UNIVERZITA PARDUBICE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2009 Ivo Snoza

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Interaktivní dynamické vývojové diagramy

Ivo Snoza

Bakalářská práce

2009

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Katedra informačních technologií
Akademický rok: **2008/2009**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Ivo SNOZA**

Studijní program: **B2646 Informační technologie**

Studijní obor: **Informační technologie**

Název tématu: **Interaktivní dynamické vývojové diagramy**

Zásady pro vypracování:

Program bude umožňovat sestavit algoritmus pomocí základních značek vývojových diagramů. Uživatel bude mít možnost postupně skládat vývojový diagram ze značek pro čtení vstupních hodnot, přiřazovacích příkazů, rozhodování, cyklů s podmínkou na začátku, na konci nebo s pevným počtem opakování, výpis výstupních hodnot. Jednotlivé strukturované příkazy mohou být do sebe vnořovány. Dále program bude obsahovat vyřešené vývojové algoritmy: Největší společný dělitel, Mocnina, Převod do dvojkové soustavy, Faktoriál, Kvadratická rovnice. V těchto programech budou dialogová okna, která umožní interaktivní vstupy, zobrazení postupného krokování. Součástí programu bude Nápověda.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

PŠENČÍKOVÁ, JANA Algoritmizace. [s.l.] : [s.n.], 2007. 120 s. ISBN 80-86686-80-9. ČSN ISO 5807

KNUTH, DONALD E Umění programování,Základní algoritmy. [s.l.] : [s.n.], 2008. 672 s. ISBN .978-80-251-2025-5

Vedoucí bakalářské práce: **Ing. Soňa Neradová**
Katedra softwarových technologií

Datum zadání bakalářské práce: **15. ledna 2009**

Termín odevzdání bakalářské práce: **15. května 2009**

Univerzita Pardubice
Česká republika

L.S.

doc. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan
vedoucí katedry

V Pardubicích dne 31. března 2009

Prohlášení autora

Prohlašuji:

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

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

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 14.8.2009

Ivo Snoza

Anotace

Cílem této práce je vytvořit aplikaci pro sestavování algoritmů pomocí základních značek vývojových diagramů a následnou vizualizaci průchodu tímto diagramem. V první části je čtenář obeznámen se základními pojmy a symboly vývojových diagramů použitých v aplikaci. Dále je rozebrán návrh aplikace a její funkčnost.

Klíčová slova

vývojové, diagramy, algoritmy, interaktivní, dynamické

Title

Interactive dynamic flowcharts

Annotation

Purpose of this work is to create application for assembling algorithms by basic symbols of flowcharts and then visualization of through this diagram. In the first section reader is acquainted with basic terms and symbols of flowcharts used in application. Next part is focused on application layout, structure and function.

Keywords

flowcharts, algorithms, interactive, dynamic

Obsah

Seznam obrázků

1. Úvod

Vývojové diagramy - část učiva, která neminula snad nikoho, kdo se kdy ve škole učil algoritmizaci nebo programování. Ne vždy byla tato látka tou nejoblíbenější. Zcela jistě to bylo z několika důvodů. Kreslení diagramů na papíry, škrtání, gumování, obtížná představa fungování nakresleného algoritmu.

Tato práce je zaměřena právě na vytvoření zjednodušujícího prostředku pro tvorbu algoritmů pomocí základních značek vývojových diagramů a dodržování norem při jejich tvorbě. Aplikace je navržena ve vývojovém prostředí Borland Delphi 7.0.

Teoretická část této práce je zaměřena na popis a úděl symbolů, které byly zahrnuty do aplikace. Také zde budou vysvětleny zásady, které s jejich tvorbou souvisí.

V praktické části budou vysvětleny použité metody pro kreslení a uchovávání vektorových dat, editace diagramu a jeho jednotlivých částí, vyhodnocování algebraických výrazů a samotné fungování procházení navrženým diagramem.

V závěru budou popsány problémy, které provázely tvorbu aplikace, jejich řešení a možnosti v rozšiřování aplikace.

2. Vývojový diagram

Je to typ diagramu, který reprezentuje algoritmus nebo proces zobrazující kroky jako značky různých tvarů. Jejich pořadí je určováno spojnicemi. Odborněji řečeno, jde o symbolický algoritmický jazyk, který je tvořen přesně definovanými značkami. Tyto značky jsou spojovány v celek spojnicemi tak, aby tvořily logickou posloupnost vykonávání. Může být tvořen tak, že daný problém řeší v obecné rovině bez zřetele na speciální vlastnosti konkrétního počítače a programovacího jazyka. Ovšem v praxi se často sestavuje právě na základě vlastností programovacího jazyka.

Vývojové diagramy jsou tedy vhodným nástrojem nejen pro výuku začínajících programátorů, ale i pro publikování algoritmů, které jsou velmi často realizovány v konkrétním programovacím jazyce, což není zrovna vhodná volba pro vysvětlování problémů širší škále programátorů. A navíc každý program je dnes dílem, na který se vztahuje autorský zákon.

Pro kreslení vývojových diagramů platí od roku 1996 česká státní norma ČSN ISO 5807 *„Zpracování informací. Dokumentační symboly a konvence pro vývojové diagramy toku dat, programu a systému, síťové diagramy programu a diagramy zdrojů systému“*.

Norma specifikuje symboly používané v dokumentaci pro zpracování informací a poskytuje návod pro jejich použití:

a) ve vývojových diagramech toku dat;
b) ve vývojových diagramech programu;
c) ve vývojových diagramech systému;
d) v síťových diagramech systému.

Aplikace tvořená v rámci této práce je zaměřena na vývojové diagramy programu. Ty se skládají z následujících prvků:

a) symboly zpracování pro vlastní operace zpracování, včetně symbolů definujících stanovený tok, který má být dodržen při zachování logických podmínek;
b) spojnice indikující tok řízení;
c) zvláštní symboly pro usnadnění čtení a zápisu vývojového diagramu.

2.1. Symboly vývojových diagramů

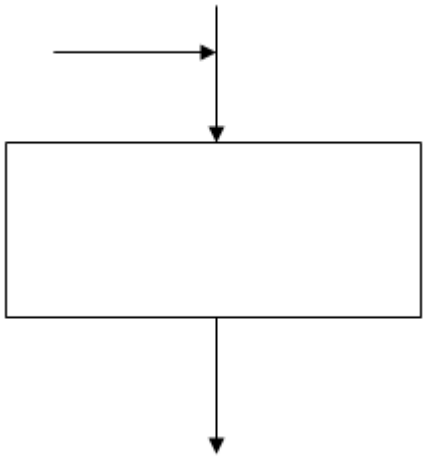
Symboly vývojových diagramů představují grafické značky přesně definovaného významu a byly navrženy tak, aby bylo možné dovnitř vepsat krátký popis, který zpřesňuje funkci symbolu v diagramu.

2.1.1. Zpracování

Tento symbol představuje jakýkoliv druh funkce zpracování, například provádění definované operace nebo skupiny operací, jejichž výsledkem je změna hodnoty, formy nebo umístění informací nebo stanovení, který tok z několika směrů se má sledovat. Vstupů může mít několik, ale nejčastěji se vstupující spojnice spojí do jedné, která pak vstupuje do symbolu, nejčastěji to bývá shora. Výstup má vždy jen jeden a bývá vyveden dolů.

Obrázek 1: Symbol zpracování

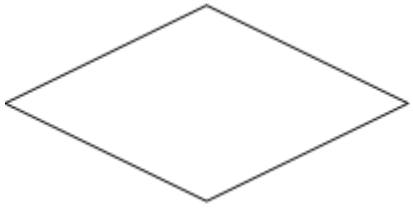
Příklad:



Obrázek 2: Symbol zpracování - příklad

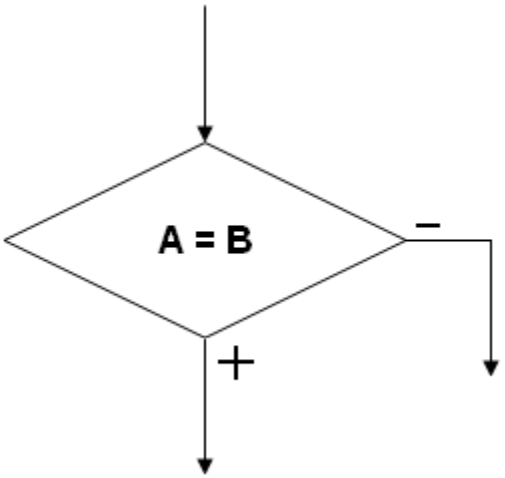
2.1.2.Rozhodování

Symbol představuje rozhodovací nebo přepínací funkci. Má jeden vstup, který může být výslednou spojnicí z několika jiných, a několik možných výstupů. Aktivuje se ten výstup, u kterého je splněna podmínka. Nemělo by se stát, že počet pravdivých podmínek nebude roven jedné. Výstupy mohou být označeny popisky, které jasně definují podmínku, při které je výstup aktivován. Tento symbol mívá nejčastěji dva výstupy, obecně může mít ale i výstupy tři nebo jeden větvený výstup.

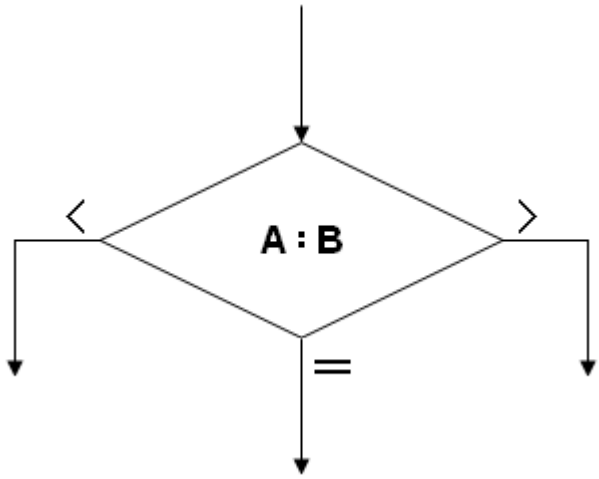


Obrázek 3: Symbol rozhodování

Příklady:



Obrázek 5: Symbol rozhodnutí - příklad 1



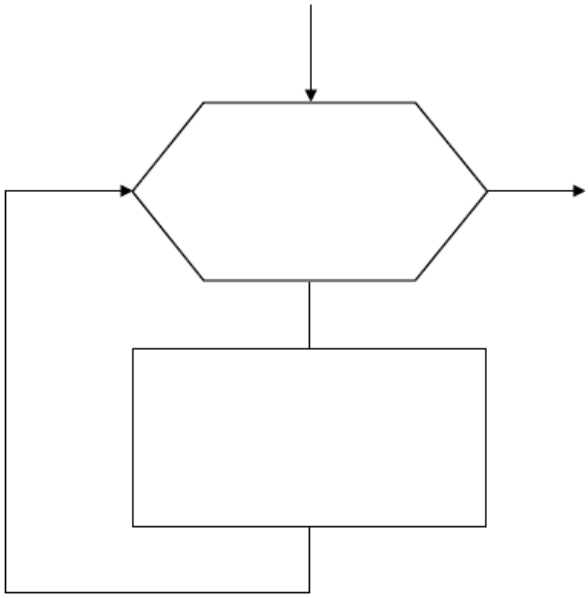
Obrázek 4: Symbol rozhodnutí - příklad 2

2.1.3.Příprava

Symbol přípravy představuje úpravu nebo modifikaci činnosti, která mění vlastní postup následné činnosti, např. vyjmenování hodnot, kterých nabývá proměnná cyklu. Symbol má dva vstupy, jeden sekvenční, druhý pro návrat po prvedení daného bloku operací a dva výstupy, jeden vstupující do daného bloku operací a druhý sekvenční, který pokračuje do další části programu.

Obrázek 6: Symbol příprava

Příklad:



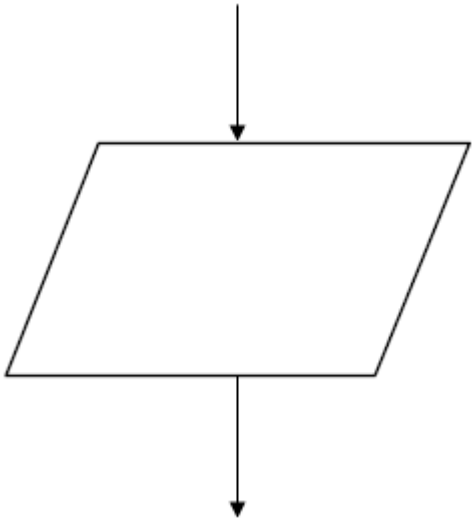
Obrázek 7: Symbol přípravy - příklad

2.1.4.Data

Tento symbol představuje data, nosič dat není specifikován. Tyto data tedy můžou být vstupní či výstupní, tj. dodání dat pro zpracování nebo zpracovaná data do požadované formy výstupu. K tomuto symbolu je možné připojit symbol, který bude typ dat charakterizovat. Symbol má jeden vstup a jeden výstup.

Obrázek 8: Symbol data

Příklad:



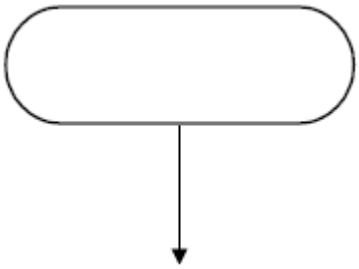
Obrázek 9: Symbol data - příklad

2.1.5.Mezní značka

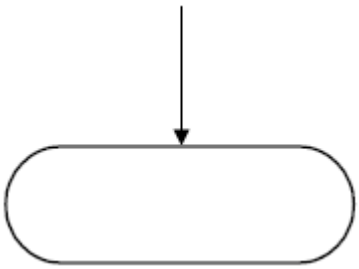
Tato značka symbolizuje vstup z vnějšího prostředí, tedy počátek algoritmu, nebo ukončení algoritmu a výstup zpět do vnějšího prostředí. Může také značit začátek nebo konec podprogramu. Mezní značka počátku má jeden výstup a je bez vstupu. Mezní značka konce má jeden vstup a je bez výstupu.

Obrázek 10: Mezní symbol

Příklady:



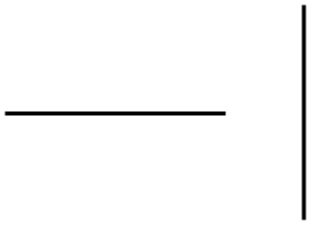
Obrázek 11: Mezní symbol - příklad 1



Obrázek 12: Mezní symbol - příklad 2

2.1.6.Spojnice

Tento symbol představuje tok dat nebo řízení a slouží ke spojování symbolů vývojového diagramu. Jeho grafická podoba je vodorovná nebo svislá čára. Standartní směr toku je zleva doprava a shora dolů. Pokud to takto není, je vhodné čáru opatřit šipkou pro zvýšení názornosti.



Obrázek 13: Symboly spojnice

2.2. Konvence

I přes to, že vývojové diagramy a jejich kreslení se řídí podle norem a zásad, jsou zde jisté výjimky, které není nutné dodržovat, ale jejich užívání je běžné a uznávané.

2.2.1.Symboly

Grafický vzhled symbolů byl navržen tak, aby bylo možné umístit do každého z nich popisek, který upřesní funkci symbolu v diagramu. Tyto popisky by měly být psané tak, aby se daly číst zleva doprava a shora dolů. Mohou být vyjádřeny slovně nebo matematickými symboly. Měly by být stručné a jasné.

Symboly by měly být v celém diagramu stejně velké a mezi sebou mít, pokud možno, stejné vzdálenosti.

2.2.2.Spojnice a spojování

Nejen spojnice, jejichž směr není klasický, tj. zleva doprava nebo shora dolů, je pro větší názornost možné opatřit šipkou.

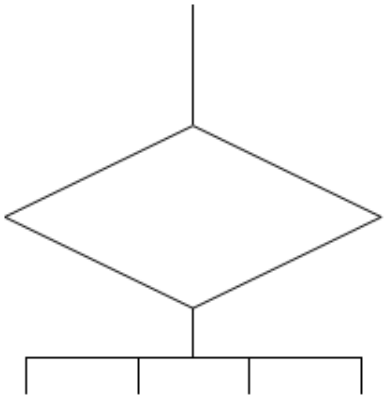
Nedoporučuje se křížení spojnic, nicméně není zakázané. Pokud ke křížení dojde, neznamená to žádný vzájemný vztah ani žádnou logickou souvislost.

Několik spojnic může být připojeno na jednu výstupní. Mělo by platit, že jednotlivá spojení budou od sebe odsazena a nemělo by docházet v jednom bodě k více než jednomu spojení.

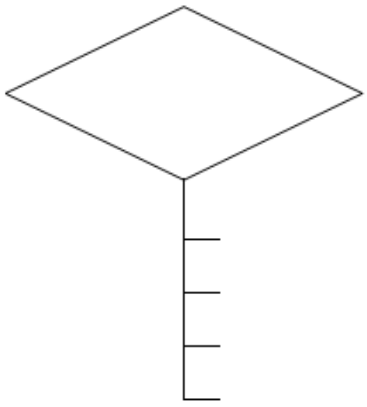
2.2.3. Zvláštní konvence – vícenásobné výstupy

Toto se týká symbolu rozhodování, kde počet výstupů záleží na počtu podmínek. Máme-li symbol s dvěmi nebo třemi podmínkami, je možné vést výstupy z rohů symbolu spojnicemi k jiným symbolům. Pokud ale nastane situace, že podmínek máme více, výstup vedeme ze spodní části symbolu a ten se následně větví do potřebného počtu spojnic. Tuto větev je možné znázornit horizontálně nebo vertikálně.

Příklady:



Obrázek 15: Symbol rozhodování s horizontální větví



Obrázek 14: Symbol rozhodování s vertikální větví

3. Implementace návrhu diagramu

Před samotným psaním takto celkem rozsáhlé aplikace, je nutné si dobře promyslet systém, kterým budou uchovávána data tak, aby bylo co nejjednodušší je ukládat či načítat ze souboru, zpracovávat je do grafické podoby, modifikovat. Jak jsem již psal v úvodu, pro tvorbu aplikace jsem si vybral vývojové prostředí Borland Delphi 7.0, které využívá programovací jazyk Object Pascal.

3.1. Vytvoření plátna

Plocha, na které jsou vykreslovány všechny značky, popisky či spojnice, by měla být dost prostorná na to, aby se na ní vešly i rozsáhlejší projekty. A protože běžné monitory by nebyly svými rozměry dostačující, aby se na ně vešlo celé kreslící plátno, použil jsem následující metodu. Nejprve jsem vytvořil v designeru posouvací okno (TScrollBox) s názvem *SB1*. Dále jsem vytvořil proměnnou *Papir* typu TImage, což je také standartní komponenta v Delphi, a jako rodiče jsem jí přiřadil právě TScrollBox.

```
Papir.Parent := SB1;
Papir.OnMouseDown := SB1.OnMouseDown;
Papir.OnMouseUp   := SB1.OnMouseUp;
Papir.OnMouseMove := SB1.OnMouseMove;
```

Události *OnMouseDown*, *OnMouseUp* a *OnMouseMove* jsem přiřadil totožné od ScrollBoxu. Těmito příkazy jsem dosáhl posuvné plochy jakékoli velikosti a zachycení událostí myši nad plátnem (*Papir*). Nyní je možné definovat metody obsluhující tyto tři nejpodstatnější události nad naším kreslicím plátnem.

3.2. Prvky na plátně

Teď když máme vytvořené kreslicí plátno, je nutné vymyslet takové typy dat, které budou vhodné pro sestavování vlastních diagramů. V závorkách je uveden název datového typu deklarovaného ve zdrojovém kódu aplikace.

3.2.1. Souřadnice (TBod)

Každý prvek zobrazený na plátně musí mít nějaké souřadnice. V Delphi jsou již předdefinované nějaké typy souřadnic. Ale žádná z nich mi nevyhovovala a tak jsem si vytvořil vlastní, abych věděl jak správně s ní pracovat a ovládat. Nicméně tato struktura není zase tak složitá, stejně tak i její deklarace není složitá a mohu jí zde napsat.

```
Type TBod = class
        x : longint;
        y : longint;
        constructor create;
        destructor destroy;
      end;
```

3.2.2. Popisek (TDesc)

Popisek se využívá u výstupů symbolu rozhodování. Atribut **axis** typu TBod, kterým je pouze ukazatel na souřadnice spojového bodu, spojený s atributem **offset** dávají dohromady polohu bodu. Offset je zde proto, aby se popisek mohl umístit do vhodné polohy vzhledem ke spojovému bodu.

3.2.3. Spojový bod (TFunc_point)

Spojový bod je také jedním ze základních kamenů sestavování diagramů. Je to bod, ke kterému se připojuje spojnice, proto tedy spojový bod. V podstatě každý ze symbolů, který na plátně vytvoříme má minimálně jeden a například symbol spojnice je z nich dokonce tvořen. U klasických symbolů jsou jejich pozice pevně dané a definují vstup či výstup symbolu.

Má více atributů, které je důležité si vysvětlit, aby byla lépe pochopena podstata spojových bodů. Následující atributy udávájí to, jak se tento prvek bude na plátně chovat a jevit.

```
avail   : boolean;
hidden  : boolean;
movable : boolean;
inout   : boolean;
```

Atribut **avail** odvozen od slova avaiable (dostupný) je užitečný k tomu, že není vždy potřebné, aby spojový bod byl dostupný uživateli, ale je nutné, aby existoval. Příklad můžeme hledat u symbolu rozhodování s větveným výstupem, kde právě větev je tvořena také ze spojových bodů. Právě tam je pořeba, aby body tvořící kostru větve, kromě posledního bodu, nebyly v žádném případě přístupné.

Další atributy jsou nám k něčemu pouze v případě, že atribut **avail** je nastaven u tohoto spojového bodu na *true*.

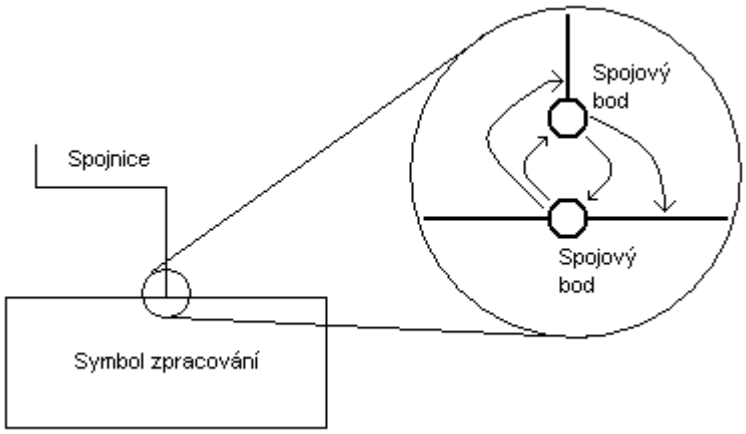
Atribut **hidden** (schovaný) nám udává viditelnost spojového bodu. V drtivé většině případů je nastaven na *false* a pouze když je k tomuto bodu připojena spojnice, je nastaven na *true*, tedy skrytý.

Atribut **movable** (schopný pohybu) je použit u symbolu spojnice. Je to atribut, který udává pohyblivost spojového bodu po plátně. Tato změna souřadnic je možná tažením myši po plátně. U klasických symbolů, kde mají spojové body své místo je tento atribut nastaven na *false*, protože není potřeba, aby s ním uživatel mohl pohybovat.

Atribut **inout** (in/out) udává, zda tento spojový bod reprezentuje vstup nebo výstup. Je-li hodnota nastavena na *true*, pak jde o vstupní spojový bod. Při *false* jde o výstupní spojový bod.

Těmito čtyřmi atributy máme tedy definovány vlastnosti spojového bodu. Dále něco o vztazích mezi spojovými body a symboly.

Spojový bod se vždy připojuje do páru. Je tedy nutné, aby každý spojový bod měl informaci, s jakým druhým spojovým bodem je spojen a navíc jakému symbolu tento spojový bod patří.



Obrázek 16: Ukázka připojení informací spojových bodů

Informace o druhém spojovém bodě se nám hodí k získávání souřadnic tohoto bodu, protože jedním ze spojovaných symbolů je vždy spojnice. A když si uvědomíme, že souřadnice prvního a posledního bodu spojnice jsou závislé na souřadnicích bodu ze strany symbolu, na který je spojnice připojena, jsou pro nás v aplikaci při vykreslování nepostradatelné.

Informace o připojených symbolech se nám bude hodit při procházení diagramem, kdy jsou potřeba při přecházení ze symbolu na symbol.

```
    conn_object : Pointer;
    conn_fpoint : TFunc_point;
```

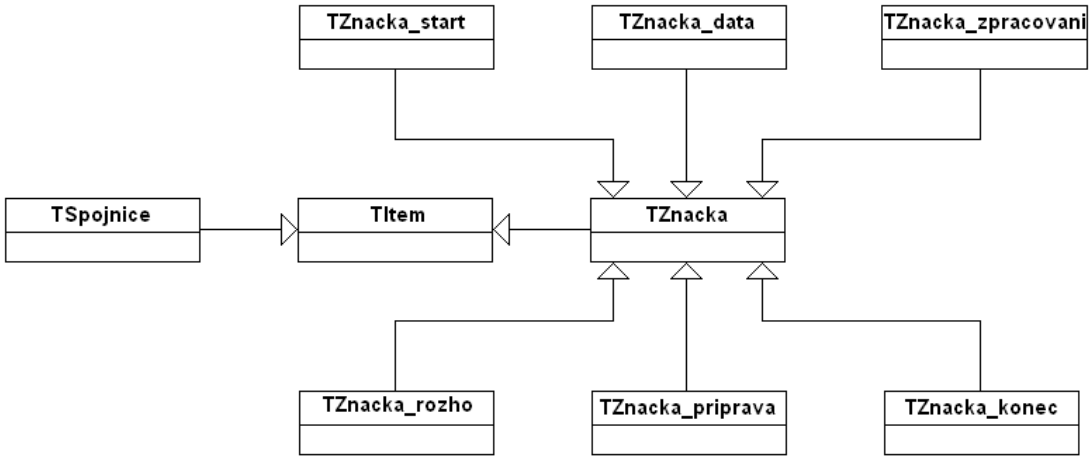
Poslední atribut spojových bodů je **caption** (popisek). Je důležitý z hlediska vzhledu diagramu, protože se doporučuje popisovat výstupy u symbolu rozhodování. Takový výstup je v této aplikaci reprezentován právě popisovaným spojovým bodem.

```
    caption : TDesc;
```

3.2.4.Symboly

Z pohledu uživatele jsou symboly na plátně to hlavní, s čím pracuje. Základní třídou všech symbolů je v programu třída nazvaná *TItem*, která má společné atributy a metody pro všechny dále tvořené symboly. Protože ve vývojových diagramech jsou si spojnice a ostatní značky rovny a tudíž by se zdálo být logické, že si budou rovny i ve struktuře tříd. Spojnice jsou ale velmi specifickým prvkem ve vývojových diagramech a mají mnoho rozlišných vlastností. V hierarchické struktuře jsou tedy potomky třídy *TItem* třídy *TSpojnice* a *TZnacka*. Od *TZnacka* pak dále dědí vlastnosti

všechny jednotlivé symboly. Jsou to třídy *TZnacka_start*, *TZnacka_konec*, *TZnacka_rozho*, *TZnacka_data*, *TZnacka_priprava*, *TZnacka_zpracovani*. Struktura je graficky znázorněna na obrázku 17 pomocí UML diagramu.



Obrázek 17: UML diagram tříd symbolů

3.3. Spoje

Spoje se tvoří spojnicemi, které mají vždy počátek v některém ze symbolů kromě spojnice a končí v jakémkoliv symbolu včetně spojnice. V aplikaci se spoje tvoří v daném pořadí. Nejprve se tedy vybere výstupní bod některého ze symbolů, následně se tvaruje spojnice a nakonec je zavedena do vstupu jiného symbolu. Tím se zajistí správná orientace spojnice a také to, že bude vždy mít oba konce připojeny. Je to vcelku jednoduchá logika, za kterou se ovšem skrývá dosti komplikací.

3.3.1. Připojování spojnic

Postupoval jsem od základních metod pro připojení spojnice ze vstupu do výstupu. Tyto metody se nacházejí v hlavní části zdrojového kódu nazvaném *u_vyvdiag*. Pro tvorbu jakékoliv nové spojnice mám privátní pomocný atribut nazvaný *new_cara*.

```
    new_cara : TSpojnice;
```

Je-li tedy vybráno kreslení spojnic a myší kliknuto na výstupní spojový bod symbolu, je volána metoda *set_connFromOutput*. Zde se vytvoří nová instance typu *TSpojnice* a ukazatel na ní je uložen do *new_cara*. Dále tato metoda připojí spojnici

k symbolu, ale zatím spojový bod na symbolu nepřipojuje ke spojnici. Spojením je myšleno vyměnění informací, jak bylo popsáno v kapitole o spojových bodech. Toto jednostranné spojování je z toho důvodu, že když si uživatel rozmyslí kreslení spojnice a zruší ho, pak se nemusí atributy spojového bodu na symbolu vracet do původního stavu a spojnice je destruována.

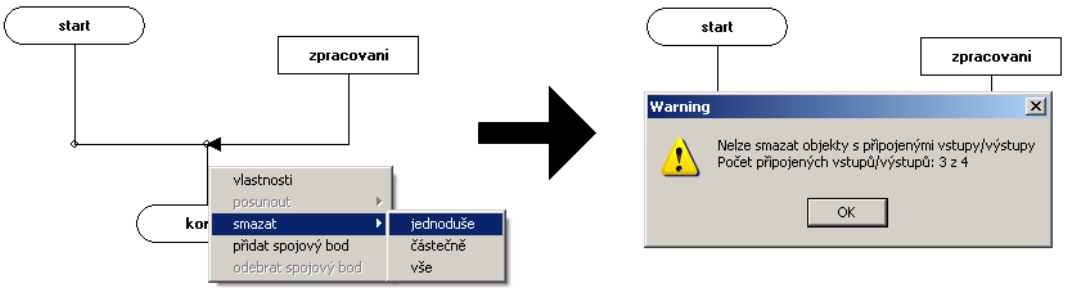
Pokud tedy uživatel dokončí tvorbu spojnice tím, že ho připojí na vstupní bod jakéhokoliv symbolu, pak je volána metoda *set_connToInput*, která připojí zbývající spojové body. První se tedy připojí spojový bod symbolu, ze kterého spojnice vychází. Následně se připojují oba spojové body u konce spojnice.

Nyní tedy oba symboly, které spojnice spojuje, mají informace o spojnici a o tom, kam a odkud vede. Tímto způsobem je možné tvořit i připojení spojnice na spojnici a dělat tak někdy vcelku složitou spojnicovou strukturu.

3.3.2.Mazání spojnic – jednoduchá metoda

Z toho důvodu, že spojení může existovat i mezi spojnicemi navzájem, bylo nutné vymyslet více způsobů jejich mazání či mazání jiných symbolů, ke kterým jsou spojnice připojeny.

Tento způsob je nejjednodušší a nejprostší. Pokud mažeme spojnici či jiný symbol, ke kterému je připojena jiná spojnice, pak nám aplikace nedovolí tuto spojnici smazat a oznámí nám, že nelze mazat objekt, který má obsazen některý ze vstupů či výstupů. U spojnice může být připojen pouze vstupní a výstupní spojový bod, což je podmínka existence spojnice. Způsob takového mazání symbolů může být někdy velmi zdlouhavý, protože je potřeba mazat značky postupně po jedné.



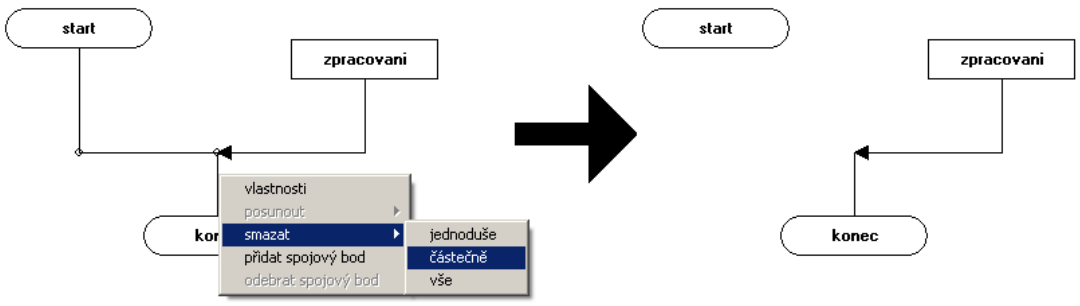
Obrázek 18: Ukázka mazání jednoduchým způsobem

Příklad na obrázku 18 znázorňující symboly start a konec spojené první spojnicí a symbol zpracování, který je spojen k první spojnici. Pokud tedy chceme smazat první spojnici mezi startem a koncem, program nám ohlásí chybu, protože k této spojnici je připojena jiná.

3.3.3. Mazání spojnic – rozšířená metoda

Více sofistikovaným způsobem je rozšířené neboli částečné mazání. Tímto způsobem smažeme spojnici od počátku postupně po částech až do místa, kde je na spojnici připojená jiná spojnice. V případě, že spojnice, na kterou aplikujeme tento způsob mazání, nemá na sebe připojené jiné spojnice, maže se spojnice celá.

Algoritmus tohoto odstraňování je takový, že spojnici, kterou chceme odstranit, cyklem projíždíme postupně po spojových bodech, kterými je tvořena a kontrolujeme, zda je k tomuto bodu připojena jiná spojnice. Z kapitoly o připojování spojnic víme, že informací k tomu máme dostatek. Pokud tedy nalezneme bod, ve kterém je spojen s jinou spojnicí, pak zbytek bodů spojnice, kterou chceme odstranit postupně přidáme ke spojnici, kterou jsme našli. Nesmíme zapomenout na to, že ke zbytku původní spojnice mohly být připojeny jiné. A těm musíme, stejně jako symbolu, ke kterému je připojena, změnit jejich informace o připojené spojnici. Kdybychom nechali původní a z paměti odstranili tu spojnici, kterou chceme odstranit, pak by mohlo docházet k chybám při přístupu do paměti.

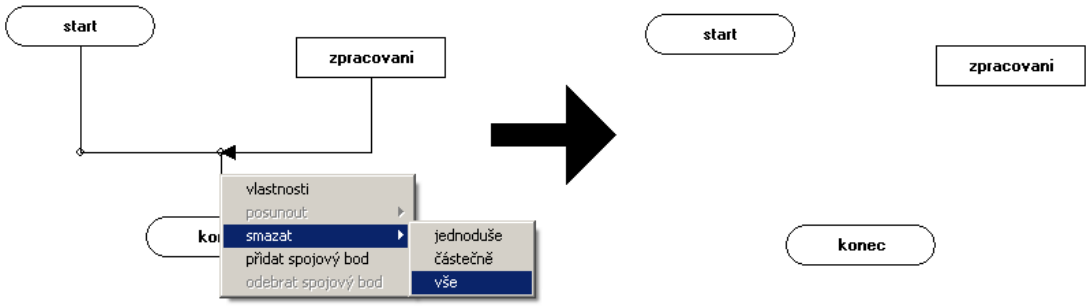


Obrázek 19: Ukázka mazání rozšířeným způsobem

3.3.4.Mazání spojnic – úplná metoda

Třetí způsob smazání spojnice je tak trochu „tvrdým“ způsobem, kdy se smaže celá struktura spojnic připojená na spojnici, kterou chceme odstranit.

K této metodě odstranění jsem použil datovou strukturu zásobník (*TStack*). Zásobník inicializuji tím, že do něj vložím první prvek a to spojnici, kterou chci smazat. Dále už v cyklu, který končí tím, že je zásobník prázdný, provádím průchod spojnicí, která je na vrcholu zásobníku, po jednotlivých spojových bodech. Pokud k nějakému bodu z těch na spojnici je připojena jiná spojnice, pak se ta spojnice vloží na vrchol zásobníku. Pokud ke spojnici, která je na vrcholu zásobníku, není připojena jiná spojnice, pak se tato nejprve odpojí od symbolu, ze kterého vychází, a od spojnice, ke které je připojena. Následně se spojnice destruuje.



Obrázek 20: Ukázka mazání úplným způsobem

3.3.5.Editace spojnice

Připojování a odstraňování spojnic jsou důležité funkce pro návrh diagramu. Občas se ale stane, že při návrhu spojnice uživatel zapomene na nějaký ten spojový bod, ke kterému pak chce připojit spojnici jinou, nebo mu nějaký ten spojový bod přebývá a chce ho odstranit.

Odebrání spojového bodu ze spojnice je o něco jednodušší. Z pozice kurzoru myši určíme, nad kterým spojovým bodem se nachází. Ten potom najdeme v listu spojových bodů, který má každý objekt včetně spojnic, a z tohoto seznamu ho smažeme a spojnici znovu vykreslíme.

Pokud jde o přidávání spojového bodu na spojnici, pak musíme zjistit, nad kterou částí spojnice je kurzor myši, abysme ji pak mohli správně zařadit do listu spojových bodů. To uděláme tak, že cyklem projedeme list spojových bodů a určíme podle souřadnic sousedních bodů a souřadnicí kurzoru, zda je kurzor nad danou částí spojnice. Zde jsem využil znalosti z lineární algebry, kdy jde v podstatě o určení polohy bodu a úsečky, kde úsečka je určena dvěma krajními body. Pak už stačí vytvořit nový spojový bod a dát mu souřadnice kurzoru myši.

3.4. Vykreslování prvků na plátno

3.4.1. Uchování symbolů v paměti

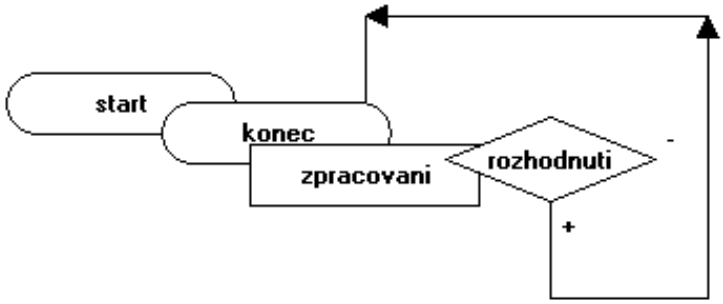
Jako vhodný kontejner pro uchovávání symbolů v paměti počítače jsem si vybral už předem připravený datový typ *TList*. Do tohoto kontejneru se ukládají ukazatele na objekty v paměti. TList má dost vhodných metod k jeho obsluze. Mezi těmi, které jsem často používal jsou:

- **add** – přidá prvek na konec seznamu;
- **delete** – odebere prvek ze seznamu na daném indexu;
- **insert** – vloží prvek do seznamu na daný index.

Prvky jsou v kontejneru řazeny tak, že poslední vytvořený objekt se přidá na první pozici seznamu. Pokud se jedná o spojnici, je přidávána až na úplný konec seznamu.

3.4.2. Vykreslování

Jednoduchá metoda pro vykreslení prvků na plátno spočívá v průchodu seznamu od posledního prvku po první a volání metody *Draw* z třídy *TItem* na každý prvek. Jak vyplývá z předešlé kapitoly o uchovávání prvků v kontejneru, nejprve jsou vykresleny spojnice a pak teprve značky. To znamená že prvek, který je první v seznamu bude vykreslen jako poslední. Tím je zajištěno, že spojnice nikdy nebudou vykresleny nad symboly a také to, že symbol, který je v seznamu první, je vykreslen nad všemi ostatními.



Obrázek 21: Ukázka vykreslování

Z obrázku 21 a předchozího výkladu je patrné, v jakém pořadí se prvky v seznamu nacházejí. Nejprve je vykreslena spojnice, která se nachází na poslední pozici v seznamu, dále jsou vykreslovány symboly v pořadí start, konec, zpracování, rozhodnutí.

3.4.3. Změna pořadí vykreslování

Někdy je potřeba změnit pořadí vykreslování symbolů. Například, pokud se překrývají. To by se ovšem ve vývojových diagramech nemělo stávat, ovšem nápady uživatelů jsou různé. Proto při návrhu je možné měnit pořadí vykreslování symbolů. Jde pouze o přemisťování prvků v seznamu symbolů. Je možné posunout o jeden symbol dopředu nebo dozadu a úplně dopředu nebo úplně dozadu. K tomu jsem využil metdou *Exchange* z třídy kontejneru *TList*. Při výměně musí být aplikovány kontroly hranic seznamu a také, abychom nezaměňovali symboly se spojnicemi.

3.5. Práce se souborem

Pro práci se soubory, pokud v aplikaci pracujeme s objekty, se ve vyšších programovacích jazycích používá tzv. serializace objektu, což je způsob, jak z objektu vytvořit sekvenci bajtů, která se pak dá uložit na médium a zpětně načíst a vytvořit původní objekt. Toto realizovat v Delphi ovšem není úplně nejlehčí a proto jsem se rozhodl zvolit si vlastní způsob, který možná není úplně tak optimální, ale budu mu dobře rozumět. Jde o ukládání klasickým způsobem ve formě textového souboru.

3.5.1. Ukládání do souboru

Je nutné si uvědomit, co vše se musí do souboru uložit pro to, aby se z uložených dat dal zpětně složit celý diagram včetně vlastností symbolů. Dále je potřeba realizovat metody, kterými se z dat v souboru diagram složí.

Začal jsem jednoduchým průchodem seznamu objektů a do souboru jsem vkládal typy uložených symbolů. Na začátek každého takového řádku jsem přidal znak ^ pro jednodušší identifikaci při načítání. Dále jsem u každého objektu uložil jeho atributy a to tak, že na každý řádek jeden. Formát řádku nesoucí hodnotu atributu je

```
název_atributu:hodnota
```

Tyto řádky nemají žádný specifický první znak. Název atributu tedy musí být jedinečný.

Některé symboly provádějí výpočty, které jsou uloženy ve formě klasické rovnice nebo algebraického výrazu. Například symbol zpracování těchto může mít v podstatě neomezené množství. Ty se pak ukládají speciálně ve formátu a to do souboru až za všechny atributy.

```
ExprList {
::výraz_1
::výraz_2
..
::výraz_n
} ExprList
```

Obecné atributy pro všechny symboly se ukládají do souboru pomocí metody *SaveToFile*. Specifické potom metodou *SaveSpecify*.

Spojové body se u klasických symbolů ukládat nemusejí, protože jsou vytvářeny při konstrukci symbolu a jsou statické a nelze je během editace diagramu měnit.

Naopak u spojnic, které jsou tvořeny výhradně ze spojových bodů, je potřeba ukládat tyto body do souboru. U ukládání spojnic do souboru nastává ovšem podstatný problém. Jak již bylo v předchozích kapitolách napsáno, tak spojnice má vždy počátek a konec v jiném symbolu. Počátek tedy vždy v klasickém symbolu a

konec může být i na spojnici. Když si ale uvědomíme, že takovéto spojování je realizováno pomocí ukazatelů na dynamické objekty v paměti, pak nelze tedy ukládat do souboru například adresu ukazatele nebo něco podobného, protože při příštím vytvoření objektu do paměti může mít a na 99% bude mít úplně jinou adresu než při prvním vytvoření. Z tohoto důvodu jsem přešel k řešení, kdy každý objekt má svoje jedinečné identifikační číslo. K tomu jsem si vytvořil novou třídu s názvem *TSequence* mající vlastnost *peak*, které je možné nastavit počáteční hodnotu a která nám vždy vrátí nové a unikátní *ID* pro objekt.

Dále je třeba si do souboru uložit odkaz na konkrétní spojový bod na symbolu, ze kterého spojnice vychází nebo do kterého vstupuje. Připojení ke spojovým bodům je v programu realizováno také pomocí ukazatelů, ale zde se stejný problém jako u symbolů dá vyřešit jinak a jednodušeji. Protože na každém symbolu je pevně daný počet spojových bodů, které se nemohou v seznamu spojových bodů konkrétního symbolu nijak přesouvat, pak můžeme do souboru uložit pozici spojového bodu v seznamu.

Spojových bodů na spojnici je vždy více než jeden a je tedy potřeba je nejak strukturovat. Seznam spojových bodů tedy v souboru začíná řádkem

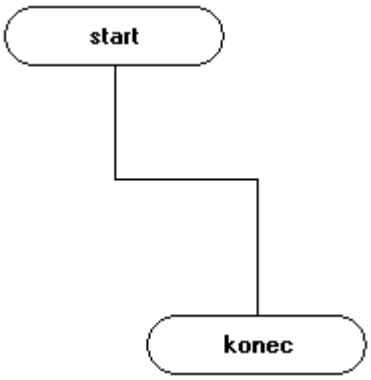
```
FPList{
```

a končí řádkem

```
}FPList
```

V tomto seznamu jsou pak ukládány jednotlivé spojové body včetně jejich atributů. Jeden spojový bod je také ohraničen a v souboru vypadá takto

```
func_point{
x:230
y:198
offset_x:0
offset_y:0
}func_point
```



Obrázek 22: Jednoduchý diagram

Na obrázku 22 je znázorněn jednoduchý diagram, kde se nachází pouze dva symboly a jedna spojnice, která je spojuje. Po uložení do souboru bude tento diagram vyjádřen textově následujícně:

^TZnacka_konec
zleva:175
shora:266
id:2
nazev:konec
typ:Konec
vyska:30
sirka:110
barva_okraje:0
barva_pisma:0
barva_vyplne:16777215
^TZnacka_start
zleva:104
shora:112
id:1
nazev:start
typ:Start
vyska:30
sirka:110
barva_okraje:0
barva_pisma:0
barva_vyplne:16777215
^TSpojnice
id:3
from_obj_id:1
from_port_no:0
to_obj_id:2
to_port_no:0
FPList{
func_point{
x:159
y:198
offset_x:0
offset_y:0
}func_point
func_point{
x:230
y:198
offset_x:0
offset_y:0
}func_point
}FPList

Je nutné neopomenout fakt, že pokud spojnice jsou připojeny ke dvěma symbolům, pak při vytváření musí tyto symboly být vytvořeny před tou konkrétní spojnicí. Jinak by docházelo k chybám, protože by se program pokoušel připojovat k objektům, které ještě nebyly vytvořeny. Není tedy třeba ukládat první a poslední spojový bod na spojnici, protože tento bod je připojen k symbolům a má vždy pevně dané atributy, které uživatel nemůže nijak změnit.

To ale není vše, z čeho se celý model skládá. Co ještě nebylo popsáno, ale co se do souboru také musí uložit jsou proměnné a jejich typy, které jsou použité při výpočtech. Ty se ukládají na začátek souboru a každý tento řádek začíná speciálními znaky **?:** a za nimi následuje název proměnné a její typ. Pak to v souboru vypadá následovně:

```
Vars{
?:a as extended
?:b as extended
?:c as extended
?:jmeno as string
?:prijmeni as string
}Vars
```

3.5.2. Načítání ze souboru

Čtení souboru se provádí postupně od prvního po poslední řádek a každý tento řádek se analyzuje a provede se příslušná operace. Data v souboru jsou rozdělena do logických částí u kterých se dá přesně určit, co znamenají.

Pokud tedy začneme s načítáním řádků ze souboru, pak nejprve jsou na řadě proměnné, které se včetně jejich typů zanesou do výpočtového modulu (tento modul je více popsán v následujících kapitolách).

Následují symboly, které se vytvoří do paměti hned s prvním řádkem, který začíná znakem ^ a za ním následuje typ symbolu, který se má vytvořit. Na dalších řádcích v souboru se mu nastavují atributy. Většina symbolů má svoje specifické atributy, které se načítají ze souboru pomocí metody *LoadSpecify*. Té se parametrem předá načtený řádek a podle obsahu tohoto řetězce se provede nastavení atributu.

Jako poslední uložená část v souboru jsou spojnice. Ty jsou určeny spojovými body, kde první a poslední spojový bod není uložen v souboru, protože ho lze

vytvářet bez konkrétních atributů určující souřadnice spojového bodu. Důležité jsou tedy informace odkud a kam vede. To je určeno na prvních řádcích u každé spojnice.

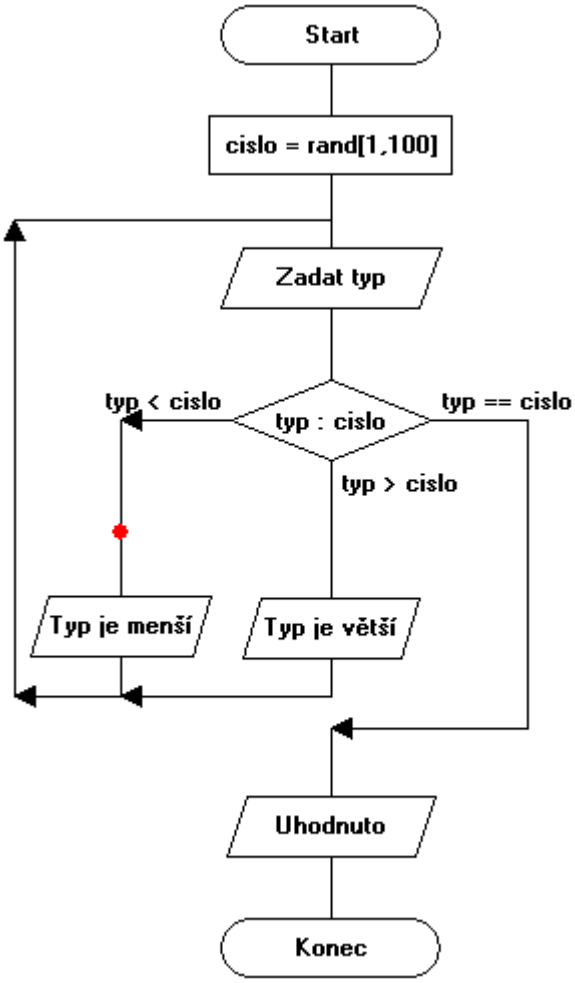
```
^TSpojnice
id:3
from_obj_id:1
from_port_no:0
to_obj_id:2
to_port_no:0
FPList{
...
}FPList
```

Jak je z ukázky vidět, tak na prvním řádku je volána konstrukce spojnice, dále se jí přiřadí identifikační číslo. Následující řádek určuje, že spojnice je vedena ze symbolu s id=1 a spojového bodu 0, tedy úplně prvního v seznamu. Stejně tak i symbol, do kterého spojnice vede. Ten má id=2 a jeho spojový bod je také 0, tedy první v seznamu spojových bodů.

Tyto data se zpracují pomocí metod, které jsou totožné, jako při ručním připojování nově tvořené spojnice. Jsou to metody *set_connFromOutput* a *set_connToInput*. Načtené spojové body se po napojení spojnice na symboly přidávají do seznamu spojových bodů.

4. Průchod diagramem

Počátek průchodu diagramem vždy začíná symbolem *start* a končí symbolem *konec*. Průchod je na plátně znázorněn vybarveným kroužkem. Každý ze symbolu při jeho průchodu provádí nějakou akci. Tyto akce je možné definovat ve vlastnostech každého symbolu. Bývá to vyhodnocování matematických nebo řetězcových výrazů.



Obrázek 23: Ukázka diagramu při průchodu

Na obrázku 23 je ukázka hotového a funkčního diagramu pro hádání čísel. Kroužkem je znázorněn průchod, který zrovna prošel rozhodovacím modulem a směřuje do modulu data, který je výstupní a zobrazí okno s hlášením, že zadané číslo je menší než hádané. Zde jsou vidět i šipky na konci každé části spojnic, která znázorňuje tok dat v nestandartním směru.

4.1. Realizace animace

Je důležité, aby uživatel viděl samotný průchod diagramem a mohl si měnit rychlost animace. To znamená najít vhodnou komponentu, se kterou by toto šlo vytvořit. Použil jsem tedy komponentu *Timer*. Která volá metodu *OnTimer* po daném intervalu. Tento interval je možné měnit a pomocí posuvníku, který jsem zabudoval do aplikace, je ho možné měnit.

Vždy když se zavolá metoda *OnTimer*, pak program posune symbol animace o kousek dál nebo pokud dojde do nového symbolu, pak vykoná jeho specifickou operaci. Po vykonání této operace se rozhodne, kam dál se vybarvený kroužek znázorňující tok dat bude pohybovat. Pokud se jedná o spojnici, pak je trasa jednoznačná.

4.2. Výpočtový modul

Pro vytvoření modulu, který umí vyhodnocovat matematické výrazy, včetně počítání s proměnnými či základními matematickými funkcemi tak, aby spolehlivě fungoval, je velmi náročné a zdlouhavé. Proto jsem zvolil již tento modul hotový a velmi dobře připravený pro použití. Jmenuje se uCalc a jeho licence jsou volné pro použití k nekomerčním účelům a jeho implementace nesmí být stěžejní součástí celé aplikace, což v mém případě není. Tato neplacená verze má jistá omezení, která se ale netýkají využití v mé aplikaci.

Obrovskou výhodou tohoto modulu je, že dokáže z řetězce znaků vypočítat matematický výraz, ale i provádět operace s řetězci. To znamená větší možnosti při sestavování vývojových diagramů a jejich fungování.

Protože ale ne vše, co se týká výpočtů a práci s proměnnými mi v uCalc vyhovovalo, vytvořil jsem si svou vlastní třídu, která tento modul používá tak, jak je v mé aplikaci potřeba.

4.2.1. Datové typy

Jak bylo v kapitole výše zmíněno, modul uCalc umí pracovat nejen s čísly, ale i s řetězci. To znamená dostupnost více datových typů.

Číselných datových typů je mnoho, ale ne všechny jsou si navzájem kompatibilní a mohly by působit uživateli potíže. V aplikaci jsem dal k dispozici pouze jeden číselný reálný datový typ nazvaný *extended*, který je schopný počítat s čísly v rozsahu $3{,}6*10^{-4951}$ až $1{,}1*10^{4932}$. To by mělo vystačit na jakékoliv počty ve vývojových diagramech.

Datový typ řetězce je také jen jeden a to s názvem *string*. Je tvořen jako dynamické pole znaků, takže jeho kapacita je teoreticky neomezená.

4.2.2. Funkce

Pro výpočty jsou k dispozici základní matematické funkce a operátory. Pro číselné typy jsou k dispozi:

- goniometrické funkce (sin, cos, tan, atan) – ve stupních;
- logaritmické funkce (log, exp);
- mocninné funkce a operátory (sqrt, ^);
- operátory sčítání, odčítání, násobení, dělení, celočíselného dělení, unární negace, porovnání čísel;
- logické operátory *and* a *or*;
- a další (abs, ceil, int, frac, sgn, pi, mod, rand, baseconvert);

a pro řetězcové pak:

- operátor sčítání;
- operátory porovnání.

4.3. Vykonávání akcí a trasování

Symboly zpracování, příprava, rozhodování a data mají své specifické funkce, při kterých se pracuje s výpočty či proměnnými. Každý z těchto symbolů má svou metodu, která se stará o zpracování předem definovaných úkonů a další metodu pro nasměrování do dalších segmentů vývojového diagramu. Symboly mezí nebo spojnice nemají význam při výpočtech, ale spíše při směrování.

4.3.1. Symbol zpracování

Může obsahovat víceméně libovolný počet výpočtů a přiřazeních hodnot do proměnných. Tyto operace se vykonávají v takovém pořadí, jak jsou seřazeny v editačním boxu ve vlastnostech symbolu.

Na tomto symbolu je pouze jeden výstup a tak je další postup směrován výhradně na tento.

4.3.2. Symbol rozhodování

Počet podmínek, které rozhodují o dalším postupu v diagramu, může být libovolný. Zpracovávají se také modulem pro výpočty a to v pořadí, v jakém jsou seřazeny.

Pokud je podmínka pouze jedna, pak se podle její pravdivosti rozhoduje směru dalšího postupu. Jsou tedy v tomto případě dva výstupy a z toho jeden pro pravdu a druhý pro nepravdu. Jejich umístění si uživatel může zvolit.

Symbol se dvěmi podmínkami by se neměl vyskytovat, neboť pravdivost těchto podmínek musí vždy určit jednu ze dvou cest, kudy dál v diagramu pokračovat, což by v tomto případě nemuselo být vždy tak. A proto by měla být použita pouze jedna podmínka, která vždy vrátí *pravda* nebo *nepravda*.

Je-li v symbolu definováno více podmínek, pak je další postup směrován na první výstup, u kterého je podmínka pravdivá. Pokud ale žádná z podmínek není pravdivá, pak nastane chyba a průchod diagramem je ukončen.

4.3.3. Symbol příprava

Počet cyklů, který by měl být celočíselný, je zpracováván opět jako výraz, který si definuje sám uživatel. Ale protože by bylo složité určovat už před počítáním, zda výsledek z tohoto výrazu bude celočíselného nebo reálného typu, tak jsem se rozhodl pro takové řešení, že z výsledku, který vzejde z výrazu se vezme pouze celočíselná část.

Pokud je počet podmínek splněn, překročen nebo při jeho počítání dojde k chybě, je další postup směrován na sekvenční výstup, který pokračuje do dalších částí vývojového diagramu. V opačném případě je postup vyveden do bloku, který upravuje další postup vývojového diagramu.

4.3.4.Symbol data

Jde-li o vstupní typ dat, pak se pomocí dialogového okna zadávají postupně všechny hodnoty do proměnné, která byla definována. Jako vstup může být i matematický nebo řetězcový výraz.

Pokud je tento symbol určen jako výstupní, pak se v dialogových oknech postupně zobrazují zprávy, které si uživatel sám definuje. Tyto zprávy by měly usnadnit chápání dosaženého výsledku. Z tohoto důvodu jsem musel vytvořit způsob, kterým budou uživatelé moci definovat zprávy tak, aby v nich bylo možné zobrazit jakýkoliv text doplněný o výpis hodnoty z libovolné proměnné. Řešením je přidání před název proměnné dva znaky $ bezprostředně za sebou a bez mezery před názvem proměnné. Takové výskyty se při výpisu nahradí za hodnotu z proměnné. Počet a typ proměnných je neomezený.

***Příklad**:* Máme proměnnou *delka*, která je typu *extended* a během postupu vývojovým diagramem jí byla přiřazena nebo vypočtena hodnota 5. Pak tedy pro výpis informace o délce úsečky bysme použili formátovaný řetězec ve tvaru:

```
Délka úsečky je $$delka cm.
```

a výsledek, který by se nám zobrazil při vykonání akce na tomto symbolu je:

```
Délka úsečky je 5 cm.
```

4.3.5.Mezní symbol

Ať už se jedná o počáteční nebo koncový symbol diagramu, tak nad ním samotné trasování neprobíhá. Vždy buď začne nebo skončí v jeho jednom vstupním nebo výstupním bodě.

4.3.6. Symbol spojnice

Spojnice sama o sobě nevykonává žádnou akci, ale pouze zprostředkovává přechod mezi jinými symboly.

Vizualizace trasování spojnice se tedy vykonává mezi sousedními spojovými body, ze kterých jsou spojnice tvořeny a vykreslovány. Zde jsem musel znovu využít znalosti z lineární algebry, kdy jsem musel počítat souřadnice vizualizačního prvku (vybarveného kroužku). Výpočty musejí být alespoň tak přesné, aby vizualizace vypadala tak, že probíhá nad spojnicí a zároveň, aby se dala kontrolovat rovnost souřadnice vizualizační značky a bodu, do kterého směřuje. Pokud by tento výpočet nebyl přesný, pak by mohlo dojít k výchylkám od dané trasy, což by mohlo způsobit další nepřesnosti či nefunkčnost průchodu diagramem.

Speciální případ, kdy konec spojnice je napojen na jinou spojnici jsem musel řešit tak, že při vstupu na další spojnici jsem musel zjistit z jakého bodu na ní bylo vstoupeno a od teprve od tohoto bodu pokračovat dál v trasování na spojnici nové pro trasování. V případě, že by se toto neřešilo, pak by při přechodu ze spojnice na další docházelo k tomu, že vizualizační symbol začne trasovat spojnici od počátku místo od bodu vstupu na spojnici.

5. Závěr

Cílem této práce bylo vytvořit aplikaci, která napomůže porozumět začínajícím programátorům při studiu základů algoritmizace. V této aplikaci je možné sestavovat vývojové diagramy libovolné struktury a logiky. Při tvorbě symbolů a celého systému spojování symbolů jsem se snažil dodržovat normy tak, aby jim navržený diagram odpovídal. Ovšem ne vždy to bylo v mých schopnostech či možnostech. Jedná se například o logiku vykreslování spojnic, kdy by měly být výhradně svislé nebo vodorovné.

Při průchodu diagramem si uživatel může volit rychlost animace. Má možnost průchod pozastavit a následně spustit nebo ho zastavit úplně a editovat diagram.

Nejsložitější byl začátek, kdy bylo potřeba vymyslet systém spojování symbolů tak, aby byl lehce použitelný jak pro editaci diagramu, tak i pro následné procházení. Konkrétní nápad jsem dostal teprve až po delší době promýšlení, když jsem si uvědomil všechny souvislosti a nutnosti, které se budou vyskytovat a budou potřeba.

Další potíže jsem měl při hledání modulu, který by byl vhodný pro vyhodnocování výrazů. Naštěstí jsem našel vhodný nástroj, který mě překvapil i svou obecností, co se týká datových typů. Tím se aplikace rozšířila o možnost práce s řetězci, o čemž jsem v počátku vůbec neuvažoval.

Aplikace je navržena tak, že je možné další rozšiřování. Do případného dalšího vývoje by bylo vhodné vytvořit vlastní modul pro výpočty, aby se aplikace stala více nezávislou.

Seznam použité literatury

[1] PŠENČÍKOVÁ, Jana., *Algoritmizace*. [s.l.] : [s.n.], 2007. 120 s. ISBN 80-86686-80-9.

[2] TAUFER, I., HRUBINA, J., TAUFER, J., *Algoritmy a algoritmizace: vývojové diagramy, sbírka řešených příkladů*. Pardubice: Univerzita Pardubice, 2001.

[3] KNUTH, Donald E., *Umění programování: Základní algoritmy*. Přeložil David Krásenský. 1. vyd. Brno : Computer Press, 2008. 648 s. ISBN 80-251-2025-2.

[4] HÁLA, Tomáš. *Pascal: Pro střední školy*. Brno : Computer Press, 1999. 279 s. ISBN 80-7226-180-0

[5] ČSN ISO 5807. *Zpracování informací : dokumentační symboly a konvence pro vývojové diagramy toku dat, programu a systému, síťové diagramy programu a diagramy zdrojů systému* Praha : Český normalizační institut, 1995. 26 s.

[6] CORBIER, Daniel. *uCalc Fast Math Parser* [online]. c2009 [cit. 2009-08-11]. Dostupný z WWW: <http://ucalc.com/>.