ZÁPADOČESKÁ UNIVERZITA V PLZNI

FAKULTA PEDAGOGICKÁ

KATEDRA VÝPOČETNÍ A DIDAKTICKÉ TECHNIKY

REALIZACE JEDNODUCHÉHO BLOKOVĚ ORIENTOVANÉHO SIMULAČNÍHO PROGRAMU

BAKALÁŘSKÁ PRÁCE

Vojtěch Podzimek

Přírodovědná studia, obor VT

Vedoucí práce: Mgr. Jan Hodinář

Plzeň, 2014

Prohlašuji, že jsem diplomovou práci vypracoval samostatně s použitím uvedené literatury a zdrojů informací.

V Plzni, 5. dubna 2014

.....

vlastnoruční podpis

ZDE SE NACHÁZÍ ORIGINÁLNÍ ZADÁNÍ KVALIFIKAČNÍ PRÁCE

OBSAH

ÚVOD 2
1 SIMULACE 3
1.1 SYSTÉM 3
1.2 MODEL 3
1.3 MODELOVÁNÍ 4
1.4 SIMULACE 4
1.5 DRUHY SIMULACE 5
1.5.1 Simulace ve výcviku 5
1.5.2 Simulace ve výuce 5
1.5.3 Simulace v praxi 6
1.5.4 Simulace počítače 6
1.6 POČÍTAČOVÁ SIMULACE 7
1.6.1 Simulační programovací jazyk 8
1.6.2 Dělení počítačových simulací 8
2 VOLBA PROSTŘEDÍ A PROGRAMOVACÍHO JAZYKA 10
2.1 SIPRO 10
2.2 VOLBA PROGRAMOVACÍHO JAZYKA 10
2.3 VOLBA PROSTŘEDÍ 11
2.4 POUŽITÉ KNIHOVNY 12
2.4.1 Swing 12
2.4.2 JFreeChart 13
2.4.3 JExcelAPI 14
3 TVORBA APLIKACE 15
3.1 POSTUP TVORBY 15
3.1.1 Vzhled aplikace 16
3.1.2 Struktura aplikace 16
3.1.3 Grafika, obrázky 17
3.2 POPIS PROGRAMU 17
3.2.1 Struktura 18
3.2.2 Třídy 18
3.3 VÝPOČET 27
3.4 PŘIDÁNÍ BLOKU 29
4 UŽIVATELSKÁ DOKUMENTACE 31
4.1 SPUŠTĚNÍ 31
4.2 TVORBA SCHÉMATU 32
4.3 SPUŠTĚNÍ VÝPOČTU 34
5 ZÁVĚR 36
RESUMÉ 37
SEZNAM LITERATURY 38
SEZNAM OBRÁZKŮ, TABULEK, GRAFŮ A DIAGRAMŮ 39
PŘÍLOHY I
PŘÍLOHA A – KONSTRUKTOR BLOKU SUMA I
PŘÍLOHA B – METODA FUNKCE I
PŘÍLOHA C – PANEL S PARAMETRY JEN ČÁST TŘÍDY I
PŘÍLOHA D – REGISTRACE BLOKU VE TŘÍDĚ BLOKSETDIALOG II
PŘÍLOHA E – ZDROJOVÉ KÓDY, APLIKACE, JAVADOC (PŘILOŽENO NA CD) II

Úvod

V bakalářské práci se zabývám hlavně tvorbou aplikace, která má za úkol usnadnit výuku a přiblížit studentům danou problematiku. Toto téma jsem si vybral právě kvůli programování. Problematikou programování se zabývám celou dobu bakalářského studia, a proto si myslím, že je vhodné využít získané zkušenosti a vědomosti v bakalářské práci. Dalším faktorem výběru byla i skutečnost, že pokud se mi podaří vytvořit dobrou aplikaci, bude nadále používána a možná i vyvíjena.

Požadavky na aplikaci jsou následující. Aplikace má být blokově orientovaná a má plnit funkci programu SIPRO v rozsahu předmětu Řízení a simulace vyučovaném na Katedře výpočetní a didaktické techniky. Aplikace musí být spustitelná v prostředí MS Windows. Jelikož SIPRO je aplikace naprogramovaná pro prostředí DOS[1], neobsahuje dnes již přívětivé grafické rozhraní pro uživatele. Proto další požadavek na aplikaci je, aby byla co možná nejpřívětivější pro uživatele a umožňovala mu lehčí realizaci daného zapojení. Posledním požadavkem byla možnost exportu výstupních dat do programu MS Excel.

V první části bakalářské práce se zabývám simulací obecně. Snažím se přiblížit problematiku simulace a vysvětlit klíčové pojmy. V další části poukazuji na hlavní výhody simulace, na to, kde všude se používá a jaké velké výhody přináší. Poté se zaměřuji více na počítačovou simulaci, která se dnes hlavně využívá.

V poslední části práce se zaměřuji na aplikaci samotnou. Uvádím zde jednotlivé kroky vývoje a snažím se popsat klíčové třídy [2]a metody. Při popisu aplikace, vycházím z toho, že někdo bude chtít program modifikovat ať už přidáním bloku nebo novými funkcemi. Na konci je uvedena uživatelská dokumentace, která přiblíží konkrétní zacházení s aplikací a seznámí čtenáře se všemi funkcemi. Závěr je zaměřen na zhodnocení aplikace a cílů, které jsme měli zadané.

[1] Diskový operační systém s textovým uživatelským rozhraním
[2] Základní prvek objektově orientovaného programování, slouží jako předpis pro objekty

1 SIMULACE

Simulace je velmi obecné a rozšířené téma. Abychom byly schopni správně vystihnout podstatu simulace, je třeba si nejprve definovat pojmy systém, model a modelování. Tyto pojmy se v souvislosti se simulací vyskytují všude a je proto nutné vědět, co znamenají.

1.1 SYSTÉM

Slovo systém je používáno v hojné míře ve všech oborech. Přílišným používáním, hlavně v médiích, ztratil tento termín konkrétní význam a stal se prázdnou frází. Technické obory, které pracují se simulací, zvláště pak technické řízení, definovalo tento termín jako „objekt se vstupními a výstupními signály svázanými přes své vnitřní stavy pomocí obyčejných diferenciálních nebo diferenčních rovnic“ [1].

V případě simulace však nabývá tento termín jiného významu. V simulaci vždy studujeme nějakou abstraktní věc. Taková věc buď může opravdu existovat (člověk, firma, škola…), nebo o ní uvažujeme, že by existovat mohla (výrobní linka, pracovní proces, stroj…). Dále záleží, zda tato věc je závislá na čase. Pokud ano, jedná se o dynamický systém, pokud ne, používáme termín statický systém [1][2].

1.2 MODEL

Termín „model“ se dříve používal hlavně ve významu předlohy. V odborném jazyce zůstal z této praxe termín „funkční model“, a to ve smyslu prvního navrženého výrobku, který pracuje správně dle očekávání. Jedná se hlavně o správnou funkci výrobku, vzhled výrobku zde nehraje roli.

V simulaci termín model označuje analogii mezi dvěma systémy. Jedná se o vztah mezi systémy modulované a modulující. Je dán tím, že každému modulovanému systému P je přiřazen modulující systém Q. Tyto systémy mají své atributy, tzn. každému atributu g pro P je přiřazen atribut h pro Q a pro hodnoty g a h je dána nějaká relace, její charakter není obecně nijak omezen. Jedná se vlastně o vyjádření vztahu mezi reálným světem a modelem [1].

1.3 MODELOVÁNÍ

V češtině má slovo „modelování“ několik významů. Jeden z nich je dávat věcem nový tvar. Neexistuje však všeobecně přijatá definice, která by vyjadřovala správný význam slova. Avšak pokud jde o modelování ve smyslu výzkumné techniky, je tento význam vymezen přesněji.

„Podstatou modelování ve smyslu výzkumné techniky je náhrada zkoumaného systému jeho modelem (přesněji: systémem, který jej moduluje), jejímž cílem je získat pomocí pokusů s modelem informaci o původním zkoumaném systému [1].“

1.4 SIMULACE

Má několik významů, v obecné mluvě značí simulace předstírání nemoci, duševní poruchy apod. Tento význam spadá spíše do psychologie, nás bude simulace zajímat z hlediska technických oborů, kde má význam hlavně ve smyslu výzkumné techniky.

„Simulace je výzkumná technika, jejíž podstatou je náhrada zkoumaného dynamického systému simulátorem s tím, že simulátorem se experimentuje s cílem získat informace o původním zkoumaném dynamickém systému [1].“

Z toho lze vyvodit, že hlavním úkolem simulace je napodobení skutečného systému, stavu nebo procesu. Vymodelovat reálný svět, věc nebo hypotetické situace tak, aby bylo možné studovat jejich chování a vlastnosti v různých podmínkách. Simulace je často používaná tam, kde realizace skutečného problému je příliš drahá, nebo nebezpečná. Často jsou velmi flexibilní a lze je postupně rozšiřovat a doplňovat nebo naopak zjednodušovat či rozdělovat v závislosti na požadovaném výstupu z daného modelu.

Z toho je patrné, že simulace je užitečný nástroj v mnoha oborech. Její využití lze najít téměř všude. Od akademických účelů až ke zcela praktickým. Např. program CAD dokáže simulovat dopravní situace, kde na základě nastavených parametrů napodobuje silniční provoz. Tyto parametry se často získávají právě z reálného prostředí. Pomocí nich se pak snažíme nasimulovat co nejreálnější výstup. Mezi simulace řadíme i počítačové hry a videohry, které simulují určité příběhy, situace.

1.5 DRUHY SIMULACE

Simulace lze dělit do několika kategorii. Obecně se dělí podle oboru, do kterého spadají nebo dle jejich využití. Dělení existuje ale samozřejmě mnohem víc např. podle výstupu, charakteru, materiálu, modelu apod. Dále si uvedeme odvětví, kde se se simulací setkáme nejčastěji.

1.5.1 SIMULACE VE VÝCVIKU

Velké zastoupení simulací nalezneme právě ve výcviku. Tam mohou sloužit jednak k simulaci prostředí, které je příliš nebezpečné na výcvik, nebo k simulaci vybavení používaném v terénu, které je příliš drahé. Takovéto simulace mají za úkol připravit zaměstnance na krizové situace, které mohou nastat. Tento druh simulace využívají zejména vojenské a záchranné služby.

Pokud se zaměříme na zdravotnictví, zjistíme, že bez simulací by nebylo tak vyspělé a spolehlivé jako je nyní. Simulátory jsou zde používané k modelování rozhodovacích situací. Tyto simulace pak mohou poskytnout představu o důsledcích potencionálních rozhodnutí. Mohou pak lépe zvolit léčbu s menším rizikem důsledků. Neméně významnou úlohu má simulace při výcviku nových lékařů. Studenti se školí na různých simulátorech, ty jsou více či méně podobné reálnému subjektu. Kvalita získaných zkušeností je závislá na kvalitě simulátoru. Do praxe pak nastupují s jistými zkušenostmi, které získaly bezpečně na simulátoru [3][4].

Ale najdeme je i na jiných místech. Například pilotní simulátor je velmi využívaný pro simulaci letadla, jak pro piloty, tak i cestující. Existuje i nový typ počítačové simulace, která připravuje učitele na praxi. Jako velká výhoda takového simulátoru se uvádí snížení zklamání začínajících učitelů, kteří jsou často nespokojeni se svými výsledky, jelikož si myslí, že lze udržet žákovu pozornost nepřetržitě. První výzkumy naznačují, že studenti, kteří pracovali se simulátorem, hodnotí lépe své schopnosti učit. Takovýchto simulátorů existuje celá řada, v naprosté většině se jedná o počítačové simulace [5].

1.5.2 SIMULACE VE VÝUCE

Podobné jako simulace ve výcviku. Zde je kladen ale důraz spíše na pochopení dané problematiky a přiblížení chování v praxi. Simulace ve výuce tedy slouží pro vytvoření zjednodušeného reálného prostředí pro snazší pochopení problematiky a klíčových pojmů.

Takovéto simulace mohou být přímo tvořeny pro výklad látky, nebo mohou být formou hry. Např. obchodní simulace seznamuje studenty managementu s jednotlivými obchodními transakcemi bez jakéhokoliv rizika. Student si tímto osvojí dovednosti z reálného světa, aniž by mohl přijít k jakékoliv újmě. Simulace tedy je jednak pro studenty zábavnější formou výuky, ale hlavně jim přiblíží reálné fungování daného tématu. Simulace má význam i v samostudiu. Student není omezen osnovou, pokud projeví hlubší zájem o problematiku, může sám provádět simulaci a analyzovat výstupy.

Velmi důležitá je simulace ve fyzice a chemii, kde její význam neustále narůstá. Má dvě základní výhody, jednak student si ji může realizovat sám a tak se osobně lépe seznámí s daným problémem a také se dá lehce uchovat pro pozdější obnovu. Takovéto simulace jsou nejčastěji realizovány pomocí počítačového programu. Zvláště ve školách, kde realizace některých pokusů je náročná na vybavení, je dobré využití simulace [6].

1.5.3 SIMULACE V PRAXI

V praxi se simulace využívá zejména z finančních důvodů. A to k vytvoření modelu, který by se měl realizovat. Na takovémto modelu zkoumají jeho vlastnosti, a zda splňuje to, k čemu je zapotřebí. Ačkoliv jsou tyto modely relativně drahé, u větších projektu se tato investice vyplatí. Čím dál častěji se setkáváme se simulací podnikových procesů, za účelem zefektivnění. Jedná se spíše o větší firmy, které mají systém a chtějí ho vylepšit, nebo zavádějí nový [2].

Podle Buriety[7] je vždy důležité před takovýmito simulacemi provést jisté kroky. Správně a úplně definovat cíl simulace. Jako je například zvýšení efektivity o 20%, nebo snížení potřebného času pro výrobu jednoho kusu o 15% atp. Další důležitý krok je, určit podrobnosti projektu. „Ve všeobecnosti platí pravidlo: modelovat s minimálním počtem elementů nutných k dosáhnutí požadovaného cíle. [7]“ A nakonec je dobré shromáždit údaje potřebné na samotnou simulaci.

1.5.4 SIMULACE POČÍTAČE

Zvláštním typem simulace je simulace počítače nebo počítačového programu. Tento druh simulátoru někdy nese označení emulátor. Mezi simulací a emulátorem ovšem existuje jistý rozdíl. Tato problematika je dobře rozebrána v Peterkovi[8]. Zde uvádí, že cílem simulace je získat informace o systému, které z nejrůznějších důvodů nemůžeme

získat přímo. Kdežto emulace slouží k simulaci něčeho, o čem víme vše, co potřebujeme, ale nemáme jej k dispozici. „Rozdíl mezi simulací a emulací je tedy především v tom, k čemu slouží – simulace k získání nových poznatků o určitém systému, zatímco emulace umožňuje zajištění jeho funkcí jinými prostředky [8].“

S emulátorem se můžeme setkat velmi často. Například při návrhu webových stránek se používá emulátor, který simuluje všechny webové prohlížeče. Takže pro optimalizaci nepotřebuji mít všechny prohlížeče nainstalované, optimalizaci pak provedu pomocí emulátoru. Lze simulovat i celý operační systém např. populární DOSBOX pro simulaci DOSu je stále hojně používaný.

1.6 POČÍTAČOVÁ SIMULACE

Simulace je velmi obecné téma, které zasahuje téměř všude. V minulé kapitole jsem nastínil, kde všude se můžeme se simulací setkat a jaké obrovské výhody přináší do vybraných oborů. Simulace může mít mnoho podob od různých figurín až po uměle vytvořená místa. Dále budeme hovořit jen o simulaci počítačové.

Dříve, když nebyly počítače, se simulace prováděla na speciálních zařízeních. Dnes se provádí převážně na počítači, tento druh simulace se také označuje jako „číslicová“. V počítačové simulaci je modelem počítačový program. Tento program má za úkol simulovat určitý systém z reálného světa a určit jeho výstupní hodnoty, které se budou co nejvíce blížit reálnému systému. Tyto hodnoty ovlivňují vstupní parametry. Na základě změny parametru je uživatel schopen ovlivňovat výstup dat a hledat tak optimální řešení daného problému [1].

Často jsou simulační programy značně složité, jelikož musejí být analogií nějakého složitého děje. Podle Křivého a Kindlera[1] pro tvorbu programu existuje několik způsobů, které se používají. První způsob je sestavit program z jednotlivých částí. Každá takováto část je tvořena podprogramem, který provádí více či méně náročné výpočty. Z těchto bloků se poté sestaví výsledný program. Další způsob je využít univerzálního programu. To je program, který je řízen vstupními daty. Na základě nich se pak určuje cesta výpočtu, kterou se program vydá. Dále se využívá programovacích jazyků, které popisují požadavky na výpočet simulace. Jsou buď interpretovány programem, nebo jsou překládány do zdrojového kódu. V neposlední řadě jsou používány objektově orientované programovací

prostředky, jakožto základ pro definice adekvátních problémově orientovaných programovacích prostředků.

1.6.1 SIMULAČNÍ PROGRAMOVACÍ JAZYK

Jedná se o speciální programovací jazyky, které jsou realizovány proto, aby usnadnily psaní simulačních programů. Jde o určité pokyny, které řídí práci počítače, zároveň by měli být srozumitelné i pro ne-programátora. Takovéto jazyky se využívají k matematickému modelování reálných událostí. Jazyky tohoto typu se snaží být pro daný obor co nejsrozumitelnější.

„Stručně řečeno, přínos simulačních jazyků spočívá v prvé řadě v tom, že chce-li uživatel takového jazyka sestavit simulační program, jehož běh by na počítači realizoval simulační model jistého systému, popíše tento systém v simulačním jazyce, a to podobně, jako by ho popisoval svému kolegovi v profesi (snad jen někde se bude vyjadřovat více „po lopatě“), a popis je pak už transformován do příslušného strojového programu automaticky, v dnešní době téměř výhradně kompilací [1].“

Protože simulačních programů existuje celá řada, existuje i více způsobu dělení. My si zde jazyky rozdělíme dle systému, který popisují. Systémy mohou být tří druhů, pokud v daném systému existuje transakce, jedná se o systém T (transakce). Jestliže systém má jen atributy, transakce v něm tedy neprobíhají, jedná se o systém typu A (atribut). Systém AT (atribut-transakce) je systém, který má jak atributy, tak i v něm probíhají transakce. Dle těchto systémů pak nesou označení i simulační jazyky, tedy A, T, AT. Do jazyku typu T patří také objektově orientované programování, pro které jsem se rozhodl ve své práci. Obecnější dělení je pak úzce spjato s dělením počítačových simulací, které si popíšeme v další kapitole [1].

1.6.2 DĚLENÍ POČÍTAČOVÝCH SIMULACÍ

Počítačovou simulaci lze rozdělit dle charakteru simulovaného systému. Jestliže je spojitý, hovoříme o spojité simulaci. Pokud je simulovaný systém diskrétní, hovoříme o diskrétní simulaci. Systém může být ještě kombinovaný, to znamená, má vlastnosti, jak pro spojitou, tak nespojitou simulaci. Takováto simulace pak nese označení kombinovaná diskrétně-spojitá [2][9][1].

Dalším používaným dělením je podle předvídatelnosti událostí. Pokud simulace nemění vstupní data, ale mění výstup, tak se jedná o stochastickou simulaci. Součástí výpočtu je pravděpodobnost nebo nějaký náhodný prvek. Na základě nich pak můžeme získávat jiná výstupní data, aniž bychom jakýmkoliv způsobem ovlivnily vstup. Pokud výsledek závisí pouze na vstupních datech, znamená to, že pokud nezměníme vstup, výstup je pořád stejný, tak se jedná o deterministickou simulaci [1].

Další dělení můžeme vyvodit z výstupu samotné aplikace. Jedná se o dělení na statické a dynamické výstupy. Pokud hovoříme o statickém výstupu, myslí se tím např. nějaký graf, který se vykreslí a je zobrazen jako obrázek, který už se nemění. Naproti tomu dynamické systémy se vykreslují postupně v čase, jedná se o animaci.

Uvedu zde ještě jedno rozdělení, které je důležité pro složitější výpočty. A tím je právě způsob výpočtu, který může být zpracováván buďto na jednom počítači (místní zpracování), nebo na několika propojených počítačích (distribuované zpracování).

2 VOLBA PROSTŘEDÍ A PROGRAMOVACÍHO JAZYKA

Jak již bylo zmíněno v úvodu, mým úkolem bylo vytvořit blokově orientovaný program, který bude plnit funkci simulačního programu SIPRO, v rozsahu předmětu Řízení a simulace. Na základě požadavků na aplikaci jsem se rozhodoval, který programovací jazyk zvolit. Podle programovacího jazyka jsem pak vybíral programovací prostředí. Jaký jazyk jsem vybral a proč je rozebráno v této kapitole.

2.1 SIPRO

SIPRO je rychlý blokově orientovaný simulační program pro počítače třídy IBM PC/XT-AT. Byl vyvinut na Katedře automatizační techniky a řízení, Fakulty strojní, VŠB – Technické univerzity v Ostravě. Používá se především pro laboratorní a výukové potřeby. Nabízí tvorbu simulačního modelu pomocí množiny předdefinovaných typů bloků, realizujících základní přenosy a to spojité i diskrétní. Program je výkonnou pomůckou pro řešení simulačních úloh malého a středního rozsahu, jmenovitě do 255 bloků.

2.2 VOLBA PROGRAMOVACÍHO JAZYKA

Zvolit programovací jazyk pro mě nebylo těžké. Vycházel jsem z toho, že cílem je vytvořit přívětivé prostředí pro uživatele. Proto jsem musel vybrat jazyk, ve kterém je možná co nejlepší podpora pro tvorbu grafického uživatelského rozhraní (GUI). Jak jsem zmínil v předchozí kapitole, další důležitý požadavek je podpora objektového programování. Doposud jsem se setkal jen se třemi jazyky, které toto splňují, Object Pascal[3], C++ [4]a Java[5]. Jelikož nejvíce používám Javu, byla pro mě jednoduchou volbou. V následujícím odstavci se pokusím shrnout některé její vlastnosti, díky kterým jsem ji zvolil.

Nejvíce vyzdvihovaná vlastnost Javy je dána její univerzálností. Zejména pak nezávislosti co se platformy týče. To je dáno tím, že místo skutečného strojového kódu se vytváří tzv. „bajtkód“. Na daném zařízení pak stačí mít nainstalovaný interpret Javy tzv. Java Virtual Machine, některé mikroprocesory umožňují spustit Javu hardwarově. Právě díky těmto vlastnostem je Java rozšířena téměř do všech zařízení [10][11][12].

[3] Rozšíření programovacího jazyka Pascal o objektově orientované programování
[4] Programovací jazyk, který podporuje objektově orientované programování
[5] Objektově orientovaný programovací jazyk

Další důležitá vlastnost vyplývá z toho, že Java je open source[6]. Má širokou uživatelskou základnu, která neustále jazyk rozvíjí. Java sama osobě je velmi jednoduchý jazyk, který je dodáván s velikým počtem knihoven (knihovny jsou často nazývané jako „API"), se kterými lze pracovat. Právě knihovny dělají z Javy tak mocný nástroj na programování. Pomocí nich lze tvořit velice efektivně a pohodlně. Naučit se Javu jako jazyk samotný je jednoduché, skutečnou výzvou je naučit se využívat knihovny a funkce v nich dostupné. Je třeba také zmínit, že Java je silně objektová, byla již s tímto záměrem vytvořena, a zcela mu podléhá [12].

Jako všechno, tak i Java má také negativní vlastnosti. Ta, se kterou se nejčastěji setkáváme, je její pomalejší spuštění. To je dáno tím, že program se musí nejprve přeložit a pak jej teprve spustit. Jsou různé mechanismy, které se snaží toto zpoždění eliminovat, nicméně určité zpomalení je tu stále [10].

2.3 VOLBA PROSTŘEDÍ

Volba správného prostředí je důležitá část vývoje. Zvláště v případě rozsáhlého programu. Pokud programátor dodržuje jisté zásady a doporučení, pomocí správného prostředí lze pak provádět úpravy a modifikace programu velice efektivně. Jelikož jsem si vybral Javu jako programovací jazyk, rozhodoval jsem se mezi Eclipse a NetBeans. Tyto prostředí patří mezi nejpoužívanější a jsou na srovnatelné úrovni. Já jsem se rozhodl pro NetBeans, protože bez nutnosti instalace dalších doplňků nabízí velmi povedené prostředí pro tvorbu GUI. V něm se nechá vytvořit návrh, ke kterému se automaticky vygeneruje kód. To je veliké zjednodušení pro vývoj naší aplikace, nemusíme se zabývat složitým umisťováním komponent.

NetBeans je primárně vytvořen pro vývoj aplikací v jazyce Java, podporuje ale jiné programovací jazyky, jako je např. C/C++, PHP[7], Groovy [8]nebo HTML5[9]. Nabízí mnoho různých doplňků, které rozšiřují možnosti programování. Je to open source projekt s rozsáhlou komunitou vývojářů. NetBeans je naprogramován v Javě, což mu umožňuje rozšíření do různých platforem. Proto jej lze spustit na operačních systémech Windows,

[6] Počítačový software s otevřeným zdrojovým kódem
[7] Skriptovací programovací jazyk
[8] Objektově orientovaný jazyk pro platformu Java
[9] Nová verze značkovacího jazyka HTML

Linux, Mac OS nebo Solaris[10]. Na jeho popularitě má velký podíl jeho uživatelské rozhraní, které je přehledné efektivní a intuitivní.

2.4 POUŽITÉ KNIHOVNY

Jak jsem zmínil už dříve, knihovny jsou důležité části celého konceptu programovacího jazyka Java. Ty nejužitečnější a nejpoužívanější knihovny jsou přímo součástí Javy. Existuje ovšem plno dalších knihoven, které jsou volně ke stažení. Najdeme i mnoho placených knihoven, které jsou určené k prodeji. Knihovnu si ovšem může každý vytvořit sám podle své představy, o jejím publikování pak rozhodne sám. V následujícím textu rozepíši, jaké knihovny jsem použil a proč.

2.4.1 SWING

Jedna z nejdůležitějších knihoven pro vytvoření přívětivého rozhraní pro uživatele je Swing. Právě pomocí Swingu máme možnost vytvářet grafické prvky a přidávat jim vlastnosti a funkce. Dají se jím vytvořit jak klasické formulářové aplikace, tak i složité animace. Pomocí Swingu můžeme vytvářet různé okna, rámečky, tlačítka, labely, seznamy, tabulky… Pro specializované požadavky je výhodné použít knihovny připravené právě na konkrétní problém. Dopomůže nám jednoduchými příkazy získat výsledek, který bychom jinak zdlouhavě programovali, a zvyšuje přehlednost kódu.

Swing vychází z knihovny AWT[11]. Po objevení některých základních chyb (např. závislost na platformě) v AWT byl vývoj AWT zastaven. AWT i Swing využívají instance třídy *Component*, každý takovýto objekt reprezentuje nějaký grafický prvek, který bude zobrazen na monitoru. Od *Component* je odvozen *Container*, který umožňuje uvnitř sebe sama vykreslovat další objekty. Každý *Container* má k dispozici právě jednu instanci speciální třídy, která je označena jako správce rozvržení. Správce rozvržení obsahuje logiku pro rozmisťování komponent[12]. Dále také ovlivňuje i jejich ukotvení a způsob změny velikosti v závislosti na změně velikosti nadřazeného *Containeru*. Právě volba správného správce nám umožní umístit jednotlivé komponenty podle potřeby [12][11].

[10] Unixový operační systém
[11] Nástroj pro tvorbu grafického uživatelského rozhraní
[12] Jednotlivé grafické prvky

Základním prvkem ve Swingu je třída *JComponent*. Ta je potomkem třídy *Container*, to znamená, že každý potomek třídy *JComponent* v sobě může mít další komponenty. Každá komponenta, vytvořena ve swingu, je potomkem třídy *JComponent* a před svým názvem má písmeno „J“ *JButton*, *JLabel* apod. Tím se rozlišuje od komponent AWT [12][11].

Swing má nastaven defaultní vzhled svých komponent, který byl určen vývojáři Swingu. Tento vzhled se označuje „Metal“ a je si velmi podobný na všech platformách. Swing na rozdíl od AWT umožňuje nastavit svůj vlastní vzhled, nemusí tedy přejímat vzhled od operačního systému. Uživatel může použít libovolný vzhled, který si naprogramuje [12].

Jednou z nejdůležitějších vlastností Swingu je obsluha událostí. Události jsou signály přijímané od operačního systému, většinou na základě akcí od uživatele. Pomocí událostí program komunikuje s uživatelem a určuje, co se bude dít. Nejčastěji se jedná o pohyb myší, stisknutí dané klávesy, změnu rozměru atp. Odchycení události se realizuje tak, že se na danou komponentu zavěsí posluchač, v Javě označen jako *eventListener*. Každý typ posluchače může naslouchat definovanému typu události. Na jiné, než ty definované, nereaguje. Posluchač pak vyvolá určenou metodu, kterou jsme naimplementovali.

2.4.2 JFreeChart

Knihovna JFreeChart je univerzální nástroj, který slouží k výrobě grafu. Veliká výhoda této knihovny oproti konkurenci je, že je open source. Přesto je k dostání stále velké množství podobných knihoven. Já jsem zvolil JFreeChart, z důvodu její jednoduchosti a relativně dobré podpory z řad uživatelů. Tvůrci dávají k dispozici dobře popsaný javadoc[13], avšak příručka pro vývojáře je pouze ke koupi.

JFreeChart umí vytvářet jednak základní grafy, jako jsou bodový, spojnicový, kruhový atp., také je schopen vykreslovat například i Ganttův [14]graf nebo různé typy burzovního grafu. Dále nabízí rozšířené uživatelské nastavení, které si může uživatel lehce zobrazit pravým kliknutím na graf. Zde může modifikovat vzhled, různé rozměry, osy nebo přímo tisknout. Zvládne například i vybarvování plochy pod křivkou, mezi křivkami nebo nad křivkou, tato vlastnost pak viditelně zpřehlední složitější grafy [13].

[13] Automaticky generovaná dokumentace k programům založených na Javě
[14] Druh pruhového diagramu

Nevýhoda JFreeChar spočívá v jeho neschopnosti zobrazovat dynamické procesy v reálném čase. To se dá obejít rychlým překreslováním celého grafu, avšak toto řešení je neúnosné z hlediska výpočetního výkonu. Tímto způsobem se zvýší výpočetní výkon celé aplikace, což je nežádoucí. JFreeChart dále neumí zobrazovat 3D grafy, knihovna vynáší body pouze do dvou směrů. Pro naši aplikaci však žádný takovýto úkon nepotřebujeme, proto volbou knihovny JFreeChart nic nezkazíme.

Princip programování v JFreeChart je stejný jako u ostatních takových knihoven. Skládá se z pěti kroků, které jsou standartní, a není v nich žádná záludnost. V prvním kroku je hlavním úkolem vytvořit datový model. Každý graf má jiný datový model, proto je nutné vybrat jej pro správný graf. V dalším kroku naplníme model skutečnými daty, které chceme zobrazit v grafu. Vstupní data se zase odvíjejí od vlastností daného grafu. Když máme založený model naplněný daty, můžeme vytvořit samotný graf. Zatím jej máme vytvořený jako instanci třídy, není zatím zobrazený. Nyní můžeme provádět úpravy grafu, měnit zobrazení bodů, čar, barev… Vlastnosti, které lze měnit, se odvíjejí opět od typu daného grafu. Nyní zbývá jen zobrazit samotný graf [13].

2.4.3 JExcelAPI

Jelikož byl požadavek na možnost exportu výstupních dat do MS Excel, tak bylo zřejmé, že budeme potřebovat nějakou knihovnu pro tuto funkci. Právě pro svoji jednoduchost a srozumitelnost mě zaujala knihovna JExcelAPI. Tato knihovna umí vytvářet a číst XLS soubory, umí formátovat jednotlivé buňky a také vkládat PNG obrázky do souboru. To jsou právě funkce, které budeme potřebovat pro naši aplikaci, JExcelAPI je tedy pro naše účely ta pravá. Tato knihovna je stejně jako JFreeChart open-source [14].

Zde jsem uvedl jen tři hlavní knihovny, které jsem použil pro vytvoření aplikace. Pokud bychom se podívali do kódu, zjistili bychom, že knihoven je používáno více. Jedná se o knihovny, které jsou přímo součástí Javy. Například knihovna Math je zde využívaná pro počítání s přesnými čísli, nebo třeba knihovna IO, pro ukládání a načítání dat.

3 TVORBA APLIKACE

V této kapitole ukazuji, jak jsem postupoval při tvorbě programu. Snažím se přiblížit jednotlivé kroky a vysvětlit jejich podstatu. Podrobněji je zde rozebrána struktura aplikace, jelikož je při návrhu velmi důležitá. Poté popisuji aplikaci konkrétně, vysvětluji zde základní funkce tříd, a jakou mají úlohu v aplikaci. V další části rozebírám hlavně výpočet, vysvětluji jeho princip a nedostatky. Na konci této kapitoly ukazuji postup přidání jednoduchého bloku do aplikace.

3.1 POSTUP TVORBY

Před samotným psaním programu je třeba si ujasnit jisté konvence, které je dobré pak všude dodržovat. Co se týče jazyka, kterým budeme popisovat jednotlivé metody a atributy, zvolil jsem češtinu. Jelikož celá práce i komentáře budou psané v češtině, myslím, že je to správná volba. Pro standardní metody budeme ovšem zachovávat popis v angličtině, jako jsou getry[15], setry[16], add atp.

Dalším důležitým aspektem je pravidlo pojmenování metod, tříd, balíků[17], atributů. Použil jsem zde konvenci „velblouda“, což znamená, že neodděluji slova v popisku podtržítkem, ale velkým písmeny. To je v Javě běžně užívané pravidlo, které se vyplatí dodržovat. S tím souvisí i to, že třídy začínají velkými písmeny, vše ostatní pak malými. Pokud nebude jiná potřeba, atributy v třídě mají status *private,* metody mají status *public*. Vycházíme z toho, že k atributům přistupujeme přes metody.

Co se týče formátování kódu, budeme dodržovat následující pravidla. Na jednom řádku by měl být jeden příkaz, deklarace jedné proměnné. Otevírací závorka bloku je na konci řádku, uzavírací na samotném řádku. V řídících strukturách budeme vždy používat složené závorky a to i v případě, že je v bloku pouze jedna funkce.

Komentáře kódu by měli být u každé metody a třídy. U třídy pak stručný popis, jméno autora popř. verze programu. Problémové pasáže kódu budou doplněny o detailnější komentář. Budeme předpokládat, že čtenář zdrojového kódu bude znát Javu a bude schopen se v kódu orientovat.

[15] Metoda, která slouží pro získání atributu z objektu třídy
[16] Metoda, která nastavuje vlastnosti objektu třídy
[17] Způsob jak oddělit prostor v Javě, podobné adresářové struktuře

3.1.1 VZHLED APLIKACE

Jako první krok jsem zvolil návrh aplikace. Sepsal jsem jednotlivé funkce, které aplikace musí splňovat. Rozvrhl jsem jednotlivé prvky a určil rozměry aplikace. Oproti programu SIPRO, máme nevýhodu kvůli grafické prezentaci jednotlivých bloků. Jsme tedy omezeni velikostí plátna, na kterém budeme bloky zobrazovat. Navrhl jsem velikost, která je přijatelná na všech dnešních monitorech s tím, že pokud by byl problém s rozmístěním bloků, udělalo by se plátno rolovací nebo oddalovací.

Jednotlivé rozšiřující nabídky a nastavení jsou vytvářené v průběhu vývoje aplikace, dle toho, jak je potřeba. Práce s aplikací by měla být intuitivní a jednoduchá. Pro komunikaci s ní uživatel používá dobře známé a zaběhlé pokyny, které se vyskytují ve většině počítačových aplikací.

Design aplikace je stylizován do moderního stylu Windows 8, na této platformě jsem jej i vyvíjel. Což nemá žádný negativní vliv na starší platformy. Barvy jsou zvolené v kombinaci bílé, světle modré a rudé červené. Během vývoje aplikace se několik věcí změnilo, musel jsem tedy původní návrh lehce přizpůsobit podle potřeby funkčnosti.

3.1.2 STRUKTURA APLIKACE

Dalším důležitým aspektem při tvorbě aplikace je volba struktury. Ta se dá relativně dobře určit, když víme, co máme programovat. Poté se k této představě snažíme přiblížit, nikoliv však na úkor funkčnosti. Už při návrhu jsem bral ohled na to, že někdo možná bude chtít v budoucnu aplikaci rozšířit. Proto jsem chtěl oddělit zobrazovací část od řídící. Snažil jsem se co nejvíce přiblížit modelu MVC.

MVC

MVC (Model-View-Controler) je architektura aplikace, která má za úkol rozdělit program do tří vrstev, které by na sobě měly mít co nejmenší závislost (v ideálním případě žádnou). Jedná se o oddělení uživatelského rozhraní od modelu a controleru (do češtiny často překládáno jako řadič). Model se zde myslí určité seskupení informací. Dobrým příkladem je zde databáze, nebo nějaké konfigurační soubory apod. Controler je zde ve smyslu jednotky, která provádí operace právě s modelem na základě uživatelských událostí. Pokud se dodrží filozofie této architektury, máme pak možnost jednak lehké obměny jednotlivých vrstev (samozřejmě musíme respektovat jednotlivé operace mezi vrstvami).

Ale také nám umožňuje vytvářet aplikace nezávisle na sobě. Domluví se operace, které budou jednotlivé vrstvy umět, a dále může každou vrstvu tvořit někdo jiný nezávisle na sobě. To je ohromná výhoda při tvorbě rozsáhlých programů, které se jinak realizují o poznání hůře. Další výhoda je bezpečnost aplikace, jednotlivé vrstvy jsou zapouzdřené a přístup k nim se provádí jen přes operace. Tato architektura se hojně využívá při vývoji webových stránek, které jsou pak nezávislé na vzhledu (šabloně) [15].

Právě z těchto důvodu a kvůli přehlednosti jsem rozvrhl aplikaci do třech hlavních balíků – *kontrolér*, *zobraz*, *bloky*. Kontrolér by měl reprezentovat vrstvu řadiče. Tvořit jakési rozhraní mezi zobrazovací vrstvou a bloky, výpočty, nastavením. Do jaké míry se mi povedlo zachovat architekturu MVC, zhodnotím při popisu programu.

V návrhu jsem zvážil i použití jednotlivých tříd. Už z logiky věci se nabízí využít dědičnost alespoň u bloků. Jelikož všechny bloky mají většinu vlastností stejnou, určím rodičovskou třídu, z které naprostou většinu vlastností zdědím. Budou se lišit jen ve funkci, počtu vstupů, vstupních parametrech, názvu a vzhledu. Zde využijeme velmi mocného nástroje objektového programovaní, kterým je polymorfismus. Pomocí této vlastnosti pak můžeme použít tutéž metodu v kontextu různých tříd. Dědičnost jednotlivých grafických prvků zde budeme zanedbávat, jelikož to není předmětem návrhu.

3.1.3 GRAFIKA, OBRÁZKY

Poslední část návrhu je příprava jednotlivých grafických prvků a obrázků. Jednotlivé bloky jsou vytvářené vektorovou grafikou v programu Inkscape. Tyto bloky jsou pak převedeny do formátu PNG. Velikost těchto obrázků jsem odvodil od velikosti plátna, kde budou umisťovány. Ikony jednotlivých tlačítek jsou staženy z internetu pod licencí GNU[18].

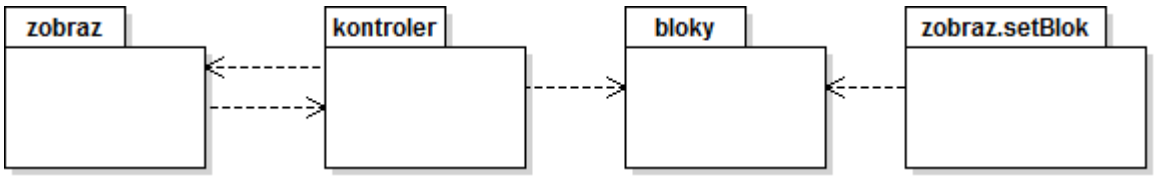
3.2 POPIS PROGRAMU

Při tvorbě programu jsem narazil na několik problému, které mi nedovolil dodržet architekturu, tak jak jsem si ji navrhl. Několikrát jsem byl nucen přehodnotit vlastnosti tříd, abych neomezil funkčnost výsledné aplikace. Nyní si ukážeme, jak je aplikace ve skutečnosti naprogramovaná.

[18] Licence, která umožňuje objekt používat, modifikovat i šířit pro nekomerční účely

3.2.1 STRUKTURA

Jak jsem již uváděl v předchozí kapitole, snažil jsem se program zasadit do architektury MVC. Celá aplikace je rozdělena do několika balíků. Základní zobrazovací balík *zobraz* obsahuje základní zobrazované prvky aplikace. Z něho vychází balík *zobraz.setBlok*, který obsahuje okna s nastavením jednotlivých bloků. Dalším balíkem je *kontrolér*. Zde se provádí jednotlivé výpočty a operace s bloky, nastavuje se prostředí a provádí simulace. Pomocí těchto tříd pak reagujeme na uživatelské příkazy. Posledním balíkem je *bloky*, který obsahuje jednotlivé bloky s jejich funkcemi a vlastnostmi. Kromě nich však obsahuje také třídy komponent pro návrhy schémat, jako je *uzel*, *konektor*, *koncovka* a *hodnota*. UML diagram balíků můžeme vidět na obrázku 1.



Obrázek 1 - Struktura balíků

Na obrázku vidíme, že architektura nebyla zcela dodržena, jelikož neobsahuje vazbu balíku *bloky* na *zobraz*. Nicméně tento nedostatek jsme ochotni tolerovat vzhledem k jednoduchosti aplikace. Náš stav tedy vychází z následující úvahy. Uživatel zadá příkaz, kontrolér jej zpracuje (dle povahy příkazu buď využije ostatní balíky či nikoliv) a vrátí bloku *zobraz* změny, které on nastavil. Důležitá vlastnost této architektury je, že zobrazovací balík (*zobraz*) nekomunikuje přímo s modelem (*bloky*), takže jej zajímá jen balík *kontrolér*, se kterým komunikuje. To by mělo vést ke zjednodušení budoucího rozšíření.

3.2.2 TŘÍDY

Nyní si popíšeme třidy jednotlivých balíků. Zaměříme se zejména na jejich klíčové funkce a vztahy s ostatními třídami. Spouštěcí třída je zde *ProgSim*, která je umístěna v balíku *appProgSim*. Má za úkol vytvořit základní instance tříd, čímž nastartuje samotnou aplikaci. Všechny metody a UML diagramy jsou popsány v javadocu, který je přiložen k práci, proto dále vystihneme jen základní princip fungování a popíšeme si zásadní metody a atributy.

Balík bloky

Třídy v tomto balíku vycházejí z jedné rodičovské třídy *Blok*. Ostatní bloky jsou potomky této třídy. Dále zde najdeme třídy *Koncovka*, *Uzel*, *Konektor* a *Hodnota*, které

nemají žádný vztah s ostatními třídami. Koncovky zde slouží jako jakési zásuvky do daného bloku. Každý blok jich má tolik, kolik má vstupů a výstupů, do nich se pak připojují jednotlivé konektory. Konektor zde má funkci spojovat bloky do sebe, skrze zásuvky. Nakonec zde máme uzel. Uzel se umisťuje na konektor a umožňuje připojení dalšího konektoru. Díky uzlům pak můžeme libovolně napojovat jednotlivé bloky. Třídu *Hodnota* zde uvádět nebudu, jelikož je primitivní, má za úkol ukládat hodnotu s časem.

Blok

Jak jsem již zmínil, jedná se o hlavní třídu v tomto balíku. Tato třída je značně složitá, jelikož obsahuje potřebné metody pro pohyb s bloky po plátně. Nyní si popíšeme nejdůležitější atributy pro pochopení principu fungování.

Zásadní vlastnost, kterou musí každý blok mít, je znát komponenty, které jsou k němu připojené. To by šlo obejít ukládáním těchto informací do souboru nebo speciální třídy. Nicméně pro naši aplikaci se snažíme využívat možnosti objektově orientovaného programování, proto k uložení těchto informací v bloku jsme využili kolekce, které nám umožní ukládat celé objekty a libovolně pak s nimi zacházet. Máme tedy čtyři kolekce, pro vstupní bloky, konektory, koncovky a hodnoty. Pomocí koncovek jsme pak schopni detekovat jednotlivá připojení a určit zda jsou připojena či nikoliv. Pomocí konektoru pak zjistíme vstupní bloky. Pro zjištění vstupních bloku zde máme metodu *getVstup*, kde jako parametr zadáváme číslo vstupu od 0.

Další důležitý atribut bloku je, zda blok pracuje, či nikoliv. Toho využíváme při výpočtu simulace zvláště pak, pokud jsou bloky zacyklené. Atribut *nainitován*, nám určuje, zda daný blok prošel už všemi časovými intervaly. Pokud ano, přeskočíme jeho další výpočty a žádáme jen hodnotu pro daný čas. Toto je výhodné pro zjednodušení časové náročnosti programu, nebudeme počítat to, co je již vypočítáno. Další použité atributy slouží především k určování a nastavování polohy bloku, často jsou to objekty typu *Point*.

Třída využívá klasických funkcí objektového programování, jakými jsou getry a setry. Pomocí nich pak z objektů získáváme jednotlivé vlastnosti. V konstruktoru bloku inicializujeme jeho výchozí polohu, připravujeme kolekce pro komponenty a přidáváme události, na které chceme reagovat. Důležitá je zde metoda *nastavBlok*. Na základě

nastavení zde určujeme, zda blok bude reagovat na události nebo nebude, a posunujeme koncovky na základě pohybu, který vykonal blok.

Zcela klíčová metoda ve třídě *Blok* je ***funkce***, která nám reprezentuje právě funkci daného bloku. Jelikož každý blok má jinou funkci, tak tato metoda je určena k překrytí. Každá třída, která je potomkem této třídy, si překryje metodu dle své potřeby. Tím u každé třídy můžeme nadefinovat její chování. Na podobném principu zde fungují i metody *priprav* a *vynuluj*. Pomocí těchto metod máme zaručeno, že při každé nové simulaci budeme vycházet ze stejných hodnot.

Důležité je detekovat zpětnou vazbu obvodu, k tomu nám dopomáhá metoda *navazSpojení*. Její úkol je prostý, zeptá se bloku, jenž je připojen na vstup daného bloku, zda právě pracuje. Pokud odpoví, že ano, detekovali jsme zpětnou vazbu. Průběh výpočtu budeme detailněji rozebírat dále.

Pohyb jednotlivých bloků je realizován právě v této třídě a to metodou *addMoveEvent*. Protože potřebujeme tento pohyb zapínat a vypínat, je zde i metoda *removeMoveEvent*. Metoda využívá třídu *MouseAdapter*, u této třídy překryjeme metody *mouseDraged* a *mouseMoved*. Musíme rozlišovat několik úrovní, jednak umístění našeho plátna vůči obrazovce a pak polohu myši na obrazovce. Odečtením těchto dvou hodnot dostaneme souřadnice na plátně. Jednoduchou podmínkou pak určíme, kdy se myš pohybuje na plátně a kdy nikoliv. Nyní zbývá už jen nastavit bloku nové souřadnice umístění.

Tímto jsme vystihli nejdůležitější metody a atributy třídy *Blok*, které jsou pro správný chod nezbytné. Co se týče pak jednotlivých bloků, musíme u nich definovat vlastnosti, které má každý blok specifické. Jedná se o nastavení počtu vstupů a výstupů, ikony a popis. Tyto parametry se udávají již v konstruktoru. Dále musíme nadefinovat metodu *funkce*, která nám určuje chování celého bloku. Pokud potřebujeme, překryjeme metody *vynuluj* a *nastav*, není to však nutné.

Koncovka

Funkci koncovky jsem popisoval výše, zde si uvedeme klíčové atributy a metody této třídy. Základní atributy *vstup* a *výstup* jsou typu boolean[19], slouží k určení, zda se jedná o

[19] Logický datový typ, který má pouze dvě hodnoty TRUE a FALSE

vstupní či výstupní koncovku. To je důležité pro kontrolu správnosti návrhu, jelikož nemůžeme připojovat vstup na vstup, nebo výstup na výstup. Nebo také výstupní koncovka být připojena nemusí, kdežto koncovka vstupní být připojena musí. Právě z tohoto důvodu je zde další parametr *pripojen*. Neméně důležitý je tu atribut *blok*, který značí blok ke kterému je koncovka připojena.

Další metoda, se kterou je třeba se seznámit je *addMoveEvent*. Ta nám slouží právě k vytváření konektorů mezi jednotlivými koncovkami. Zjednodušeně to funguje tak, že když stiskneme levé tlačítko myši na koncovce, uloží se nám koncovka jako vstupní. Jestliže tlačítko nepustíme a najedeme na jinou koncovku, uloží se nám tato koncovka jako výstupní, tím vytvoříme konektor. Během tohoto procesu nesmíme tlačítko uvolnit nikde jinde mimo koncovku, jinak se bloky neuloží a konektor nevznikne.

Uzel

Uzel simuluje výstupní koncovku, která se nachází na konektoru místo na bloku. Představme si situaci, kde máme dva bloky blok A a blok B. Blok A má pro jednoduchost jeden výstup a blok B jeden vstup. Propojíme je konektorem, na který upevníme uzel. Přidáme blok C také s jedním vstupem, místo toho abychom vedli konektor na výstup bloku A, vyvedeme ho na uzel. Výsledek je stejný a navíc nám umožňuje lepší manipulaci se schématem. Tato třída má také metodu *addMoveEvent*, s tím rozdílem, že zde řešíme i přemisťování uzlu, je to vytvořeno na stejném principu jako u bloků. Kromě této metody už nenabízí tato třída nic zajímavého.

Konektor

Třída *Uzel* je z hlediska funkčnosti a principu propojování bloků velice důležitá. Jednak nám tvoří grafické vazby jednotlivých bloků, ale také nastavuje jednotlivým blokům jejich vstupní bloky. Pro výpočet simulace je klíčové, aby každý blok znal všechny své vstupní bloky.

Ve třídě jsou dva druhy konstruktorů. Klasický konstruktor pro vytvoření konektoru blok-blok a pak konstruktor pro vytvoření konektoru uzel-blok. Zajímavé atributy jsou zde *pointy* a *uzly*. Jsou to opět kolekce, *pointy* jsou zde pro vykreslování samotného konektoru, aby věděl, přes které body se má vykreslit. *Uzly* zde představují všechny uzly na daném konektoru, přes tyto uzly se konektor také vykresluje. Pro přidání uzlu zde máme metodu

addUzel. Důležité je zde určit správné pořadí uzlů v kolekci, jelikož při vykreslování konektoru se berou popořadě. Konektor má definované atributy *zdroj* a *cíl*, do těchto proměnných pak ukládáme bloky propojené konektorem. Který blok je cílový a který zdrojový, určujeme právě pomocí zásuvek. Pokud je zásuvka výstupní, tak blok, na kterém je upevněna, je zdroj. V opačném případě bude cíl.

Konektor žádné události nemá, jelikož se vykresluje na základě bodů nebo uzlů, které má v sobě uložené. To znamená, pokud hýbeme uzlem, konektor se přizpůsobí, při pohybu blokem respektive koncovkou, konektor udělá to samé.

Balík setBlok

Jedná se o skupinu tříd, které tvoří grafické rozhraní pro uživatele. Pomocí ní může pohodlně nastavit jednotlivé vlastnosti a parametry bloků. Tyto parametry se pak napojí na konkrétní instanci bloku.

BlokSetDialog

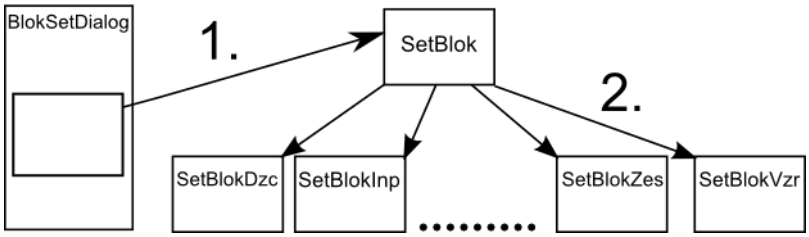
Třída vytváří základní rámec nabídky nastavení bloků. Zde jsou zobrazené vlastnosti, které mají bloky společné. Nějaký počet vstupů a výstupů, jméno, popis, a zda je výstupní. Oblast parametry však může mít každý blok odlišnou. Pro vyřešení tohoto problému jsem přidal rodičovskou třídu *SetBlok*.

Tato třída komunikuje přímo s jednotlivými bloky. Třída má v konstruktoru parametr blok, což je právě blok, který chceme nastavit. Používáme zde rodičovský objekt Blok, který pak přetypujeme na konkrétní blok. Dále zde najdeme metodu *mujInit*, pomocí této metody nastavujeme hodnoty jednotlivých grafických komponent dle druhu bloku. Právě pro tento účel máme ve třídě Blok getry a setry. Metodou *nastav* pak přiřadíme hodnoty k bloku.

SetBlok

Třída *SetBlok* reprezentuje panel pro parametry. Obsahuje jen jednu metodu *nastav*, která já určena k překrytí. Od této třídy dědí jednotlivé panely, které slouží k nastavení parametrů konkrétních bloků. My potřebujeme, aby se pro každý blok zobrazovalo odlišné nastavení parametrů. Toho docílíme přetypování rodičovského objektu *Blok* na konkrétní blok a nastavením panelu, který má každý blok jiný. Postup je tedy takový, že dvojím klinutím na blok, vytvoříme objekt, kde parametrem bude právě ten

blok, na který jsme klikli. Nyní vytvoříme instanci třídy *SetBlok* (Obrázek 2 č. 1), tuto instanci na základě objektu přetypujeme (Obrázek 2 č. 2). Po nastavení se zavolá metoda *nastav*, kterou obsahují všechny potomci, ta nám zajistí uložení nastavení. Výsledek je takový, že v základní třídě *BlokSetDialog*, se nastaví panel pro parametry bloku, na který právě klikneme.



Obrázek 2 - Princip nastavení parametrů

Hlavní výhoda toho složitého řešení je jednoduchá rozšiřitelnost o nové bloky, jelikož stačí vytvořit nový panel pro parametry a ve třídě *BlokSetDialog* zaregistrovat panel pro blok. O nastavení, které mají bloky stejné, se stará třída *BlokSetDialog*.

Balík kontroler

Tento balík obsahuje třídy, které se starají o komunikaci mezi datovou a zobrazovací vrstvou. Dále jsou zde třídy, které obstarávají výpočet a ukládají výsledky. Ukládání a exportování výstupních dat se provádí také v této třídě. O vizualizaci dat se stará zobrazovací vrstva.

Kontroler

Kontrolér je jedna z hlavních tříd celé aplikace, tvoří ji metody, pomocí nichž reagujeme na události vyvolané uživatelem. Shromažďuje informace o všech blocích, uzlech, konektorech. Pro uložení těchto prvků, využívá kolekce, podobně jako třída blok ukládala vstupy a výstupy. Právě pomocí kolekcí pak můžeme přistupovat k jednotlivým blokům, uzlům, konektorům.

Kontrolér disponuje metodami *addBlok*, pomocí nichž vytváříme bloky na základě stisknutí tlačítka. Tato metoda vytvoří instanci třídy daného bloku a pak ho přidá na plátno. Polohu na plátně, kam blok umístíme po stisku tlačítka, nám určuje metoda *najdiPozici*. Tato metoda obsahuje cyklus, který nám vytváří body na plátně a zjišťuje, zda je tam volné místo. Pokud místo je volné, vrátí tento bod, na něj se pak umístí blok. Tak jako metoda komponenty přidává, tak je i odebírá. Odebírání funguje tak, že kontrolér nastaví dané

komponentě atribut *vymaz* na true a vymaže ji ze své kolekce. Při překreslování se pak komponenta odstraní z plátna. Důležité je odstranit komponentu celou. To znamená, konektor se všemi uzly, které na něm jsou, blok se všemi svými koncovkami.

Tato třída se stará i o různé kontroly před zahájením samotné simulace. Kontroluje, zda jsou připojené všechny vstupy, zda máme správně nastavenou vzorkovací periodu atp.

Nastavení

Třída Nastavení slouží k uchování jednotlivých vlastností a nastavení. Obsahuje statické atributy, ke kterým se přistupuje přímo přes třídu. Je rozdělena na uživatelské a systémové rozdělení. V uživatelském nastavení jsou uloženy různé barevné nastavení, šířka konektoru, velikost koncovky a uzlu. V systémové části například najdeme, který režim je aktivní, zda je povoleno hýbání s bloky a různé pomocné atributy.

Výpočet

Třída výpočet provádí samotný výpočet bloků v čase. Je potomkem třídy *JDialog,* to z důvodů zobrazování průběhu výpočtu. K tomu využíváme komponentu JProgressBar. Na ten nastavíme počáteční a koncové hodnoty, dle počtu bloků pak na něm v průběhu výpočtu nastavujeme hodnoty. Samotná simulace probíhá ve dvou krocích, první je výpočet hodnot jednotlivých bloků v konkrétní čas za dobu celé simulace. Druhá fáze výpočtu pak spočívá v získání vypočítaných hodnot od jednotlivých bloků, jimi pak naplníme výstupní tabulky a datový model grafu.

První fáze je zde reprezentovaná třídou *InitBlok*. Tato třída je součástí třídy *Vypocet* a má implementované rozhraní *Runnable*, což znamená, že probíhá v jiném vláknu než zbytek programu. Jedná se tedy o paralelní zpracování, nikoliv však za účelem zefektivnění výpočtu, ale proto, že komponenta JProgressBar to vyžaduje.

Z kontroléru si převezmeme všechny bloky určené k simulaci. A postupně je vyzveme k inicializaci, tedy samotnému výpočtu. Výsledky si uchovává každý blok sám, do té doby, než se spustí nová simulace. Výpočet si vysvětlíme podrobněji dále. Úkolem druhé fáze je získat hodnoty od všech bloků pro konkrétní čas a uložit je do datového modelu grafu a tabulky. Pro tento účel je zde třída *Prepocitej*, která je rovněž spuštěna ve vlákně. Datový model tabulky plníme po sloupcích, k tomu využijeme instanci třídy *Vektor*. Pro datový model grafu použijeme objekt z knihovny JFreeChart a to *XYSeries*.

ExcelExporter

Ukládá výsledky simulace do souboru XLS. Pro tuto potřebu zde máme knihovnu JExcelAPI. Základní objekt je zde *WorkBook,* ze kterého se vytvoří *sheet* (česky list). *Sheet* upravujeme buďto standardně přes jednotlivé buňky, k tomu se využívá objekt *Label*, nebo můžeme přidávat objekty typu *WritableImage*. Graf zde ukládáme jako obrázek PNG, který pak převedeme na *WritableImage*. Tento objekt pak jednoduše přidáme do *sheetu*. Převedení grafu na obrázek nám umožňuje knihovna JFreeChart. Tabulku do *sheetu* přidáváme právě pomocí *Labelu*, kde určíme pozici buňky a obsah, který do ní chceme vložit, převedeme na *String* a uložíme do buňky. Schéma zapojení je ukládáno podobně jako graf.

Další třídy v balíku kontrolér nejsou nijak zajímavé z hlediska fungování aplikace. Třídy NactiFIle a UlozFile slouží standardně k uložení objektů do souboru. Zde je pouze důležitě dodržet, aby třídy, které chceme ukládat, měli implementováno Serializable. Tím zajistíme, že objekt může být uložen nebo poslán sítí. Třída Simulace zde spouští výpočty a ukládá data. Využívá ji třída VystupDialog pro získání výstupních dat, která bude popsaná níže.

Balík zobraz

Zde jsou uloženy třídy, které slouží ke komunikaci s uživatelem. Nebo zobrazení výstupních dat grafy, tabulky. Vždy se jedná o grafické rozhraní. Tyto třídy nemají přímo přístup k blokům, pro práci s bloky využívají balík kontroler.

SiproFrame

Je nejobsáhlejší třída celého programu. To je dáno tím, že je zde grafické rozhraní hlavního rámce aplikace. Tato třída má za úkol jen reagovat na události vyvolané uživatelem a spouštět metody z balíku *kontrole*r. Nemá smysl zde rozepisovat jednotlivé metody a komponenty. Důležité je jen zmínit, že jsou zde tři panely, panel pro bloky, pro ovládání a pro pohyb. Panel pro bloky je komponenta JTabbledPane, což znamená, že můžeme přidávat další záložky do panelu. Tím máme zaručeno, že budeme mít vždy dostatek místa na bloky. Panel pro ovládání je značně obsáhlý a složitý, proto je ve své vlastní třídě *Platno*.

Platno

Platno je třída, která je potomkem JPanel. Právě na plátně vytváříme schémata zapojení. Pokaždé, když se plátno překresluje, kontroluje jednotlivé komponenty, zda nemají aktivní atribut *vymaz.* Právě na základě tohoto atributu pak komponenty odstraní či nikoliv. Další funkce plátna je vykreslovat pomocnou přímku pro tvorbu konektorů, která simuluje budoucí konektor.

VystupDialog

Prezentace výstupních dat simulace je graficky znázorněna právě pomocí této třídy. Vlevo najdeme panel pro jednotlivé bloky, zde si určujeme, které bloky budou zobrazeny na výstupu, defaultně jsou zobrazeny všechny. Pro načtení bloků do panelu a určení, zda jsou výstupní, využíváme metodu *initCheckBoxi*. Metoda *nastavBloky* nám pak dle hodnot CheckBoxu nastaví dané bloky. Vyvolání této akce uživatel udělá tlačítkem pod CheckBoxi *Nastav*.

Dále zde najdeme panel pro graf. Inicializaci grafu zde provádí metoda *initGraf*, která vychází z knihovny JFreeChart. Nejprve vytvoříme graf a určíme jeho typ, zde je reprezentován jako ChartFactory.createXYLineChart. Z vypočítané simulace si vezmeme data, která bude graf reprezentovat. Pro rozšířené nastavení grafu si odvodíme *renderer*, pomocí něhož pak můžeme upravovat vykreslování grafu. Nyní zbývá jen graf zobrazit, nejprve vytvoříme *ChartPane* a poté ho jen přidáme do našeho panelu.

Pod grafem najdeme dvě tlačítka *Skrýt/Zobrazit přímky* a *Skrýt/Zobrazit body*. Metody těchto tlačítek využívají právě *rendereru* k určení, co se má vykreslit. V dolní liště nalezneme funkce – jako export do Excelu, zvětšení grafu nebo tabulky na celou obrazovku. Export byl popsán již v balíku *kontroler*, co se týče zobrazování grafu a tabulky na celou obrazovku máme tu na to dvě metody, *zobrazGraf* a *zobrazTabulku*. Tyto funkce jsou vhodné pro detailnější pohled na graf nebo tabulku, obzvláště je to výhodné v případě používání více monitorů, kde můžeme na každý monitor umístit jedno zobrazení. Poslední část výstupu zabírá tabulka, model pro tabulku máme připravený již ve třídě simulace, stačí si jej pouze vzít.

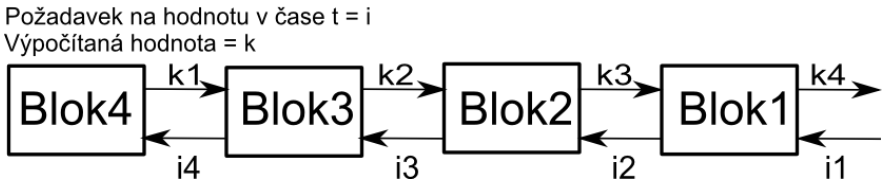
Vstupní parametry simulace má za úkol zajistit třída *StartDialog*. Odebere parametry, vyvolá různé kontroly schématu a vytvoří instanci třídy Simulace, tím započne samotný výpočet.

3.3 VÝPOČET

Výpočet jednotlivých hodnot v čase je nejdůležitější část simulace. Já jsem k ní přistoupil jako k obecnému výpočtu. Nezajímáme se o jednotlivé bloky, zajímá nás jen hodnota jednoho konkrétního bloku v daný moment. Jak jsem již uváděl, máme kolekci všech bloků, které budeme simulovat. Pokud uživatel neurčí jinak, vezmeme první blok, který byl přidán na plátno.

Takže máme určen první blok, od kterého chceme zjistit hodnoty pro všechny kroky určené uživatelem. U tohoto bloku zavoláme metodu *init* pro konkrétní čas. Tuto funkci zdědili všechny bloky od třídy *Blok*. Ta nám zavolá metodu funkce, kterou má již každý blok jinou. Metoda funkce nám vrací hodnotu daného bloku pro konkrétní čas. Pokud je blok vstupní, není zde žádný problém, vrátí hodnotu, ta se uloží a pokračuje se dále. Jestliže ale blok má nějaké vstupy, musíme nejprve získat hodnoty pro daný čas od bloku, který je k němu připojen na vstup. Jelikož každý blok zná bloky na svém vstupu, tak stačí pouze zavolat opět *init* u bloku na vstupu.

Toto chování se opakuje tak dlouho, dokud nenarazíme na blok, který nemá vstup, může tedy poskytnout hodnotu. Počet opakování je stanoven počtem kroků, v každém kroku musím provést výpočet. Princip fungování ukazuje Obrázek 3.



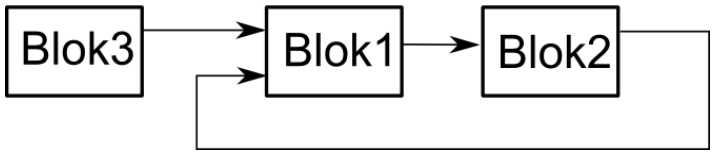
Obrázek 3 - Princi výpočtu

Až se daný blok dopočítá pro všechny hodnoty, nastaví se mu status *nainitovany* na true. To znamená, že pokud od něj budeme potřebovat hodnotu, nebudeme provádět znovu výpočet, ale vezmeme si přímo hodnotu pro daný čas.

Vezme se další blok z kolekce a otestuje se, zda je nainitovaný, pokud ano, přeskočí se. Po otestování všech bloků je výpočet ukončen a výsledky jsou zobrazeny na výstupu.

V případě jednoduchého zapojení bez zpětné vazby tento postup výpočtu funguje spolehlivě a bez problému. Jisté problémy nastávají u složitějších zapojení se zpětnou vazbou, viz Obrázek 4.

Každý blok, pokud začne provádět výpočet, nastaví atribut *pracuje* na TRUE. Tento atribut pak testujeme vždy, když žádáme nějaký nenainitovaný vstupní blok o hodnoty. Pokud je FALSE, pokračujeme dál v nám již známém postupu, jestliže je ale TRUE, vyžádáme od bloku jeho poslední vypočtenou hodnotu, to znamená tu z minulého kroku. V prvním kroku, kde ještě žádný výpočet neproběhl, vrací nulu.



Obrázek 4 - Zapojení se zpětnou vazbou

Vezmeme v úvahu zapojení, viz Obrázek 4, funkce jednotlivých bloků nemají v tomto zapojení význam. Simulace začne například od bloku 1, v tomto zapojení je jedno kdo začne, výsledek se nezmění. V první řadě blok 1 požaduje hodnotu od bloku 3, to není problém, blok 3 nemá žádné vstupy, takže hodnotu bez problému poskytne. Nyní požadujeme vstup 2, což znamená hodnotu od bloku 2. Blok 2 již vstup má, takže ho požádá o hodnotu, jelikož ale blok 1 již pracuje (*pracuje* = TRUE), vrátí mu hodnotu z minulého kroku.

Tento postup je u většiny zapojení v pořádku a dává korektní výsledky. Existují ale bloky, které mají funkce přímo závislé na čase. Jedná se o bloky, u kterých nastavujeme vzorkovací periodu tedy bloky VZR, DZC, TDP, TVA. Představme si stejné zapojení nyní, ale bloky budou mít konkrétní funkci. Blok 1 bude simulovat sumu (obyčejný součet vstupů), blok 2 bude vzorkovač s nastavenou periodou 1sec (každou vteřinu propustí vstupní hodnotu, jinak nulu), blok 3 bude třeba konstanta, která bude generovat 1. Krok řešení nastavíme na 0.1sec a čas simulace na 10sec. Simulace začne třeba opět od bloku 1, v prvním kroku přečtu 1 z bloku 3, v druhém kroku požádám blok 2 o hodnotu, je čas 0, tudíž na požadavek zareaguje a požádá blok 1 o hodnotu. Blok 1 již počítá, takže vrátí hodnotu 0. Výstup vzorkovače v čase 0 bude tedy nula, ačkoliv blok 3 generuje v obvodu 1. Celý výstupní graf se tedy posune o vzorkovací periodu. Pokud simulace bude začínat od

bloku 2, tedy vzorkovače, bude graf začínat od nuly. Jelikož suma udělá součet 0 a 1, vrátí v čase 0 jednotku. Toto nemusí nutně znamenat chybu.

Chyba nastává v momentě, kdy je více bloků závislých na čase za sebou. Bereme-li v úvahu podobné zapojení jen mezi blok 2 a blok 1, přidáme další vzorkovač. Máme tedy na vstupu konstantu, která nám generuje 1, je připojena na sumu, suma má výstup připojený na vzorkovač a tento vzorkovač má připojený výstup opět na vzorkovač. Výstup posledního vzorkovače směřuje zpět do sumy na vstup 2. Vzorkovače mají nastavenou stejnou periodu. Spustíme simulaci se stejnými parametry jako předtím a začneme simulaci od prvního vzorkovače (ten, na který je připojen výstup sumy). Začneme tedy nám dobře známým postupem, vzorkovač požádá sumu, vstup 1 u sumy je stále bez problému, pro vstup 2 suma požádá druhý vzorkovač o hodnotu. Tento vzorkovač zkusí marně požádat první vzorkovač o hodnotu, jelikož první vzorkovač pracuje, vrací výpočet pro předchozí krok, tedy 0. V dalším kole vzorkovač dva nepropouští hodnoty, tudíž vrátí nulu. Tento cyklus pokračuje pořád dokola až do další periody, tedy do jedné vteřiny, zde opět požádá druhý vzorkovač první vzorkovač o hodnotu, ale jelikož stále pracuje, vrátí předchozí hodnotu, což je opět nula. Výsledek je takový, že druhý vzorkovač bude stále ukazovat nulu.

Tento problém nastává vždy, když budeme mít ve schématu více bloků se vzorkovací periodou a budeme začínat výpočet od toho, který je ve schématu logicky dříve postavený a jejich vzorkovací perioda bude stejná. Proto je do dialogového okna pro start přidána možnost zvolit začátek simulace, tím můžeme eliminovat tyto chyby.

3.4 Přidání bloku

Pro jednoduchost budeme přidávat blok s primitivní funkcí, není účelem této kapitoly řešit složité výpočty, ale nastínit postup přidání bloku. Takže budeme přidávat třeba sumu, která již v programu je. Jako první je dobré si připravit obrázek schématické značky bloku. Pokud se značka již nachází v programu, můžeme použít ji, jinak musíme připravit obrázek v nějakém programu, já použil Inkscape. Obrázek by měl mít velikost 80x56 pixelů a být typu PNG, ale není to podmínkou. Nyní když máme připravený obrázek, můžeme začít přidávat samotný blok.

Nejprve musíme vytvořit třídu bloku. Název není podstatný, ale měl by mít určitá pravidla, aby byl kód dobře čitelný. Já používám název Blok + název bloku, v našem případě

BlokSum. Této třídě musíme v prvé řadě nastavit extends *Blok*, neboli nastavit ji jako potomka třídy *Blok*. V dalším kroku vytvoříme konstruktor Sumy, zajímá nás konstruktor s parametry *String* a *Point*, kde *String* je jméno bloku a point umístění bloku. V konstruktoru nastavíme vstupy a výstupy bloku, v našem případě suma má dva vstupy a jeden výstup. Teď už zbývá nastavit v konstruktoru jen obrázek se schématem a vyplnit popis. Vzorový konstruktor bloku najdeme v příloze A. Nyní potřebujeme určit chování bloku, tzn. nastavit metodu *funkce*, budeme ji tedy překrývat. Právě v této metodě určujeme, co bude daný blok dělat. V parametrech metody najdeme *celkovyCas* a *casPrimoTed*, právě pomocí těchto parametrů určujeme, kde se ve výpočtu nacházíme. Pokud potřebujeme, můžeme překrýt ještě metody *vynuluj* a *nastav*. Pomocí těchto metod můžeme nastavit defaultní hodnoty tak, jak potřebujeme. Naši metodu můžeme vidět v příloze B.

Pokud máme vytvořenou třídu, zbývá nám modifikovat grafické rozhraní tak, aby uživatel mohl vytvářet objekty této třídy. Musíme tedy upravit třídu *SiproFrame*, pokud používáme NetBeans, tak jednoduše přidáme tlačítko, a nastavíme Event Action. Zde musíme vytvořit objekt třídy *BlokSum*, jelikož ale dodržujeme architekturu MVC, nesmíme vytvořit objekt přímo z třídy, ale přes kontrolér, takže musíme ještě upravit třídu *Kontroler*. Zde přidáme metodu, která nám vytvoří instanci daného bloku a přidá ji jednak do kolekce bloků a také na plátno. Metodu najdeme v příloze C.

Nyní máme blok přidaný, je zobrazen jak na plátně, tak v kontroléru a můžu s ním provádět výpočty. Zatím ale stále nemáme nastavenou nabídku bloku s informacemi o něm a možností nastavení parametrů. To provedeme dvěma kroky, v prvním kroku vytvoříme opět novou třídu, u které nastavíme extends SetBlok, neboli určíme ji potomkem třídy *SetBlok*. Zde si vytvořím svůj panel, na kterém si nastavíme parametry, záleží čistě na programátorovi, jak si panel nastaví, jaké přidá komponenty. Důležité je zde komunikovat s třídou bloku, nejčastěji pomocí getru a setru. Náš blok žádné parametry nemá, proto je v příloze uveden jiný blok. Teď už jen blok zaregistrujeme ve třídě *SetBlokDialog* ukázku nalezneme v příloze D.

4 UŽIVATELSKÁ DOKUMENTACE

V této kapitole se seznámíme s aplikací. Jako první si ukážeme, jak aplikaci spustit. Výhodou aplikace je, že se nemusí instalovat, lze ji spustit rovnou z CD (Příloha E). Na CD najdeme také uložená schémata, se kterými můžeme pracovat. V další části kapitoly si popíšeme vzhled aplikace a blíže se seznámíme s jejími funkcemi.

4.1 SPUŠTĚNÍ

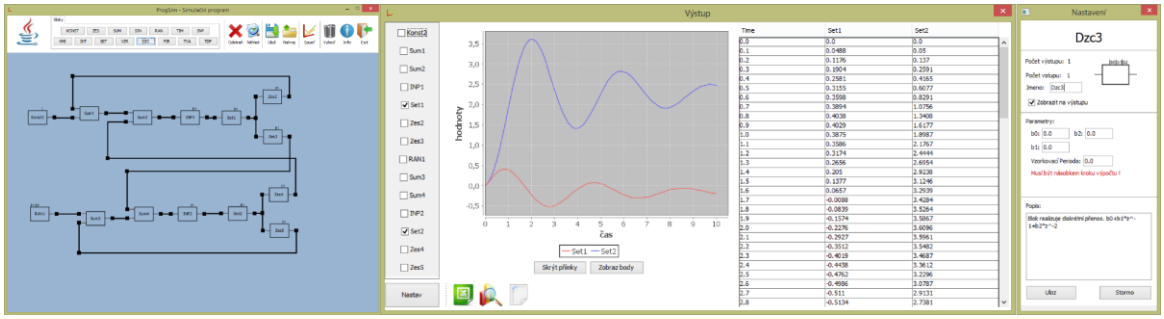
Aplikace je napsána v jazyce Java, to znamená, že můžeme aplikaci spustit na libovolném operačním systému. Abychom ale vůbec mohli aplikaci spustit, musíme mít na počítači nainstalovanou Javu. Jestli máme nainstalovanou Javu, můžeme zjistit několika způsoby. Nejrychlejší způsob je spustit příkazovou řádku (např. alt+R > cmd > enter) a napsat „java –version“. Pokud se nám objeví, že systém nezná příkaz „java“, znamená to, že na počítači Javu nemáme nainstalovanou. Jestliže se nám vypíše verze Javy, máme Javu nainstalovanou.

Pokud Javu nemáme nainstalovanou, musíme ji nainstalovat. Instalaci můžeme provést z CD, které je přiloženo k práci. Na CD se nachází složka „Java-instal“, v této složce najdeme instalaci pro Windows. Jsou zde dvě instalace pro procesory 64bitové a 32bitové. Typ instalace zvolím podle toho, jakou verzi systému Windows používám. Chci-li zjistit, jakou používám verzi systému, kliknu pravým tlačítkem na Počítač a zvolím možnost Vlastnosti. Ve skupinovém rámečku Systém je uveden typ systému. Pokud jsme připojeni k internetu, můžeme Javu stáhnout přímo z internetových stránek www.java.com.

Aplikace je otestována na nejnovější verzi 1.7.0_51. Pokud máme nainstalovanou tuto verzi Javy, máme jistotu, že nám aplikace bude fungovat korektně s již připravenými schématy. Na starších verzích může docházet k problémům s načítáním uložených schémat. Schémata uložená na CD byla vytvořena ve verzi 1.7.0_51, může se tedy stát, že na starších verzích Javy nepůjdou otevřít.

4.2 TVORBA SCHÉMATU

Nyní se blíže seznámíme s aplikací samotnou. Podíváme se na její všechny funkce a možnosti výstupu. Na Obrázku 5 můžeme vidět, jak samotná aplikace vypadá.



Obrázek 5 - Ukázka aplikace

Na úvodní obrazovce programu najdeme panel s ovládacími prvky a plátno pro tvorbu schémat. Pokud se zaměříme na panel s ovládacími prvky, uvidíme zde tlačítka se zkratkami jednotlivých bloků. Pomocí těchto tlačítek přidáváme bloky na plátno. Pokud myší najedeme na nějaký blok, zobrazí se místo loga Javy schématická značka bloku, na který jsme najeli. Pro přidání bloku na plátno na něho klikneme levým tlačítkem myši. Blok se přidá automaticky na plátno, přidávají se zleva doprava po řádcích. Pokud se na místě nachází jiný blok, přidá se o místo dál. S jednotlivými bloky můžeme libovolně pohybovat po plátně, nelze je ovšem otočit. Pohyb provedeme tak, že najedeme na blok, stiskneme levé tlačítko myši a držíme jej, poté pohybujeme s blokem, kam potřebujeme, pak tlačítko pouze uvolníme.

Pokud máme bloky, které chceme mít ve schématu, již na plátně, stačí je nastavit a propojit. Propojení bloků se provádí přes pravé tlačítko myši, najedeme myší na koncovku bloku, stiskneme pravé tlačítko myši a držíme jej, poté táhneme konektor na koncovku, kam jej chceme připojit. Zde tlačítko myši uvolníme, tím se vytvoří konektor. Vždy když najedeme myší na komponentu, která je umístěna na plátně, zvýrazní se.

U každého bloku můžeme vyvolat okno s nastavením. Nabídku vyvoláme dvojklikem na blok, reaguje jak na levé, tak i pravé tlačítko. V horní části nabídky nalezneme název bloku, schématickou značku, počet vstupů a výstupů. Název bloku můžeme libovolně měnit podle potřeby, počet vstupů a výstupu měnit nelze, je dán charakterem bloku. V nabídce můžeme určit, zda bude blok zobrazován na výstupu. Defaultně je každý blok určen, aby se zobrazoval na výstupu. Nastavení výstupních bloků lze měnit i během výpočtu. V další části

bloku nalezneme vstupní parametry, pomocí nich definujeme chování jednotlivých bloků. Pokud je u parametru nápis „Musí být násobkem kroku výpočtu!“, tak to musíme dodržet, jinak by se pak nespustil výpočet a vyskočilo by chybové hlášení. Některé bloky neobsahují žádné vstupní parametry, proto mají toto pole prázdné. V poslední části najdeme popis bloku, který popisuje charakter bloku.

Na konektoru máme možnost měnit jednak barvu, nebo přidávat uzly. Pokud na konektor klikneme dvakrát levým tlačítkem, vyvoláme nabídku pro výběr barvy. V nabídce máme pět záložek, které nám umožňují vybrat požadovanou barvu několika způsoby. Poté, co si vybereme požadovanou barvu, klikneme na tlačítko *OK* a barva konektoru se změní. Barvu nastavujeme jen pro daný konektor, umožňuje nám lépe rozeznat jednotlivé cesty a může zlepšovat orientaci ve složitějším zapojení. Jestliže klikneme dvakrát pravým tlačítkem, přidáme na konektor uzel. Uzel má ve schématu dvě funkce, za prvé slouží jako vzdálená koncovka bloku a za druhé nám umožňuje upravovat konektor. Uzel má oproti koncovce výhodu v tom, že na něho můžeme připojit neomezeně konektorů, kdežto na koncovku pouze jeden. S uzlem můžeme libovolně hýbat, obdobně jako s bloky. Pohybem uzlů určujeme tvar konektoru. Pokud máme konektor bez uzlů, vykreslí se pouze od koncovky ke koncovce. Pohybem jednotlivých bloků se pak přizpůsobuje situaci. Přidáním uzlu přidáme bod, přes který se konektor vykresluje, tudíž pohybem s uzly a bloky tvarujeme konektor dle svých představ. Pomocí těchto komponent vytváříme schéma, které chceme simulovat.

Nyní se zaměříme na funkční tlačítka v základním rámci. Zleva máme první tlačítko *Odstraň*, pokud toto tlačítko aktivujeme, změní se nám kurzor myši na křížek, jestliže nyní klikneme na nějakou komponentu na plátně blok, uzel nebo konektor, odstraníme ji. V tomto režimu můžeme dělat všechny úkony zmíněné výše, jen musíme dávat pozor na jednoduchý klik. Deaktivací tohoto tlačítka se vrátíme zpět do režimu editace, kde nemůžeme nic smazat. Vedle se nachází tlačítko *Náhled*, aktivací tohoto tlačítka vypneme zobrazování koncovek a uzlů. Efekt tohoto tlačítka je pouze estetický, avšak pro tisk nebo export daného schématu se může hodit. V další sekci najdeme tlačítka pro uložení a načtení schématu. Klinutím na tlačítko *Ulož*, vyvoláme klasickou nabídku pro výběr uložiště. Zde si vybereme, kam chceme schéma uložit. Soubory lze uložit pouze příponou .SAV, s jinými soubory aplikace nepracuje. Pokud si tedy určíme uložiště, zbývá nám jen určit název

souboru, když nezadáme žádný název, soubor nepůjde uložit. Tlačítko *Nahraj* funguje obdobným způsobem. Zobrazí se nám stejná nabídka, kde si ale místo uložiště vyhledáme náš soubor, který chceme nahrát. Pak už zbývá ho jen označit a stisknout tlačítko *Open*. Tlačítko *Spusť* pro start simulace zatím přeskočíme a podíváme se na zbytek tlačítek *Vyhoď*, *Info* a *Exit*. Tlačítko *Vyhoď* nám vymaže celé plátno, vyhodí nenávratně všechny komponenty. Pokud klikneme na *Info,* zobrazí se nám okno se základními informacemi o programu. Tlačítko *Exit* nám ukončí aplikaci, neuložená práce bude smazána. Práci můžeme rovněž ukončit červeným křížkem v pravém horním rohu.

4.3 Spuštění výpočtu

Nyní se vrátíme ke klíčovému tlačítku *Spusť*. Pokud klikneme na toto tlačítko, zobrazí se nám další okno Výpočet. Zde zvolíme délku simulace „doba řešení“ a po jaké době vyhodnocovat výsledek, neboli „krok řešení“, oba údaje jsou zadávané ve vteřinách. Dále zde máme možnost zvolit, od jakého bloku se začne simulace počítat, tato problematika je dopodrobna popsaná v kapitole Popis programu. Nyní zbývá už jen spustit výpočet. Spustí se okno s progres barem, kde můžeme sledovat průběh výpočtu. Po úspěšném výpočtu se nám zobrazí nové okno, kde vidíme hned několik věcí.

První, co nás zaujme, je graf, který tvoří největší část okna. Pokud máme určené nějaké bloky, co se zobrazí na výstupu, uvidíme zde již nějaké křivky. Vlevo od grafu vidíme několik check boxu, které nám reprezentují jednotlivé bloky na plátně. Jejich zaškrtnutí nám určuje, zda mají být zobrazeny či nikoliv. Jednotlivým zaškrtáváním můžeme zobrazovat nebo schovávat jednotlivé bloky. Volbu musíme vždy potvrdit tlačítkem *Nastav*, které je hned pod check boxi. Pod grafem si můžeme všimnout tlačítek *Skrýt/Zobrazit přímky* a *Skrýt/Zobrazit* body. Právě pomocí těchto tlačítek si můžeme zvolit, zda chceme mít graf reprezentován pouze čárami, pouze body nebo body i čárami. Mnohem větší možnosti ohledně nastavení grafu ale získáme, když klikneme na graf pravým tlačítkem myši. Zde můžeme měnit jednotlivé barvy grafu, tloušťky čar, písmo, nebo třeba rozsah os. Graf ale můžeme i uložit jako obrázek nebo přímo vytisknout. Pokud se podíváme ještě níže pod graf, všimneme si třech tlačítek.

Pokud začneme zleva, první z nich je *Export Excel*, neboli exportovat výstup do souboru XLS. Po kliknutí na toto tlačítko se nám zobrazí opět nabídkové okno pro výběr

umístění. V dalším kroku se nás aplikace zeptá na název souboru, pokud nezadáme žádný, výstup se uloží jako defaultExcel. Spolu s tímto souborem se nám na zvoleném místě vytvoří i dva obrázky typu PNG, jeden pro graf a druhý pro schéma. Dalším tlačítkem je *Detail Grafu*. Pomocí tohoto tlačítka zobrazíme graf v novém okně přes celý monitor. Toto okno nemá žádné nové funkční tlačítka, obsahuje ovšem stejné možnosti nastavení jako graf v malém provedení. Velmi podobnou funkci má i další tlačítko *Detail Data*. Zobrazí nám v novém okně tabulku s daty.

Tabulka s daty tvoří i poslední část výstupu v okně Výstup. Tato funkce je výhodná v případě, že máme více bloků k zobrazení. Pokud máme hodně výstupních bloků v malé základní tabulce, jsou data značně nepřehledná. Pomocí této funkce zvětšíme tabulku na maximální možnou velikost, data jsou pak mnohem lépe čitelná. Tabulka nám umožňuje klasické funkce, jako je třeba možnost měnit šířku sloupců nebo je různě přehazovat dle potřeby uživatele.

Výstup simulace můžeme zavřít křížkem vpravo nahoře v rohu. Samotný výstup ukládat nemusíme, stačí uložit schéma, ze kterého výpočet vychází. Pokud máme uložené schéma, stačí jen spustit znovu výpočet a máme stejný výsledek jako před tím. Po zavření výstupu můžeme pokračovat novou simulací, nebo spustit výpočet znova s jinými parametry.

5 ZÁVĚR

Pokud se zpětně podívám na vytvořenou práci, jsem s výsledkem spokojen. Aplikace splňuje cíle, které jí byly stanoveny. Podařilo se mi implementovat všechny bloky, které jsou potřeba pro výuku v předmětu a oproti programu SIPRO nabízí o něco jednoduší návrh zapojení. Nabízí několik možností, jak pracovat s výstupem simulace.

Ovšem nesmíme zapomenout také na nedostatky, které v průběhu realizace vznikly. Kdybych se měl znova pouštět do podobného projektu, určitě bych věnoval více času návrhu architektury a dbal na jejím dodržení. Z velké části je to způsobeno nedostatkem zkušeností s vývojem podobně rozsáhle aplikace. Na jisté problémy jsem narazil až při realizaci, měnit architekturu v již takto vytvořeném programu by bylo velmi náročné, proto jsem již do ní nezasahoval. Na druhou stranu jsem se snažil, aby aplikace šla jednoduše rozšiřovat jak o nové bloky, tak i o nové funkce. Myslím si, že pokud by někdo chtěl aplikaci rozšířit o dodatečné funkce nebo nové bloky neměl by mít problém. Pokud by ovšem chtěl zasahovat přímo do jádra výpočtu, musel by věnovat nějaký čas k poznání celého systému.

Další část programu, na které by šlo ještě zapracovat, je návrh samotného zapojení. Lepší manipulaci s bloky, možnost otočení bloků, lepší tvorba konektorů. V průběhu vývoje aplikace jsem našel několik knihoven, které by šly pro tento účel použít. V tu dobu jsem už ale měl tuto problematiku vyřešenou a vracet se zpět a studovat novou knihovnu by zabralo mnoho času.

V poslední řadě bych chtěl připomenout problém s výpočtem samotné simulace. S tímto problémem jsem strávil mnoho času, snažil jsem se najít nejlepší způsob, který bych naprogramoval. Ideální řešení jsem nenalezl, ale v rozsahu výuky by měl mnou navržený způsob stačit. Pokud by někdo zamýšlel provádět s tímto programem složitější výpočty, musel by výpočet optimalizovat.

RESUMÉ

The main goal of this bachelor thesis was to implement a simple simulation application. This application must meet several parameters. Firstly, the application must be block-oriented and must perform the function of SIPRO program in the scope of Control and Simulation class at KVD. Another important condition is a friendly graphical interface. Since SIPRO has been programmed for DOS operating system, its graphical interface is not friendly at present. The final requirement for the application is the ability to export the output data into MS Excel.

In the first part of the thesis, I deal with simulation in general. I try to explain the field of simulation and to point out its big advantages. In the following part, I deal mainly with the issues of the computer simulation. This type of simulation is very widespread in all fields. In the remaining part of the thesis, I describe the application itself. I explain the structure of the program and try to show, how the program can be further extended.

I divided the development of the application into three steps. In the first step, I proposed the structure of the application. This is a very important step in the development of an application, especially if we want the application to be well extendable. In the second step, I focused on creating the application itself. In particular, I focused on the calculation and graphical interface. In the last step, I added blocks and determined their behaviour.

SEZNAM LITERATURY

1) KŘIVÝ, Ivan a Evžen KINDLER. *Simulace a modelování*. Ostrava, 2001. Dostupné z: http://vendulka.zcu.cz/Download/Free/SkriptaKindlerMS.pdf. Ostravská Univerzita.

2) HUŠEK, Roman. *Simulační modely: design exmples, signaling and memory technologies, fiber optics, modeling and simulation to ensure signal integrity*. 1. vyd. Praha: SNTL, 1987, 349 s. ISBN 01-314-2291-X.

3) PROCHÁZKA, Miroslav, Karel ANTOŠ, Bruno JEŽEK a Jan VANĚK. MOŽNOSTI VYUŽITÍ SIMULACE VE ZDRAVOTNICKÉ SLUŽBĚ A ČR. *Fakulta vojenského zdravotnictví: Univerzity obrany* [online]. 2005 [cit. 2014-04-04]. Dostupné z: http://www.pmfhk.cz/VZL/VZL5_6_2005/014-Prochazka.pdf

4) VRÁB, Vladimír a Ladislav HAVELKA. Vývojové trendy přípravy vojenských profesionálů s využitím STT. *Doktríny* [online]. [cit. 2014-04-04]. Dostupné z: http://doctrine.vavyskov.cz/_casopis/2012_2/2012_2r_4b.html

5) BRDIČKA, Bořivoj. Počítačová simulace připravuje učitele na praxi. *Učitelský spomocník* [online]. 2009 [cit. 2014-04-04]. Dostupné z: http://www.spomocnik.cz/index.php?id_document=2384

6) ČERNÝ, Michal. Počítačové simulace ve výuce fyziky. *Metodický portál: inspirace a zkušenosti učitelů* [online]. 2010 [cit. 2014-04-04]. Dostupné z: http://clanky.rvp.cz/clanek/k/g/9707/POCITACOVE-SIMULACE-VE-VYUCE-FYZIKY.html/

7) BURIETA, Ján. Simulace výrobních a logistických procesů. *Academy of Productivity and Innovations* [online]. 2007 [cit. 2014-04-04]. Dostupné z: http://e-api.cz/page/69115.simulace-vyrobnich-a-logistickych-procesu/

8) PETERKA, Jiří. Simulace vs. emulace. *EArchiv.cz* [online]. [cit. 2014-04-04]. Dostupné z: http://www.earchiv.cz/a92/a210c120.php3

9) BANKS, Jerry. *Handbook of simulation: principles, methodology, advances, applications, and practice*. Norcross, Ga.: Co-published by Engineering, c1998, xii, 849 p. ISBN 04-711-3403-1.

10) HEROUT, Pavel. *Java: grafické uživatelské prostředí a čeština*. 2. vyd. České Budějovice: Kopp, 2007. ISBN 80-723-2328-8.

11) HEROUT, Pavel. *Java - bohatství knihoven*. 3. vyd. České Budějovice: Kopp, 2008, 251 s. ISBN 978-80-7232-368-5.

12) Java™ Platform, Standard Edition 7 API Specification. [online]. [cit. 2014-04-04]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/

13) Knihovna JFreeChart vyrobí z nudných čísel hezké grafy. BARTÍK, František. *Linuxexpres* [online]. 2013 [cit. 2014-04-04]. Dostupné z: http://m.linuxexpres.cz/software/knihovna-jfreechart-vyrobi-z-nudnych-cisel-hezke-grafy

14) Export dat do Excelu pomocí jExcelApi. PÁRAL, Tomáš. *VsadNaJavu.cz* [online]. 2005 [cit. 2014-04-04]. Dostupné z: http://vsadnajavu.cz/2005-05/java-j2ee/export-dat-do-excelu-pomoci-jexcelapi/

15) BERNARD, Borek. Úvod do architektury MVC. *Zdroják.cz* [online]. 2009 [cit. 2014-04-04]. Dostupné z: http://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/

SEZNAM OBRÁZKŮ, TABULEK, GRAFŮ A DIAGRAMŮ

Obrázek 1 - Struktura balíků 18
Obrázek 2 - Princip nastavení parametrů 23
Obrázek 3 - Princi výpočtu 27
Obrázek 4 - Zapojení se zpětnou vazbou 28
Obrázek 5 - Ukázka aplikace 32

PŘÍLOHY

PŘÍLOHA A – KONSTRUKTOR BLOKU SUMA

```
public BlokSum(String name,Point p){
    super(name, p);
    //přidání koncovek
    koncovky.add(new Koncovka(this,new Point(getX(),getY()+15),"vstup"));
    koncovky.add(new Koncovka(this, new Point(getX(),getY()+(39)),"vstup"));
    koncovky.add(new Koncovka(this,new Point(getX() + getWidth(), getY() + (getHeight() /
2)),"vystup"));
    ImageIcon i = new ImageIcon(getClass().getResource("/obrazky/sum.png"));
    setIconImg(i);
    setIcon(i);
    updateKoncovky();
    popis="Výstupem z bloku je součet hodnot obou vstupů. Blok bez parametrů.";
  }
```

PŘÍLOHA B – METODA FUNKCE

```
 @Override
  public double funkce(double celkovyCas,double casPrimoTed){
    vstup1=getVstup(0, casPrimoTed, celkovyCas);
    vstup2=getVstup(1, casPrimoTed, celkovyCas);
    return vstup1+vstup2;
  }
```

PŘÍLOHA C – PANEL S PARAMETRY JEN ČÁST TŘÍDY

```
public class SetBlokRan extends SetBlok {
  private BlokRan blok;
  public SetBlokRan(BlokRan blok) {
    initComponents();
    this.blok=blok;
    mujInit();
  }
  @Override
  public void nastav(){
```

```
            if(Double.parseDouble(jTextField1.getText())>=0){
            blok.setT1(Double.parseDouble(jTextField1.getText()));
            }
            if(Double.parseDouble(jTextField2.getText())>=Double.parseDouble(jTextField1.getText())){
            blok.setT2(Double.parseDouble(jTextField2.getText()));
            }else{
              blok.setT2(Double.parseDouble(jTextField1.getText()));
            }
          }
          public final void mujInit(){
            jTextField1.setText(String.valueOf(blok.getT1()));
            jTextField2.setText(String.valueOf(blok.getT2()));
          }
        .
        .
        .
        }
```

PŘÍLOHA D – REGISTRACE BLOKU VE TŘÍDĚ BLOKSETDIALOG

```
        if (blok instanceof BlokSum) {
                p = new SetBlokSum((BlokSum) blok);
                p.setName(blok.getName());
                jPanel2.add(p);
                jTextPane1.setText(blok.getPopis());
                jLabel3.setText(Integer.toString(blok.pocVystupu()));
                jLabel5.setText(Integer.toString(blok.pocVstupu()));
    }
```

PŘÍLOHA E – ZDROJOVÉ KÓDY, APLIKACE, JAVADOC (PŘILOŽENO NA CD)