy, ale objekty jsou řízeny pomocí IoC. Reference jsou řízeným objektů přidávány automaticky. Dependecy Injection, česky injekce závislostí, je konkrétní implementace principu IoC. Jsou různé možnosti, jak závislosti touto cestou nastavovat. Tato aplikace využívá setter injection, což znamená, že DI kontejner při vytváření objektu zkontroluje,

zda obsahuje vlastnosti, které mají nastavený atribut, který požaduje referenci na nějaký objekt. Pokud ano, pak se DI kontejner pokusí najít tento objekt nebo ho vytvoří a zaregistruje, čímž nastaví všechny požadované vlastnosti vytvářeného objektu. Následující kód je ukázkou z DI kontejneru vytvořeného v této aplikaci.

```
public object GetInstance(string fullNameOfClass)
{
    if (!services.TryGetValue(fullNameOfClass, out object instance))
    {
        Type type = Type.GetType(fullNameOfClass);
        instance = Activator.CreateInstance(type, true);
        services.TryAdd(fullNameOfClass, instance);
        InsertDependence(type, instance);
    }
    return instance;
}

private void InsertDependence(Type reflection, object instance)
{
    foreach (PropertyInfo property in reflection.GetProperties())
    {
        foreach (var attribute in property.GetCustomAttributes(false))
        {
            if (attribute is DependenceAttribute)
            {
                string customName = ((DependenceAttribute)attribute).CustomName;

                if (string.IsNullOrWhiteSpace(customName))
                {
                    property.SetValue(instance,
                        GetInstance(property.PropertyType.FullName));
                }
                else
                {
                    property.SetValue(instance, GetInstance(customName));
                }
                break;
            }
        }
    }
}
```

Metoda `GetInstance` zajišťuje zkontroluje, zda nějaký objekt s daným jménem již existuje. Pokud ano, vrátí referenci na daný objekt. Pokud ne, vytvoří tento objekt pomocí třídy `Activator`. Konkrétně se používá statická metoda této třídy `CreateInstance`. Její první parametr je typ objektu, který se má vytvořit a protože některé třídy mají privátní konstruktor, aby bylo zajištěno, že v aplikacu budou opravdu pouze jednou, tak druhý parametr se nastaví na true. Druhý parametr právě povoluje vytvoření objektu s privátním kontruktorem (.NET magie). Po vytvoření objektu je objekt přidán do kolekce a zavolána metoda `InsertDependence`. Tato metoda pomocí reflexe zkontroluje všechny vlastnosti daného objektu. Pokud jsou vlastnosti nastaveny pomocí atributu `Dependence`, tak se zjistí typ vlastnosti a volá se opět metoda `GetInstance`. Toto způsobí rekurzi, která tak nastaví a vytvoří všechny potřebné objekty.

4.3 Třída Robot

Třída Robot je základem spojení robota a vývojového diagramu. V této třídě se z vývojového diagramu dostávají instrukce, aby aplikace věděla, jaké metody má následně zavolat. Třída je rozdělena pomocí klíčového slova `partial` do dvou souborů. `Robot.cs` obsahuje metody pro akci robota, to je krok dopředu, otočení doleva, zvednutí diamantu a položení diamantu. Také jsou zde metody, které ovládají cykly a podmínky. Druhá část je v souboru `Robot.Commands.cs`, kde je definovaný konstruktor a pomocí příkazů (`Commands`) funkce pro tlačítka spustit, pozastavit, zastavit a reset.

V části, která je v `Robot.Commands.cs`, se nachází metoda, která pohání celý pohyb robota. V asynchronní metodě `Start_ExecuteAsync` je cyklus, který se opakuje dokud hodnota proměnné `_CanContinue` je nastavená na hodnotu true. Nastavení této proměnné zajistí, že se opakování zastaví. To se používá například při pozastavení aplikace nebo když aplikace vyhodí výjimku při pohybu robota. Jako první při každém cyklus se kontroluje, zdali je již nastavený `StartItem`. Toto je proměnná, kde se uchovává objekt z `DesignerCanvas`, který je určen podle postupu ve vývojovém diagramu. To zajišťuje vlastnost s názvem `StartItemIsSet`, která je nastavená třeba v případě, kdy se opouští cyklus nebo podmínka, kde se následující položka již nastaví. Pokud tato hodnota není nastavena, vyhledá se pomocí připojení následující položka ve vývojovém diagramu a ta se nastaví. Pokud následující položka je shodná s položkou, která ukončuje vývojový diagram (modrá elipsa), cyklus skončí a algoritmus je tímto dokončen. Pokud algoritmus

stále neskončil, cyklus má nastavenou prodlevu, než provede další úkon robota. Tato prodleva je nastavena proto, aby si mohl uživatel prohlédnout pohyb robota. Dále se volá samotná funkce pro danou značku. Asynchronní metoda `CallMethod` zavolá danou metodu podle jména, které má nastavena každá značka vývojového diagramu. Metody se při inicializaci třídy Robot uloží do statické kolekce. Aplikace používá kolekci pro normální a pro asynchronní metody (asynchronní metody jsou potřeba například pro cyklus). Tyto metody se získají v metodě `GetMethods`, která pomocí rekurze projde všechny metody třídy Robot. Metoda, která bude mít nastavený atribut `RobotMethod`, bude přidána do dané kolekce. Jestli jde o normální nebo asynchronní metodu se nastavuje v atributu ve vlastnosti `MethodType` pomocí výčtového typu.

V části v `Robot.cs` jsou pouze metody ovládající přímo robota na šachovnici. Metody pracují se třídou `ChessBoard`, kde nastavují pro jednotlivá pole šachnovice, co je potřeba. For a while cyklus zde používají podobný systém jako v metodě `Start_ExecuteAsync`. Systém je téměř stejný až na pár drobných úprav. Cykly potřebují výrokovou formuli (logickou funkci) nebo počet opakování. Tato formule nebo počet opakování se bere přímo ze značky, kterou metoda dostane z vývojového diagramu. Základní značky používají stejnou třídu a to `DesignerItem`. Cykly a podmínka používají své vlastní třídy, které z `DesignerItem` dědí. Ve zdrojích je pomocí jazyka `XAML` dědění vzhledu, pokud položka potřebuje grafickou úpravu nebo přidat něco do vývojového diagramu. Pomocí tohoto systému se přidává v cyklech ozubené kolečko symbolizující nastavení cyklu. Tyto údaje se poté získají z dané položky a podle toho se cyklus ovládá.

Kapitola 5

Závěr

V této práci byly nejprve porovnány některé stávající aplikace pro výuku algoritmického myšlení (původní aplikace Robot Karel profesora Richarda Pattise včetně její novějších verzí a aplikace Scratch). Na základě tohoto srovnání byla navržena a v jazyce C# naprogramována aplikace nová. Ta spojuje vývojové diagramy, kterými se obvykle algoritmy znázorňují, s grafickým rozhraním, ve kterém si uživatel může ověřit správnost navrženého algoritmu.

V nově navržené aplikaci se podařilo naprogramovat následující části. Uživatel má možnost vytvářet algoritmy pomocí základních struktur (posloupnost, větvení, cyklus) a tyto zakreslovat pomocí vývojových diagramů, kterými se následně v grafické části aplikace řídí Robot Karel. Ten umí základní povely, to je krok dopředu, otočení doleva, zvednutí diamantu a položení diamantu. Tyto povely umí vykonávat v rámci cyklu s pevným počtem opakování (`for`) nebo cyklu s podmínkou na začátku (`while`) podobně, jako tomu bylo u verze profesora Pattise. Aplikace též uživateli umožňuje navržený algoritmus spustit, pozastavit, zastavit a restartovat, dále pak algoritmus uložit nebo načíst dříve uložený. V grafické části aplikace může uživatel vytvářet „města“, do kterých může vkládat dva objekty: zdi a diamanty. Tato města může uživatel opět ukládat nebo načítat dříve uložená.

Kreslení vývojových diagramů je v základní podobě sice vyřešeno ale tuto část aplikace bude třeba v budoucnu upravit, neboť spojování značek vývojových diagramů občas špatně reaguje a geometrie vykreslených spojnic při složitějších algoritmech překrývá značky vývojových diagramů, což nepůsobí pěkným dojmem. V dalších verzích aplikace je také plánována možnost změny jazyka aplikace na angličtinu, přidání proměnné do vývojových diagramů a zlepšení stávajících funkcí.

Literatura

Jedlička, O. (2006), Robot Karel: Vývovjové prostředí [online]. [cit. 2018-02-03], ⟨http://karel.oldium.net/⟩.

Krejsa, J. (2014), Výuka základů programování v prostředí Scratch, (Diplomová práce), Jihočeská univerzita v Českých Budějovicích, Pedagogická fakulta, České Budějovice.

Lecián, T. (2017), Robot Karel pro výuku programování, (Bakalářská práce), Univerzita Tomáše Baťi ve Zlíně, Fakulta aplikované informatiky, Zlín.

Microsoft, a. s. (2018), Developer Network: Inversion of Control [online]. [cit. 2018-03-01], ⟨https://msdn.microsoft.com/en-us/library/ff921087.aspx⟩.

Pattis, R. E. (1981), *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal*, Praha: John Wiley & Sons. ISBN 0471089281.

Pečínková, J. (2009), *Algoritmizace*, Prostějov: Computer Media. ISBN 978-80-7402-034-6.

Schenk, C. (2009), MiKTeX [online]. [cit. 2018-02-01], ⟨http://www.miktex.org/⟩.

Scratch (2003), Scratch – Imagine, Program, Share [online]. [cit. 2018-02-03], ⟨https://scratch.mit.edu/⟩.

Synovcová, M. (1989), *Martina si hraje s počítačem*, Praha: Albatros.

Velebil, J. (2017), Diskrétní matematika: Text k přednášce [online]. [cit. 2018-03-17], ⟨http://math.feld.cvut.cz/velebil/⟩.

Seznam obrázků

Přílohy

Obsah přiloženého CD

K této práci je přiloženo CD s následující adresářovou strukturou.

- `Karel`: původní aplikace Robota Karla
- `Karel VD`: nově navržená aplikace
- `Karel VD C`: zdrojové kódy nově navržené aplikace
- `LaTeX`: text práce v prostředí LaTeX
- `Poster`: plakát o projektu
- `Prezentace`: prezentace projektu
- `Zdroje`: PDF s použitou literaturou
- `Martinek_2017_2018.pdf` – text práce ve formátu PDF

Použitý software

- **LaTeX 2$_\varepsilon$** ⟨http://www.miktex.org/⟩
- **Visual Studio Community 2017** ⟨http://www.visualstudio.com/⟩
- **WinEdt 5.3** ⟨http://www.winedt.com/⟩

Software z výše uvedeného seznamu je buď volně dostupný, nebo jeho licenci toho času vlastní Vyšší odborná škola, Střední škola, Centrum odborné přípravy, Sezimovo Ústí, Budějovická 421, kde autor téhož času studoval a vytvořil tuto práci.