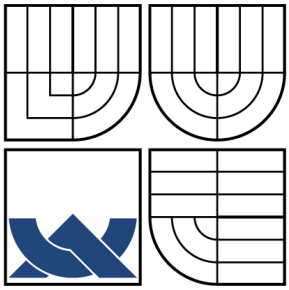
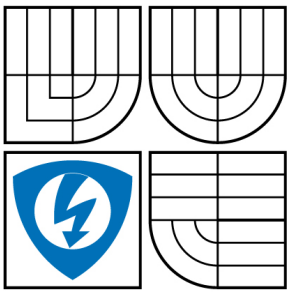


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV MIKROELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF MICROELECTRONICS

INTERNETOVÉ UŽIVATELSKÉ ROZHRANÍ PRO TVORBU ELEKTRONICKÝCH SCHÉMAT

INTERNET SCHEMATIC EDITOR

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

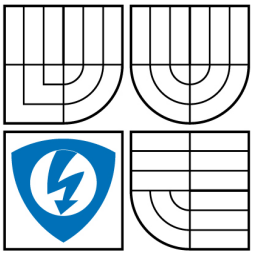
AUTOR PRÁCE — Bc. LUKÁŠ POPELKA

AUTHOR

VEDOUCÍ PRÁCE — Ing. JAROSLAV KADLEC, Ph.D.

SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav mikroelektroniky

Diplomová práce

magisterský navazující studijní obor
Mikroelektronika

Student:	Bc. Lukáš Popelka	***ID:***	89647
Ročník:	2	***Akademický rok:***	2008/2009

NÁZEV TÉMATU:

Internetové uživatelské rozhraní pro tvorbu elektronických schémat

POKYNY PRO VYPRACOVÁNÍ:

Vytvořte internetové rozhraní pro interaktivní tvorbu elektrických schémat s výsledným generováním netlistu kompatibilního s prostředím SPICE. Interaktivní prostředí bude umožňovat tvorbu jednoduchých elektronických schémat z knihovny Vámi vytvořených elektronických součástek. Výsledná diplomová práce bude obsahovat přiložené CD s Vámi vytvořeným internetovým rozhraním, knihovnou elektronických součástek a zdrojovými kódy.

DOPORUČENÁ LITERATURA:

Dle doporučení vedoucího práce

Termín zadání:	9.2.2009	***Termín odevzdání:***	29.5.2009

Vedoucí práce: Ing. Jaroslav Kadlec, Ph.D.

prof. Ing. Vladislav Musil, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Abstrakt:

Diplomová práce pojednává o tvorbě editoru elektrických schémat pracujícím pod webovým rozhranním. Editor generuje textový zápis schématu, tzv. netlist kompatibilní s prostředím Spice. Program je vytvořen v programovacím jazyku Java a využívá tak možnosti objektově orientovaného programování. Editor je umístěn na webových stránkách a spustitelný jako aplet. Práce se soustředí na výběr vhodného programovacího jazyka, návrh programu a jeho realizaci. Obsahem práce jsou také ukázky programového kódu, oken programu a schématických značek součástek. Přiřazení čísel uzlů součástek probíhá dodatečně a to až v případě požadavku na generování netlistu ze strany uživatele. Pro přiřazení čísel uzlů je použit algoritmus hledání do hloubky. Generovaný textový zápis netlistu součástek probíhá podle konvence uvedené v referenční příručce programu OrCAD PSpice.

Abstract:

The diploma thesis deals with creating of electronic schematics in editor using web interface. The editor generates electrical circuit text file according to Spice netlist specification. The program has been created in Java and takes an advantage of object oriented programming language. The editor is a part of a web page and is executable as an applet. The diploma thesis describes a programming language selection, program layout and implementation. Thesis contains programming code examples, window illustration and component drawings. Depth-first search algorithm has been used for nodes number assignment. An OrCAD PSpice reference guide was used for netlist.

Klíčová slova:

Java, objektově orientované programování, netlist, Spice, algoritmus hledání do hloubky.

Keywords:

Java, object-oriented programming, netlist, Spice, depth-first search algorithm.

Bibliografická citace díla:

POPELKA, L. *Internetové uživatelské rozhraní pro tvorbu elektronických schémat.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 85 s. Vedoucí diplomové práce Ing. Jaroslav Kadlec, Ph.D.

Prohlášení autora o původnosti díla:

Prohlašuji, že jsem tuto vysokoškolskou kvalifikační práci vypracoval samostatně pod vedením vedoucího diplomové práce, s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury. Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne 29.5.2009

…….

Poděkování:

Děkuji vedoucímu diplomové práce Ing. Jaroslavu Kadlecovi, Ph.D. za cílené orientované vedení při plnění úkolů realizovaných v návaznosti na diplomovou práci. Dále děkuji doc. Ing. Ivo Lattenbergovi, Ph.D. za konzultaci ohledně generování netlistu.

OBSAH

1 Úvod

Předkládaná diplomová práce se zabývá návrhem řešení a tvorbou programu na kreslení elektrických schémat pracující pod webovým rozhraním. Obsahem práce je návrh řešení zadaného tématu, postupu a realizace. Jsou zde popsány algoritmy použité při tvorbě programu včetně ukázek programovacího kódu. Program je vytvořen v objektově orientovaném programovacím jazyku Java. Úkolem práce však není vysvětlit principy objektově orientovaného programování (OOP). Předpokládá se, že čtenář je obeznámen alespoň se základní problematikou OOP, terminologií, potažmo programováním vůbec. Jako úvod do problému OOP spolu s výukou programovacího jazyka Java doporučuji literaturu [4], [5] nebo [7]. Dokumentace k aplikačnímu rozhraní Javy je dostupná zde [9]. Obsahuje ucelený seznam všech tříd metod Javy, včetně popisu vysvětlení jejich funkcí. V textu je také několikrát odkazováno na konvenci PSpice. Konvence PSpice je detailně popsána v referenční příručce, která je standardně součástí instalace programu PSpice OrCAD Capture, nebo je k dispozici např. zde [16].

Použitý styl formátování odpovídá konvenci pro psaní literatury obsahující části programového kódu. Všechno co je považováno za programový kód, je psáno neproporcionálním písmem. Názvy tříd začínají velkým písmenem a jsou psány bez diakritiky. Názvy metod jsou psány v anglickém jazyce, protože ne vždy je možné najít vhodný český a hlavně výstižný ekvivalent. Pokud je uvedena pouze část kódu, je začátek a konec označen symbolem tří teček (...). Tam kde se hovoří o třídě poprvé a třída je použitá z knihovny tříd Javy, je její název uveden celý, tedy včetně cesty, např. `javax.swing.JFrame`. Ne všechny metody použité v programu jsou zde uvedeny a vysvětleny. O některých je zde pouze zmíněno, některé zde nejsou uvedeny vůbec. Týká se to především metod, které jsou buď triviální, nebo je jejich výklad natolik rozsáhlý že proto tady není prostor a je nutné nahlédnout do dokumentace aplikačního rozhranní Javy [9].

Diplomová práce je rozdělena do několika částí. První část je věnována výčtem současných vhodných programových prostředků pro řešení zadané úlohy. Jednotlivé programové prostředky jsou zde porovnány, uvedeny jejich výhody a nevýhody a na závěr kapitoly je uveden vybraný softwarový produkt. Druhá část se věnuje teoretickým návrhem programu. Je zde popsán vzhled programu, ovládání, použitý grafický formát, generování netlistu a na závěr o způsob lokalizace programu. Třetí a nejrozsáhlejší část se věnuje samotné realizaci programu. Každá podkapitola popisuje určitou třídu, nebo zahrnuje určitý problém. Podle toho jsou také odvozeny názvy jednotlivých podkapitol.

2 Programové prostředky

Cílem této práce je vytvořit grafické uživatelské prostředí, které umožní interaktivní tvorbu elektrických schémat prostřednictvím webového rozhraní, je tedy nutné vybrat takový vhodný softwarový produkt, který tato kritéria splňuje. Mezi současné nejrozšířenější softwarové produkty, které připadají v úvahu při realizaci zadané úlohy patří: Macromedia Flash, Microsoft ActiveX a Java od Sun Microsystems.

2.1 Macromedia Flash

Hlavním účelem, pro který byl tento softwarový produkt Flash vytvořen jsou multimediální internetové prezentace. S postupem času se použití tohoto programu rozšířilo i o další možnosti použití, jako je např. možnost vytvoření samospustitelného programu, bez nutnosti instalace dodatečného prohlížeče. Díky integrovanému programovacímu jazyku ActionScript, jsou stránky vytvořené ve Flash schopné komunikovat se serverem a umožňují tak pružně reagovat na podněty od uživatelů stránek. Použití tohoto programového prostředí tedy nalezneme všude tam, kde je potřeba kombinovat propracovanou grafiku spolu s interaktivním přístupem k informacím.

Výhody programu Flash jsou: snadná tvorba vektorové grafiky, přívětivé grafické prostředí, intuitivní ovládání a relativně malá velikost výstupního souboru (při vhodném nastavení a v případě nepříliš graficky náročného programu).

Nevýhodou programu je nutnost dodatečné instalace prohlížeče, který umožní prohlížení souborů vytvořených v programovém prostředí Flash. Další nevýhodou je že program pro tvorbu Flash aplikací není zdarma, je nutné jej zakoupit.

2.2 Microsoft ActiveX

Jedním z dalších programů připadajících v úvahu při realizaci programu je ovládací prvek ActiveX. AktiveX je technologie vyvinutá firmou Microsoft a je to softwarový nástroj, který je možné začlenit do normálního HTML kódu (jako tomu je i u výše zmíněného programu Flash). Programové prostředí ActiveX umožňuje také realizaci vektorové grafiky, což je z hlediska zadané úlohy přínosem. Nevýhodou prostředí ActiveX je nutnost použití prohlížeče Microsoft Internet Explorer, který jako jediný ActiveX podporuje. Pro vývoj samotných prvků ActiveX je nutný komerční programový balík Microsoft visual C++. Jakožto výhoda použití ActiveX vyplívá, že při použití internetového prohlížeče Microsoft Internet Explorer, není nutné instalovat žádný dodatečný software.

2.3 Java Sun Microsystems

Poslední softwarový produkt je z dílny firmy Sun Microsystems. Jedná se o hybridní programovací jazyk, který je současně překládán a interpretován. Programy se tedy v jazyku Java nejprve kompilují do bajtového kódu a následně interpretují přes speciální program zvaný Java Virtual Machine (JVM). Ten překládá bajtový kód do strojového jazyka, který běží na daném počítači. Virtuální stroj umožňuje, aby jeden a týž program běžel na různých počítačích s různou konfigurací a různým operačním systémem, Java je tedy multiplatformní. Java jako programovací jazyk je plně objektově orientovaná. Interpretace programu napsaného v Javě prostřednictvím webového rozhranní se děje za pomocí apletu.

Jednou z hlavních výhod programovacího prostředí Java od Microsystems je to, že je zdarma ke stažení a bez jakéhokoliv omezení v použití. Dále je to pak již zmíněná nezávislost na operačním systému spolu s nezávislostí na typu použitého internetového prohlížeče.

Mezi nevýhody patří nutnost dodatečné instalace programu (JVM) pro spuštění programu vytvořeného v Javě.

Pro snadnější tvorbu objektově orientovaného programu se používají vývojová prostředí, tzv. Integrated Developement Enviroment (IDE). Vývojová prostředí obsahují řadu funkcí jež vývoj programů zpřehledňují a vytváření programu zefektivňují. Existuje řada vývojových prostředí určených pro Javu. Pro řešení projektu se zaměřím na ty produkty, které spadají do nekomerční sféry. Do této oblasti patří např. IDE NetBeans, nebo Eclipse. Oba programy patří do skupiny rozsáhlých vývojových prostředí a ve své oblasti k nejpoužívanějším. Existuje zde ještě jedno vývojové prostředí které stojí za zmínku. Spíše než o vývojové prostředí se jedná o výukový nástroj pro studenty Javy s názvem BlueJ [2]. Výukový nástroj BlueJ v sobě slučuje možnosti klasické textového zápisu programu s možností definice jeho architektury v grafickém prostředí. Objekty a třídy jsou zde reprezentovány graficky, prostřednictvím jazyka UML (Unified Modeling Language) [3]. Na rozdíl od jiných vývojových prostředí, dovoluje BlueJ pracovat samostatně s třídami a objekty, posílat mezi nimi zprávy a to bez nutnosti spouštět celý program. Což je výhodné hlavně pro testování metod, algoritmů apod. Další předností BlueJ je jednoduché ovládání, přehlednost a hardwarová nenáročnost.

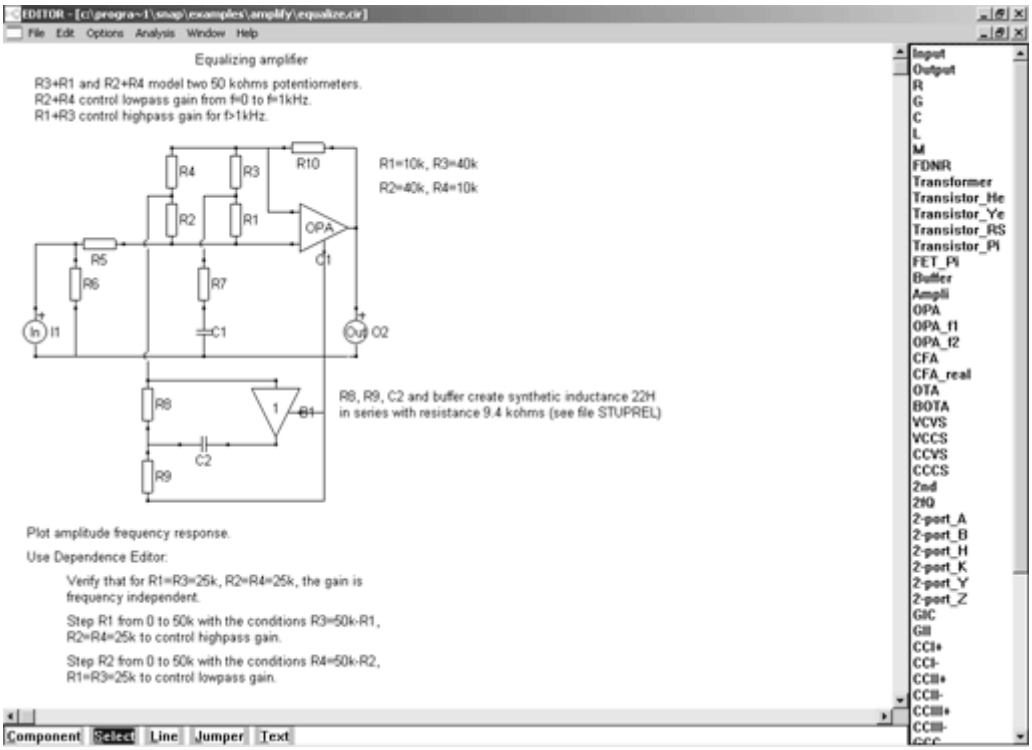
2.4 Shrnutí

Jako nejvhodnější softwarový produkt z hlediska zadané úlohy se jeví Java. Oproti ostatním výše zmiňovaným, je hlavně zdarma a poskytuje v kombinaci s vhodným IDE rozsáhlý soubor knihoven nezbytný k efektivnímu a rychlému vytvoření programu. Jako vývojové prostředí jsem proto volil prostředí BlueJ.

3 Návrh programu

3.1 Grafické uživatelské rozhraní

Inspiraci pro řešení vzhledu a ovládání programu jsem hledal v editoru schémat, který je součástí simulačního programu pro symbolickou analýzu Snap [1]. Vzhled a ovládání tohoto programu je jednoduché a intuitivní. Při návrhu vzhledu a rozmístění ovládacích prvků jsem uvažoval obdobně jako v případě editoru určeném pro Snap. Jak vypadá hlavní okno editoru pro symbolickou analýzu je vidět na obr.3.1.

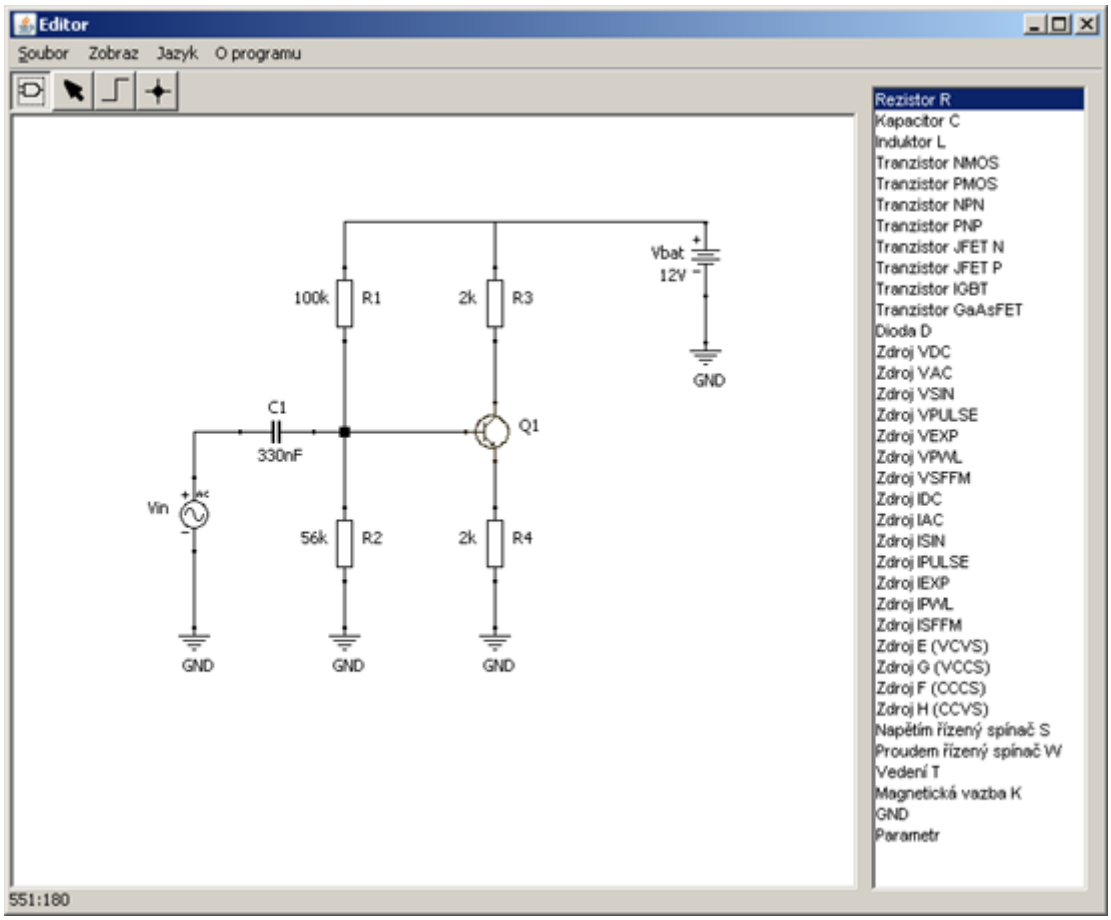


Obr.3.1: Okno editoru pro symbolickou analýzu

Okno editoru je složeno ze tří ovládacích prvků a tzv. kreslícího plátna. Ovládacími prvky editoru jsou: horní menu, seznam součástek a spodní lišta informující o stavu kurzoru. Kreslení schématu probíhá na kreslícím plátně, které zabírá největší část editoru. Součástky se vybírají ze seznamu umístěného napravo. Po výběru a umístění součástky na plátno ji lze libovolně posouvat, rotovat nebo i spojit. Důležitou věcí při ovládání programu je nastavení režimu kurzoru. Kurzor je možné nastavit do 4 různých režimů. Jsou to režimy: vytvoření součástky, výběru součástky nebo editace parametrů, kreslení propojovacích vodičů a vytvoření nevodivého spojení při křížení vodičů.

Editor vytvořený v Javě se od editoru určeného pro Snap liší v několika maličkostech. Menu pro indikaci a změnu režimu kurzoru je umístěno nahoře a každý stav je indikován vlastní ikonou. Stavy kurzoru mají stejnou funkci jako v editoru Snap kromě posledního

stavu, který má funkci odlišnou. Poslední stav kurzoru slouží k vytvoření vodivého spojení dvou křížících se vodičů. Celkový pohled na okno editoru vytvořeného v Javě je na obr.3.2.

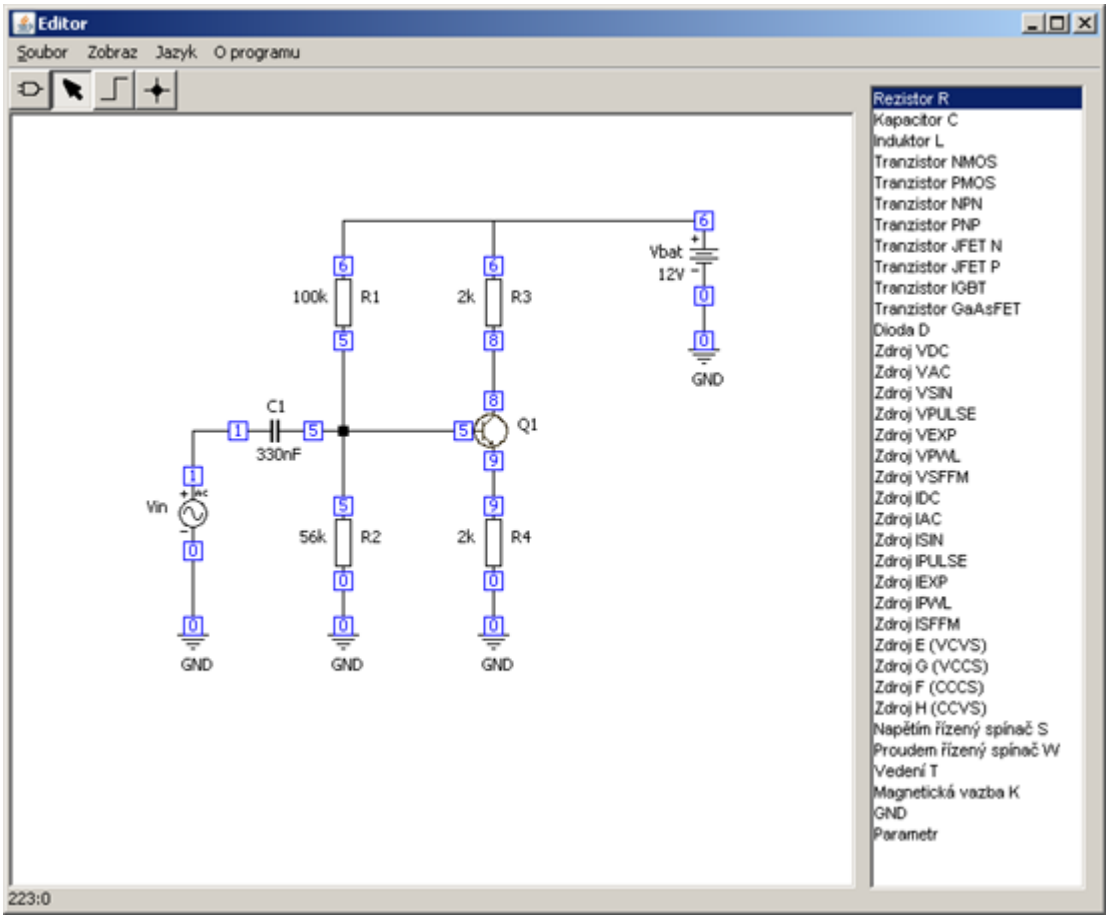


Obr.3.2: Okno editoru vytvořeného v Javě

Horní menu obsahuje položky *Soubor, Zobraz, Jazyk* a *O programu*. Položka *Soubor* má ve své nabídce možnost vytvoření nového schématu a ukončení práce s editorem. Položka *Zobraz* nabízí možnost zobrazení čísla uzlů, generování netlistu a vyvolání okna s nastavením simulace. Menu položka *Jazyk* nabízí změnu lokalizace programu na jiný.

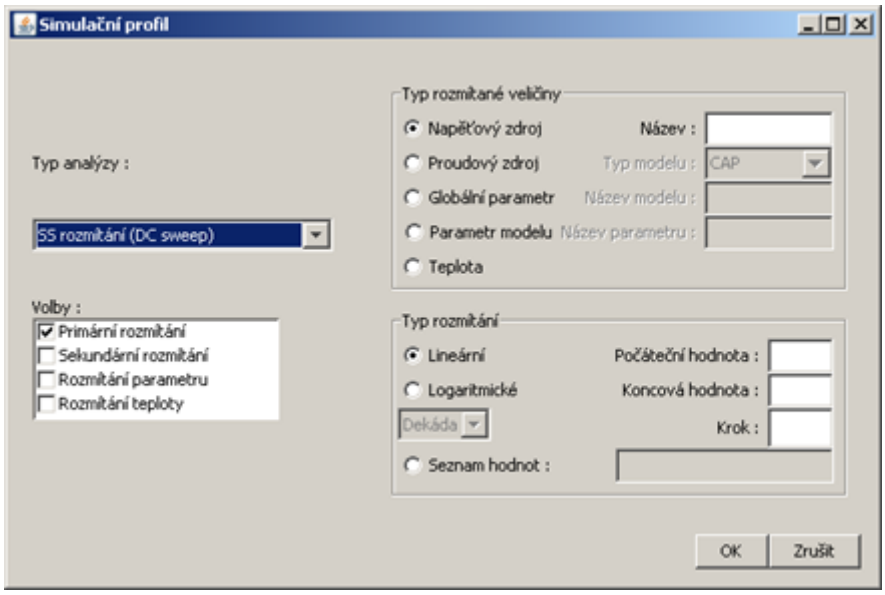
V položce soubor by se určitě hodila nabídka uložení a znovu načtení nakresleného schématu. Implementace této funkce zahrnuje jedno úskalí. Na aplikace spuštěné přes webové rozhranní, se totiž vztahují také různá omezení ohledně bezpečnosti toho, co může aplikace na klientském počítači vykonávat a co ne. Taková klientská aplikace nemá například přístup na souborový systém počítače, nemůže spouštět jiné aplikace apod. Problém s bezpečnostním omezením lze v Javě obejít a to vytvořením autorizačního certifikátu, který by umožnil přístup na souborový systém klientského počítače (takový přístup používá například internetové bankovnictví Komerční banky a.s.). K vytvoření autorizačního certifikátu je nutná registrace, která je ovšem zpoplatněná nemalou částkou. Právě kvůli těmto limitujícím omezením zde funkce načtení a uložení schématu chybí.

Funkci zobrazení čísel uzlů je vhodné použít pro zpětné porovnání vygenerovaného netlistu nebo také pro ověření správného propojení součástek. Čísla uzlů se zobrazují na místě přípojných bodů součástek, viz obr.3.3.



Obr.3.3: Zobrazení čísel uzlů

Nastavení simulačního profilu daného schématu probíhá v samostatném okně. Nemusí se tedy vkládat do vygenerovaného netlistu ručně. Simulační profil podporuje nastavení čtyř typů analýz. Jsou to: stejnosměrná analýza (DC sweep), střídavá analýza (AC sweep), analýza v časové oblasti (Transient) a výpočet pracovního bodu (Bias Point). Každá analýza má různý počet parametrů nastavení. Okno s nastavením simulačního profilu je shodné s oknem z programu PSpice OrCAD Capture a je na obr.3.4. Více je o okně simulačního profilu uvedeno v kapitole 4.7.



Obr.3.4: Okno nastavení simulačního profilu

.

3.2 Grafický formát součástek

Každá součástka editoru je na kreslícím plátně reprezentována svojí schématickou značkou. Schématická značka je vlastně obrázek, který může být v počítačové podobě uložen buď jako bitmapa nebo jako soubor vektorů, pak se jedná o tzv. vektorovou grafiku. Výhodou vektorové grafiky je její datová nenáročnost. Vektorový zápis formátu je v porovnání s bitmapovým zápisem formátu podstatně datově méně náročný. Další předností vektorového formátu je jeho možnost libovolného zvětšování a zmenšování bez ztráty kvality, což bitmapové obrázky neumožňují. Java ovšem standardně nepodporuje žádný vektorový formát obrázků. Editor navržený v Javě obsahuje konečný počet součástek, uživatel tedy nemůže vytvořit vlastní součástku, nebo stávající modifikovat. Z hlediska časové náročnosti tvorby obrázku je méně časově náročné jej vytvořit jako bitmapu v některém volně dostupném editoru, než jej vytvořit jako vektorovou grafiku textovým zápisem v programovém kódu. Jako grafický formát součástek jsem tedy zvolil bitmapovou formu zápisu.

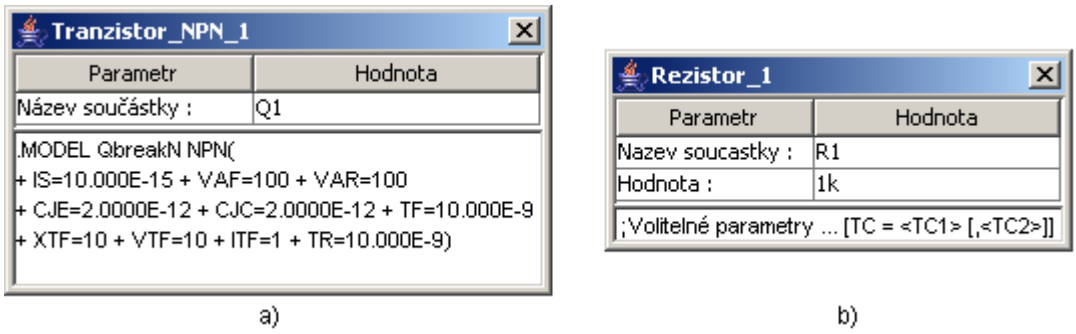
Mezi zástupce bitmapové grafiky patří formáty JPG, GIF a PNG. Zatímco formáty GIF a PNG používají bezeztrátový algoritmus komprese, tzn. nedochází ke ztrátě kvality obrázku, formát JPG používá ztrátový algoritmus komprese. Formát JPG se v převážné míře používá k ukládání fotografií. Formáty GIF a PNG navíc poskytují možnost vytvoření průhledného pozadí. Tyto obrázky tedy mohou být transparentní, s libovolnou mírou průhlednosti. Transparentnost obrázků je v editoru také využívána. Formát GIF nedosahuje tak velkého kompresního poměru ve srovnání s formátem PNG.

Jako optimální bitmapový formát obrázku, se tedy jeví formát PNG, protože je transparentní a má větší kompresní poměr oproti formátu GIF. Datová náročnost schématických značek se pohybuje v řádu stovek bajtů. Velikost obrázku největší použité součástky, vedení T, o rozměru 61 x 51 pixelů je 516 bajtů, což je ve srovnání s velikostí celého programu (cca 270kB) zanedbatelné.

Další nezbytnou vlastností editoru je vzhledem k jeho pohodlnému ovládání, nezbytnost přichytávání součástek k předem definovanému rastru pozadí. Tato funkce je v editoru vytvořeném v Javě implementována také. Zde ovšem není možné měnit rozteč rastru. Změna rastru by byla výhodná pouze v případě, že by součástky byly realizovány vektorově a nikoliv bitmapově. Pak by bylo možné změnou rastru změnit i velikost zobrazení komponent. Protože je zde použit bitmapový formát obrázků, změna rastru není nutná. Tím je sice zamezena možnost zvětšování a zmenšování velikosti součástek ve schématu, ale je tím podstatně usnadněno ovládání programu. Jako optimální rozteč je napevno zvolena hodnota 10 pixelů, což je plně dostačující i pro práci ve vysokém rozlišení obrazovky.

3.3 Parametry součástek

Každá součástka definuje svoje parametry. Vytvořením součástky, resp. jejím umístěním na kreslící plátno se nastaví její parametry na předem definovanou hodnotu. Tyto parametry je ve většině případů nutné modifikovat, zcela změnit, nebo definovat nové. Z uživatelského hlediska jsem zvolil osvědčený postup, kdy volbou stavu kurzoru do módu výběru součástky a následným dvojitým stiskem levého tlačítka myši nad součástkou se otevře okno s nastavením parametrů součástky. Možnosti a volby parametrů součástek se liší podle toho o který typ součástky se jedná, jestli součástka obsahuje model atd. Jediný parametr, který je společný všem součástkám, je její název, resp. název který je zobrazen ve schématu. Například součástka rezistor obsahuje parametry: název součástky, hodnota a volitelné textové pole, do kterého je možné vložit dodatečné parametry. Obsah textového pole se pak vypíše do netlistu, jako poslední. Naproti tomu součástka tranzistor NPN obsahuje model (standardně definovaný podle referenční příručky PSpice jako QBreakN) který je volně modifikovatelný v textovém poli. Uživatel si zde může vložit vlastní parametry modelu, které pak budou použity při generování netlistu. Okno s nastavením parametrů je na obr.3.5 a) a b).

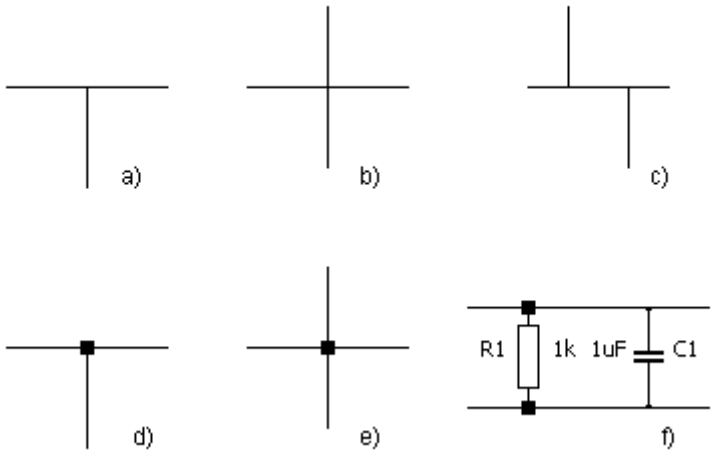


Obr.3.5: a) Nastavení parametrů tranzistoru b) Nastavení parametrů rezistoru

3.4 Propojování součástek

Jednou z věcí, nad kterou je také nutné se zamyslet před vytvořením editoru, je systém propojení součástek a pravidla pro kreslení schémat. Vodivé spojení součástek je realizováno vytvořením propojovacího vodiče mezi vývody, které mají být spojeny. V editoru určené pro Snap je jakékoliv křížení vodičů automaticky považováno za vodivé spojení. K vytvoření nevodivého křížení dvou vodičů je nutné do schématu zařadit prvek typu jumper. V tomto směru se můj návrh programu liší. K vytvoření vodivého spojení dvou křížících se vodičů je nutné na místo křížení vložit prvek spoj. Není-li použit prvek spoj, je křížení vodičů automaticky pokládané za nevodivé. Další, uživatelsky, nepříjemnou vlastností editoru je, že vzájemné spojení vývodů součástek a připojených vodičů není trvalé. Např. máme-li spojeny dva vývody součástek mezi sebou, pak tažením jedné ze součástek z jednoho místa na druhé, dojde k přerušení vodivého spojení. Nedochází tedy k automatickému překreslení spoje. Pro jeho obnovení je totiž nutné celý spoj znovu nakreslit. Z uživatelského hlediska to je sice nepříjemné, ale z programátorského hlediska je to však podstatné zjednodušen. Profesionální softwarové aplikace však mají tuto nedokonalost odstraněnou. Problém je zde vyřešen s různým stupněm úspěšnosti. Nejde vždy jen o pouhé znovu vytvoření pravoúhlého spojení při přesunutí z bodu A do bodu B, ale je nutné vzít v potaz současné otočení součástky, nebo také již vytvořené vodivé spoje ve schématu. To znamená, vytvořit vodivý spoj takovou cestou, při kterém nedojde k žádnému křížení s některou již vytvořenou součástkou, nebo překrytím s některým již vytvořeným vodičem. Cílem této práce je vytvořit jednoduchý editor grafických schémat, tedy implementace automatického vytváření propojovacích cest (tzv. routování) je sice vhodná věc, ale nikoli nutná.

Nyní je na místě definovat pravidla, podle kterých se bude kreslení schémat probíhat, tak aby bylo předem zřejmé, co bude považováno za vodivé spojení a co za nevodivé křížení. Možné varianty vodivého spojení a nevodivého křížení jsou na obr.3.6.



Obr.3.6: Možné varianty vodivého a nevodivého spojení vodičů resp. součástek

Na obrázcích a, c, d, e jsou vidět různé kombinace vodivého spojení vodičů a to jak s využitím prvku spoj tak i bez něj. Obrázek b ukazuje variantu křížení vodičů bez vodivého spojení. Varianta f ukazuje připojení součástky tzv. na sběrnici, kdy je k jednomu vodiči připojeno více součástek. Obě dvě varianty připojení součástek na sběrnici, a to jak s použitím prvku spoj nebo bez něj jsou ekvivalentní.

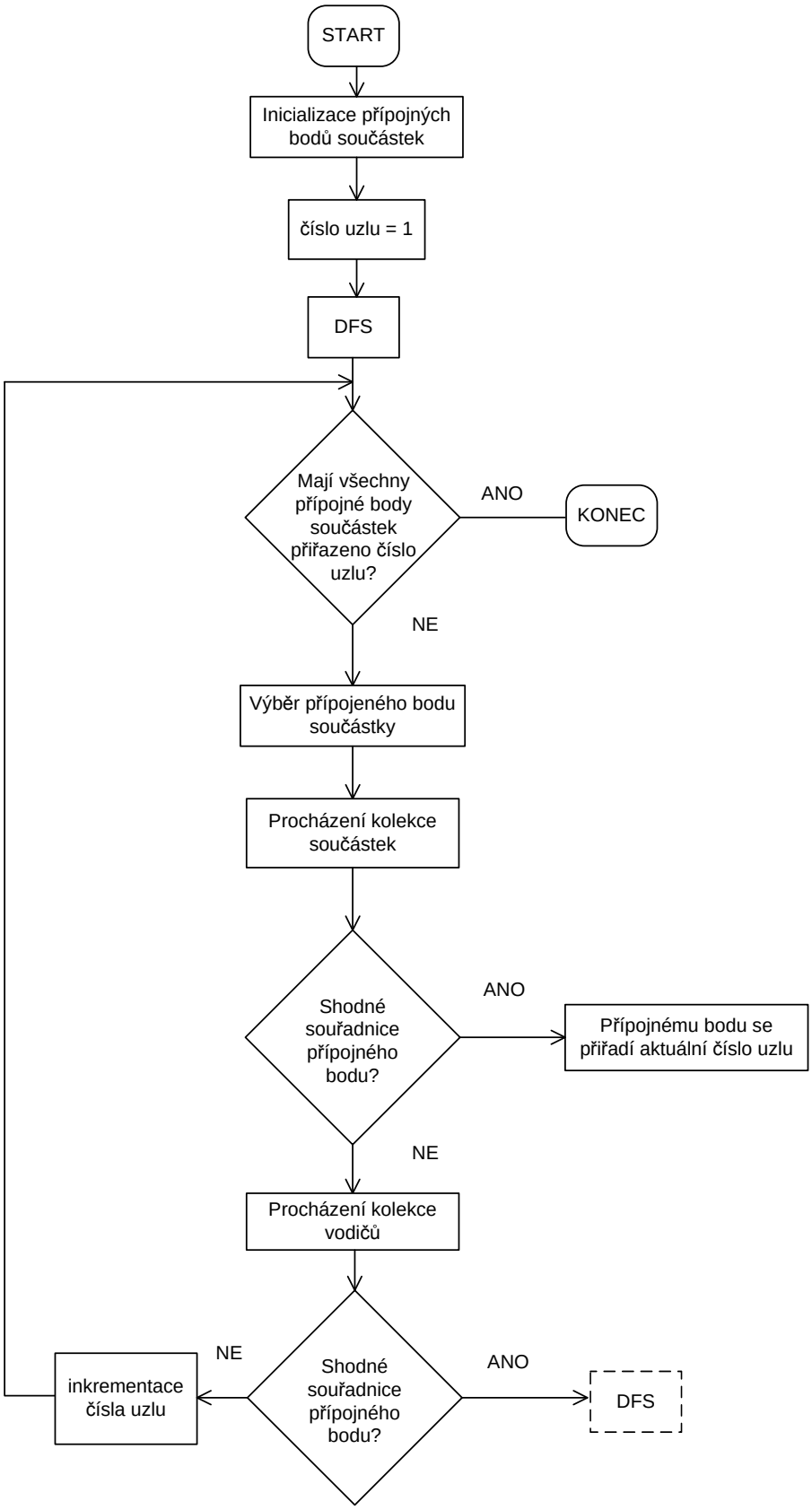
3.5 Generování netlistu

Každá součástka obsahuje předem definovaný počet přípojných bodů (resp. vývodů) pomocí kterých je zapojena do obvodu. Přípojný bod je informace o aktuálních souřadnicích bodu na kreslícím plátně. Jelikož zde není implementována funkce automatického navrhování cest, probíhá přiřazení čísel uzlům dodatečně, tedy až v případě požadavku ze strany uživatele na generování netlistu. Každý přípojný bod obsahuje atribut, ve kterém je uloženo číslo uzlu do kterého je přípojný bod připojen. Tak jako každá součástka má své přípojné body, tak i vodič má své přípojné body (resp. počáteční a koncový bod). Všechny komponenty, které jsou na kreslícím plátně vytvořeny, jsou ukládány do kolekce a to v pořadí v jakém byly vytvořeny. K vytvoření netlistu je třeba tuto kolekci rozdělit na kolekci vodičů a kolekci součástek, protože je s oběma komponentami zacházeno jinak. Algoritmus, který je pro generování netlistu použit, se nazývá prohledávání stromu do hloubky (depth-first search) [4]. Hlavní podstatou algoritmu je nejprve projít celou kolekci a inicializovat všechny přípojné body součástek. (např. tak, že se všem přípojným bodům přiřadí počáteční hodnota). Na každý přípojný bod součástky se pak zavolá rekurzivní metoda, která projde celou kolekci součástek a porovná, jestli jsou testované souřadnice přípojného bodu shodné s nějakým dalším přípojným bodem jiné součástky. Pokud jsou shodné, přípojnému bodu je přiřazeno stejné číslo uzlu, jaké má přípojný bod, ze kterého se vycházelo. Pokud nejsou shodné, projde se kolekce vodičů a ověřuje se, jestli je přípojný bod shodný s přípojným bodem z některého z

vodičů v kolekci. Pokud ano, na jeho opačný přípojný bod se znovu zavolá algoritmus procházení do hloubky. Celý algoritmus končí v případě, že všechny přípojné body součástek mají přiřazeno číslo uzlu. Vývojový diagram je na obr.3.7.

Algoritmus prohledávání do hloubky obsahuje proměnnou, odkazující se na pořadové číslo uzlu. Hodnota proměnné se inkrementuje v případě, když není nalezena žádná součástka se stejnou souřadnicí přípojného bodu, nebo na daný přípojný bod nenavazuje žádný vodič. V průběhu procházení algoritmu se postupně označují vodiče, které už byly testovány, aby se zamezilo zacyklení programu.

Obsahem každé instance součástek je metoda, která vrací textový popis součástky tak aby splňovala konvenci PSpice. Tato metoda je volána pro všechny součástky v kolekci a textový výpis netlistu je vytvořen v samostatném okně.



Obr.3.7: Vývojový diagram algoritmu procházení do hloubky

3.6 Lokalizace programu

V zájmu nabídky programu co nejširší skupině uživatelů je program lokalizován do anglického jazyka. Všechny lokalizované názvy a popisky použité v programu jsou uložené v samostatném textovém souboru umístěném společně s programem. Ke každému slovu nebo názvu je přiřazena jeho textová hodnota. Jak vypadá takový lokalizačního souboru je na obrázku obr.3.8. Ukázka obsahuje popisky textů k výběru typu simulace. Znak # se při čtení souboru přeskočí.

```
...
# 3.5 Option panel
primarySweep=Prim\u00E1rn\u00ED rozm\u00EDt\u00E1n\u00ED
secondarySweep=Sekund\u00E1rn\u00ED rozm\u00EDt\u00E1n\u00ED
parameterSweep=Rozm\u00EDt\u00E1n\u00ED parametru
tempSweep=Rozm\u00EDt\u00E1n\u00ED teploty
generalSettings=Hlavn\u00ED nastaven\u00ED
...
```

Obr.3.8: Úryvek lokalizačního souboru v české verzi (Resources_cs.properties)

Jak bylo řečeno v úvodu, programový kód neobsahuje žádné diakritické znaky, jako háčky a čárky. Není to z toho důvodu, že by programovací jazyk Java tyto znaky nepodporoval, ale je to z důvodu přenositelnosti a čitelnosti kódu na jiných platformách než je pouze operační systém Windows. Různé operační systémy mají totiž různé kódování znaků s diakritikou. Java standardně používá kódování Unicode [12], kdy jeden znak zabírá 2 bajty (16 bitů). K tomu aby se diakritické znaky zobrazili správně na všech platformách, je nutné všechny tyto znaky převést na kódování Unicode. Znak je pak v textu uložen ve tvaru `\uxxxx` kde znaky xxxx reprezentují dvoubajtovou hodnotu. Příklad zápisu některých znaků s diakritikou je uveden v tab.3.1.

Tab.3.1: Kódy Unicode některých českých znaků

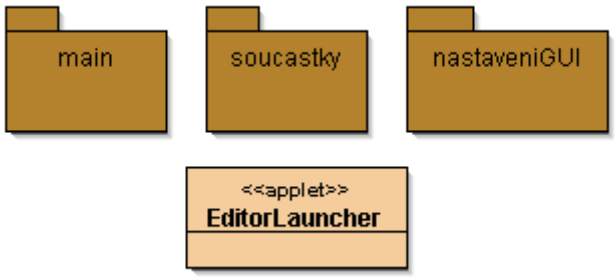
Hodnota Unicode \uxxxx	Znak
\u00E1	á
\u00ED	í
\u00FD	ý
\u00E9	é

K překladu diakritických znaků na znaky Unicode byl využit jednoduchý automatický překladač umístěný na webových stránkách [13], který umožňuje přeložit text oběma směry.

Program je navrhnut tak, že jej lze dodatečně a bez problému přeložit do jiného jazyka. Nejjednodušší postup k lokalizaci programu, je vytvoření kopie souboru z jakékoliv stávající lokalizace, přeložení do požadovaného jazyka spolu se zvolením vhodného názvu souboru (viz kapitola 4.3.4). Potom je ještě upravit menu pro výběr jazyka (viz kapitola 4.4) a program může plnohodnotně pracovat v nově lokalizovaném jazyce.

4 Realizace programu

Aby všechny třídy nebyly nashromážděny v jednom balíčku (adresáři), je program, kvůli přehlednosti rozdělen do tří balíčků. Názvy balíčků jsou odvozeny podle svého účelu. Jedná se o balíčky: `Main`, `Soucastky` a `NastaveniGUI`. Všechny tyto balíčky jsou umístěny v kořenovém balíčku pojmenovaném `src` (z angl. source). Obsah balíčku je na obrázku obr.4.1.



Obr.4.1: Obsah balíčku src

Editor je vytvořen pro verzi aplikačního rozhranní (API) 1.4.2. V případě použití programu s nižšími verzemi API proto není zaručena jeho zpětná kompatibilita. Program existuje ve dvou spustitelných formách. Primárně jako aplet a sekundárně jako samospustitelný `jar` soubor. Aplet lze spustit pouze z webového prohlížeče a zkompilovaný soubor nese název `EditorApplet.jar`. Samospustitelný soubor zase lze spustit pouze v operačním systému a nese název `PSpiceEditor.jar`. Oba soubory mají stejný obsah, s rozdílem ve spouštěcí (zavádějící) třídě, na kterou ukazuje vygenerovaný `jar`. Soubor `EditorApplet.jar` ukazuje na třídu `EditorLauncher` a `PSpiceEditor.jar` ukazuje na třídu `src.Main.Editor`, která jako jediná obsahuje statickou metodu `main(String [] arg)`. Samospustitelný soubor je výhodné spouštět s parametrem, který určí počáteční velikost vyrovnávací paměti a sníží se tak počáteční prodleva při inicializaci první komponenty umístěné na plátno. Spuštění programu přímo z příkazové řádky pak může vypadat následovně : `java –jar –Xms128m – Xmx256m PSpiceEditor.jar`. Kde jako výchozí velikost alokované paměti je nastavena na 128MB a maximální velikost využitelné paměti pak 256MB.

4.1 Třída EditorLauncher

Třída EditorLauncher je jediná třída, která se nachází v žádném balíčku, ale nachází se přímo v kořenovém balíčku celého projektu. Nejedná se o nic jiného než o spouštěcí aplet. Aplet se umístí do HTML kódu, který se pak spustí v okně internetového prohlížeče. Umístění apletu v HTML kódu je na obr.4.2.

```
<applet code="EditorLauncher.class"
  width=200
  height=50
  codebase="."
  archive="EditorApplet.jar"
  alt="Nemáte nainstalovanou nebo povolenou Javu ve Vašem prohlížeči"
</applet>
```

Obr.4.2: Applet umístěný v HTML kódu

Parametry `width` a `height` udávají velikost apletu na webové stránce (v tomto případě to je velikost spouštěcího tlačítka). Parametr `archive` udává spouštěcí třídu. Přepínač `alt` je alternativou zobrazení v případě, že na počítači kde se pokoušíme aplet spustit není nainstalována Java.

Třída EditorLauncher je odvozená od třídy `javax.swing.JApplet`. Vytvořením instance této třídy vznikne prázdné okno apletu. Do tohoto okna je možné vkládat ostatní grafické komponenty jako panely, tlačítka, textová pole, menu apod. Od svého vzniku Java nabízí knihovny pro tvorbu GUI. Jsou to knihovny AWT (Abstract Windowing Toolkit) a její novější verzi JFC (Java Foundation Classes). Novější verze poskytuje více grafických komponent, jejich propracovanější správu a hlavně možnost měnit nastavení vzhledu (např. podle použitého operačního systému), čehož je zde náležitě využito. Více je o tvorbě a umísťování komponent uvedeno zde [11].

Aplet se od ostatních spouštěcích tříd liší v tom, že nemají statickou metodu `main(String arg)`. Nicméně musí překrývat metody `init()`, `destroy()`, `start()` a `stop()`. Metoda `init()` se načte ihned při načtení apletu internetovým prohlížečem. Zpravidla se stará o inicializaci komponent. Metoda `destroy()` se spustí v případě, že je aplet ukončen, nebo dojde-li k zavření okna ve kterém aplet běží. Metoda se volá těsně před jeho ukončením. Metoda `start()` je nejprve volána po skončení metody `init()` , ale také vždy když je aplet pozastaven a po sléze znovu spuštěn. Tato situace nastane například, když je v prohlížeči aktivováno jiné okno než s apletem. V tom případě je pak volána metoda `stop()`, která pozastaví činnost apletu. Když se uživatel později vrátí zpět k činnosti apletu je znovu volána metoda `start()`. Metody `start()` a `stop()` ve třídě `EditorLauncher` není potřeba překrývat, protože se vůbec ve funkci editoru neuplatní.

Třída `EditorLauncher` pracuje pouze se čtyřmi atributy, viz obr.4.3.

```
public class EditorLauncher extends JApplet implements ActionListener
{
  public JFrame okno;
  public Editor ed;
  public JButton launchButton;
  public EditorMenu editorMenu;
...
```

Obr.4.3: Atributy třídy EditorLauncher

Atribut okno je odkaz na okno, do kterého se umístí editor, `ed` je odkaz na editor, `launchButton` je spouštěcí tlačítko editoru a objekt typu `EditorMenu` je třída realizující horní menu okna (viz kapitola 4.4). Inicializační metoda apletu je na obr.4.4.

```
public void init()
{
  try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
  }
  catch(Exception e) {
    e.printStackTrace();
  }

  ed = Editor.getEditor();
  ed.loadPictures();
  ed.colorAllComponents();
  editorMenu = new EditorMenu();

  // Změní reakci při ukončení programu

editorMenu.itemExit.removeActionListener(editorMenu.itemExit.getActionListeners()[0]);
editorMenu.itemExit.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
          okno.dispose();}
  });
...
```

Obr.4.4: Metoda init() třídy EditorLauncher

Metoda začíná nastavením vzhledu programu. Aby vzhled grafických komponent odpovídal vzhledu stejného jako je použitý operační systém na kterém je program spouštěn. Nastavení vzhledu má na starosti třída `javax.swing.UIManager`. Její metoda `setLookAndFeel(String className)` nastaví vzhled komponent. Jako parametr je zde předána návratová hodnota metody zjišťující informaci aktuálním operačním systému (`UIManager.getSystemLookAndFeelClassName()`). Získá se a uloží odkaz na editor, a provede potřebnou inicializaci obrázků (viz kapitola 4.3). V okamžiku vytvoření instance editorMenu tato instance obsahuje metodu ošetřující reakci ukončení editoru z menu (viz obr.4.32, položka itemExit). Obsahem metody je příkaz `System.exit(0),` který ihned po zavolání ukončí program. Toho je využito v případě, když se spouští editor jako samostatná aplikace. Pokud je aplikace spuštěna jako apletu je tato reakce na událost překryta událostí, která neukončí celý program, ale zavře pouze okno s editorem. Nejprve se tedy odstraní stávající posluchač událostí a pak se vytvoří nový posluchač, který jako reakci na událost uzavře okno příkazem `okno.dispose()`. Metoda init ještě pokračuje inicializací tlačítka `launchbutton`.

Metoda `createWindow()` má na starosti vytvoření a inicializaci okna editoru. Výpis je na obr.4.5.

```
public void createWindow()
{
  okno = new JFrame();
  Container cp = okno.getContentPane();
  cp.add(ed.getLeftPanel(), BorderLayout.CENTER);
  cp.add(ed.getRightPanel(), BorderLayout.EAST);

  GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
  Rectangle rec = ge.getMaximumWindowBounds();

  okno.setJMenuBar(editorMenu.getMenuBar());
  okno.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
  okno.setSize(new Dimension(rec.width, rec.height));
  okno.setTitle(ed.TITLE);
  okno.setVisible(true);
}
```

Obr.4.5: Výpis metody createWindow()třídy EditorLauncher

Vytvoří se nová instance typu JFrame. Z odkazu na editor se využívá pouze levý a pravý panel editoru, nikoliv celé okno. Na střed okna se umístí levý panel editoru a na pravou část pravý panel editoru. Jako typ rozmístění komponent v okně je zde použit layout manažer typu `java.awt.BorderLayout`. Layout manažer je objekt který udává jakým stylem jsou rozloženy komponenty uvnitř jiných komponent. O rozmístění komponent a práce s layout manažery je pojedáno zde [15]. Velikost okna se nastaví podle velikosti rozlišení na počítači kde je editor spuštěn. Odkaz na grafické prostředí systému se získá metodou `getLocalGraphicsEnvironment()` třídy `java.awt.GraphicsEnvironment`. Hodnoty maximální šířky a výšky rozlišení vrací metoda `getMaximumWindowBounds()` podle které se velikost okna nastaví. Ještě následuje ošetření události při zavření okna editoru stisknutím křížku v pravém horním rohu okna. Událost je ekvivalentní se stejnou událostí která se děje při ukončení editoru z menu. Okno se zavře, ale běh programu se nezastaví.

Ještě zbývá popsat překrytou metodu `destroy()` a ošetřující událost na stisk tlačítka `launchButton`, tj. metodu `actionPerformed(ActionEvent evt)` která je na obr.4.6.

```
public void actionPerformed(ActionEvent evt) {
  if (okno == null) {
   createWindow();
   launchButton.setText("CLOSE EDITOR");
   okno.addWindowListener(new WindowAdapter() {
      public void windowClosed(WindowEvent evt) {
        launchButton.setText("LAUNCH EDITOR");
        launchButton.setEnabled(true);
        ed.clearCanvas();
        okno = null;
      }
     });
  }
  else {
   launchButton.setEnabled(false);
   okno.setVisible(false);
   okno.dispose();
   okno = null;
  }
}
```

Obr.4.6: Výpis metody actionPerformed(ActionEvent evt) reagující na stisk tlačítka launchButton třídy EditorLauncher

Nejprve se po stisku tlačítka zjistí, jestli už okno s editorem existuje, pokud ne, toto okno se vytvoří. Změní se nápis na tlačítku informující o tom, že jeho opětovným stiskem se editor zavře. Oknu se přiřadí posluchač událostí reagující na zavření okna. Jako reakce na jeho zavření okna se změní nápis na tlačítku zpět do počátečního stavu informujícího o možnosti znovu spustit editor. Současně se vymaže obsah plátna a vymaže se odkaz na okno.

V případě že se zmáčkne tlačítko `launchButton` a odkaz na okno už v tu dobu existuje (viz druhá část podmínky) je okno uzavřeno a odkaz na něj se vymaže.

Poslední metodou, která zbývá popsat je metoda `destroy()` která se vykoná při uzavření okna prohlížeče. Výpis je na obr.4.7.

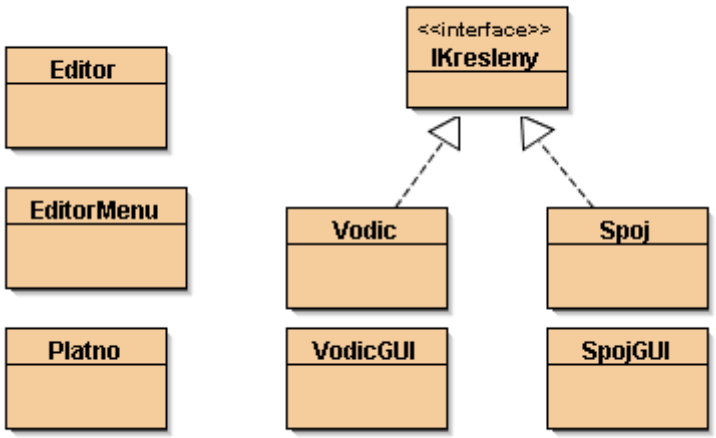
```
public void destroy()
{
  if (okno != null) {
   ed.destroySimulationProfileWindow();
   ed.dispose();
   okno.dispose();
  }
}
```

Obr.4.7: Výpis metody destroy() třídy EditorLauncher

podmíněný příkaz se vykoná, jestliže okno s editorem existuje. Nejprve se zavolá metoda na odstranění okna simulačního profilu. Pak se zavře okno s odkazem na editor následně okno s editorem ve kterém je editor umístěn.

4.2 Balíček Main

Balíček `Main` sdružuje čtyři třídy, které jsou klíčové pro chod programu, od toho plyne název balíčku. Jsou to třídy `Editor`, `EditorMenu`, `Platno` a rozhranní `IKresleny`. Jelikož třídy `Vodic` a `Spoj` není možné zařadit mezi součástky a je zbytečné kvůli tomu zakládat nový balíček, jsou umístěny zde. Zde je alespoň vidět grafické znázornění provázanosti tříd. Třídy `Vodic` i `spoj` implementují rozhranní `IKresleny` viz implementační šipky na obr.4.8.



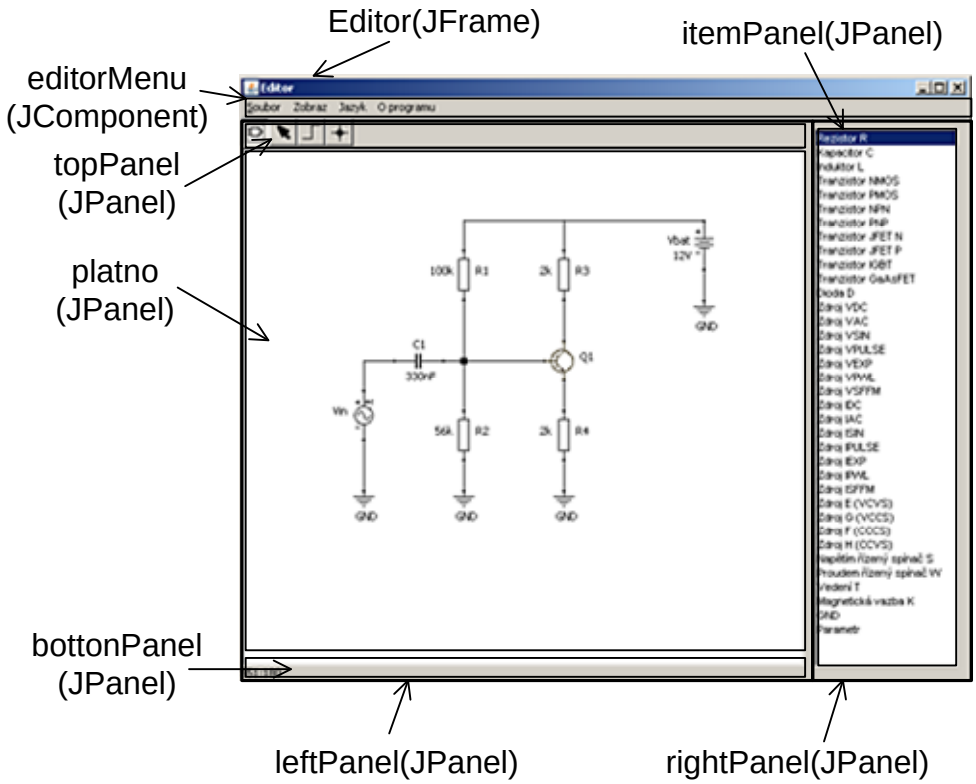
Obr.4.8: Obsah balíčku Main

4.3 Třída Editor

Třída Editor je odvozená od třídy `javax.swing.JFrame`. Toho je právě využito při tvorbě samospustitelné verze editoru. Není totiž nutné se odkazovat na jinou, dodatečně vytvořenou instanci třídy `JFrame` na kterou by se editor odkazoval. Editor se tak odkazuje sám na sebe.

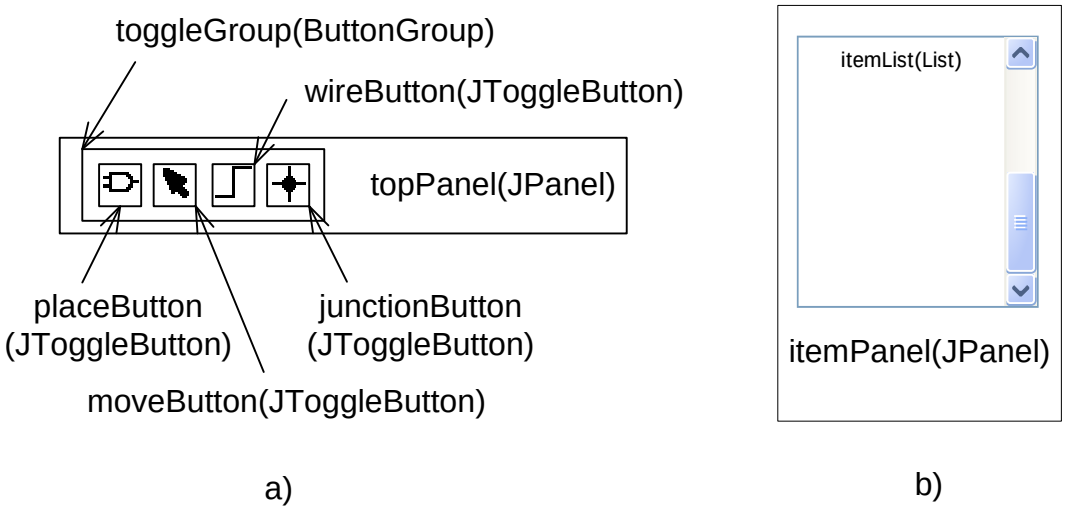
4.3.1 Okno editoru

Okno editoru je rozděleno na několik částí. Jednotlivé komponenty jsou uspořádány tak, aby ve výsledku utvořili celek tak jak je vidět na obr.3.2. Obsah okna je rozdělen do 2 hlavních panelů viz obr.4.9. U každé komponenty je nejdříve uveden její identifikátor, v závorce je pak uvedeno o kterou instanci třídy se jedná. S oběma panely také pracuje aplet (viz použité metody `getLeftPanel()` a `getRightPanel()` na obr.4.5.)



Obr.4.9: Rozdělení okna na dva hlavní panely

Levý panel obsahuje další 3 komponenty. Typ rozložení těchto komponent je daný manažerem třídy `javax.swing.BoxLayout`. Komponenty `topPanel` a `bottomPanel` mají přesně definovanou svojí velikost, zatím co komponenta `platno` nikoliv. To způsobí roztažení kreslícího plátna přes zbytek volné plochy obrazovky. Spodní panel obsahuje pouze odkaz na instanci třídy `javax.swing.JLabel` s názvem `statusText`. Tento popisek ukazuje souřadnice kurzoru a informuje o aktuálním dění při kreslení na plátně, jako je označení, přesunu součástky atd. Inicializaci levého panelu má na starosti metoda `initLeftPanel()`. Obdobně, inicializaci pravého panelu má na starosti metoda `initRightPanel()`. Pravý panel obsahuje pouze jeden panel. Tím je `itemPanel`. Tento panel obsahuje seznam součástek nazvaný `itemList` třídy `java.awt.List`, viz obr.4.10 varianta b).



Obr.4.10: Rozkreslení obsahu panelů a) topPanel b) itemPanel

Ovládací tlačítka na přepínání stavu kurzoru jsou umístěny ve skupině `toggleGroup` třídy `javax.swing.ButtonGroup`. Tím je zajištěno, že v daný okamžik může být aktivní jenom jedno tlačítko. Inicializaci tlačítkového menu zajišťuje metoda `createButtonMenu()`. Částečný výpis je na obr.4.11.

```
public void createButtonMenu()
 {
  toggleGroup = new ButtonGroup();
  placeButton = new JToggleButton(iconPlace, true);
 ...

  Dimension dim = new Dimension(iconPlace.getIconWidth()+7,
                 iconPlace.getIconHeight()+7);
  placeButton.setPreferredSize(dim);
  placeButton.setMaximumSize(dim);
 ...

  toggleGroup.add(placeButton);
 ...

  topPanel.add(placeButton);
 ...
  setCreateComponent();
 }
```

Obr.4.11: Částečný výpis metody createButtonMenu() třídy Editor

Nejprve se vytvoří instance skupiny tlačítek s názvem `toggleGroup` do které se postupně vloží všechna tlačítka. Ve výpisu programu to je vždy ukázáno na tlačítku `placeButton`, pro zbylá tlačítka jsou příkazy stejné. Instance třídy `java.awt.Dimension` definuje rozměr tlačítka. Ikona obrázku je o rozměru 21x21 pixelů. K této hodnotě je připočten okraj 7 pixelů. Poslední metoda `setCreateComponent()` nastaví při inicializaci režim kurzoru do stavu vytvoření součástky.

Inicializaci `itemListu` má na starosti metoda `initItemList()`. Částečný výpis metody je na obr.4.12.

```
public void initItemList()
 {
   if(itemList != null)
   {
     itemList.removeAll();
     itemList.removeItemListener(itemList.getItemListeners()[0]);
     itemPanel.removeAll();
     itemList = null;
   }

   itemList = new List(38, false);

   // prida seznam soucastek
   itemList.add(resBundle.getString("Resistor"));
 ...

   itemList.addItemListener(new ItemListener() {
     public void itemStateChanged(ItemEvent e) {
       setCreateComponent();}
   });

   itemPanel.add(itemList);
   // nastavi kurzor v Listu na Rezistor
   itemList.select(0);
   itemPanel.revalidate();
 }
```

Obr.4.12: Částečný výpis metody initItemList() třídy Editor

Před samotnou inicializací seznamu součástek se nejprve zjistí jestli už seznam existuje. Pokud ano, odstraní se posluchač událostí reagující na výběr součástky v seznamu, vymaže se seznam součástek a odstraní se z `itemPanelu`. Pak následuje vytvoření nové instance seznamu o 38 řádcích. Seznam se postupně naplní novými názvy součástek. Jména součástek se načítají z lokalizačního souboru, příkazem `resBundle.getString(String str)` kde parametrem je název součástky. Tímto způsobem jsou všechny součástky přidány do seznamu. K seznamu se připojí posluchač události, který reaguje na výběr v seznamu. Jakmile uživatel klikne do seznamu součástek, stav kurzoru se automaticky přepne na režim vytvoření součástky. Příkaz `itemList.select(0)` nastaví ukazatel výběru řádku seznamu na první pozici.

4.3.2 Práce s obrázky

Pro práci s obrázky je zde vytvořeno několik metod. Jakýkoli grafický objekt, se kterým se na kreslícím plátně pracuje je označen červenou barvou. V případě součástek, editor obsahuje odkazy na tyto instance a to jak neoznačených obrázků, tak i na instance označených obrázků součástek. Názvy označených součástek mají příponu „Sel“ od slova selected. Např. odkaz na obrázek rezistoru má název rezistor a jeho označená varianta pak `rezistorSel` apod. S popisem metod vztahujících se k práci s obrázky začnu jednoduchou metodou, která vrací odkaz na relativní umístění třídy v jar archívu. Tím je metoda `loadPath(String path)`. Tělo metody je na obr.4.13.

```
private URL loadPath(String path)
{
  return getEditor().getClass().getClassLoader().getResource(path);
}
```

Obr.4.13: Výpis metody loadPath(String path) třídy Editor

Tato metoda se používá při načtení obrázků součástek, viz metoda `loadPictures()` jejíž částečný výpis je na obr.4.14.

```
public void loadPictures()
{
  // obrazky stavu kurzoru
  iconPlace = new ImageIcon(loadPath("src/img/place.png"));
  ...

  // obrazky soucastek
  rezistor = new ImageIcon(loadPath("src/img/rezistor.png")).getImage();
  ...
 }
```

Obr.4.14: Částečný výpis metody loadPictures() třídy Editor

Odkazy na jednotlivé instance obrázků jsou typu `ImageIcon`. Třída `ImageIcon` je specifická tím že obrázky načítá do vyrovnávací paměti. S obrázky se v této paměti pracuje rychleji. Toho se hlavně využívá u obrázků větších datových objemů (cca nad 500kB) v našem případě, kdy má jeden obrázek velikost v řádu stovek bajtů je efekt zanedbatelný. Nyní když už jsou načteny obrázky neoznačených součástek, je ještě potřeba z nich udělat obrázky označené. Kdyby byly součástky vykreslovány vektorově pak by ve změně barvy nebyl problém, jenom by se změnila barva kreslených tvarů, součástka by se na plátno vykreslila znovu v jiné barvě. Tak se tomu děje při kreslení propojovacích čar a kreslení spojů. Ty jsou vykreslovány vektorově. U bitmapových obrázků je tomu jinak. Obrázky součástek jsou transparentní a jednobarevné. Obsahují tedy buď informaci o stupni průhlednosti (tzv. alfa kanál) nebo černou barvu. Pro obarvení obrázku součástky pak stačí najít všechny pixely, které nejsou transparentní a změnit jejich barvu, což vykonává statická metoda `repaintImage(Image obrazek)`. Výpis metody je na obr.4.15.

```
private static Image repaintImage(Image obrazek)
{
  int red = 0xffff0000; // cervena barva v HEX
  BufferedImage     image     =     new     BufferedImage(obrazek.getWidth(null),
obrazek.getHeight(null),BufferedImage.TYPE_INT_ARGB);

  Graphics2D g = image.createGraphics();  // vytvori Graphics2D object
  g.drawImage(obrazek, null, null);  // Kresli data z Obrazku do promenne image

   //cyklus ktery probehne celym obrazkem a zmeni barvu vsech
   //nenulovych pixelu na cervenou barvu

  for(int x = 0; x < image.getWidth(); x++){
      for(int y = 0; y < image.getHeight(); y++){
        int pixel = image.getRGB(x, y);
        if(pixel != 0){
          image.setRGB(x,y,red);
         }
      }
  }
  return image;
}
```

Obr.4.15: Výpis statické metody repaintImage(Image obrazek) třídy Editor

Aby bylo možné pracovat s obrázkem na úrovni pixelů, je nutné jej převést na objekt typu `java.awt.image.BufferedImage`, který umí pracovat s obrázkem na úrovni pixelů. Nejprve se tedy vytvoří kopie obrázku, který je předán jako parametr a odkaz se uloží do proměnné `image`. Lokální proměnná s názvem `g` vytvoří instanci objektu `java.awt.Graphics`, která je schopná pracovat s obrázkem (v našem případě do něj zapisovat). Cyklus projede všechny body obrázku a pokud narazí na pixely, které nejsou transparentní, obarví je na červenou barvu. Na hexadecimální hodnotu červené barvy odkazuje lokální primitivní datový typ `red`.

Tak jako jsou hromadně vytvořeny odkazy na obrázky neobarvených součástek, stejným způsobem jsou vytvořeny odkazy na všechny obarvené součástky. Metoda má název `colorAllComponents()` a je na obr.4.16.

```
public void colorAllComponents()
{
  rezistorSel = repaintImage(rezistor);
 ...
}
```

Obr.4.16: Částečný výpis metody colorAllComponents() třídy Editor

4.3.3 Singleton

Téměř všechny třídy nebo jejich instance použité v programu se nějakým způsobem odkazují na instanci editoru. Chceme-li zabezpečit aby se tyto třídy nebo jejich instance odkazovali pouze na jeden a ten stejný editor, musí být splněna podmínka, která zamezí vytváření dalších instancí editoru. Na to je vhodné aplikovat návrhový vzor, označený v anglické literatuře jako singleton. Jako singleton je pojmenován statický soukromý atribut

třídy, který odkazuje na tuto instanci. Vytváření dalších tříd je zamezeno tak, že třída `Editor` má soukromý konstruktor, tedy nikdo jiný než třída sama nemůže danou instanci vytvořit. Přístupová metoda `getEditor()` pak vždy vrátí odkaz na stejnou instanci. Výpis metody je na obr.4.17.

```
public static Editor getEditor()
{
  if(singleton == null)
    singleton = new Editor();

  return singleton;
}
```

Obr.4.17: Výpis metody přístupové getEditor() třídy Editor

Jestliže ještě není vytvořena instance editoru, vytvoří jej. Pokud už instance existuje, metoda vrací odkaz na tuto instanci. Výpis konstruktoru třídy `Editor` je na obr.4.18.

```
private Editor()
{
  initBundle();
  try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
  }
  catch(Exception e) {
      e.printStackTrace();
  }
}
```

Obr.4.18: Konstruktor třídy Editor

Metoda `initBundle()` inicializuje nastavení lokalizace programu (viz obr.4.19). Třída `UIManager` a její metoda `setLookAndFeel(String className)` nastaví vzhled programu podle operačního systému tak jako tomu je u třídy `EditorLauncher`.

4.3.4 Lokalizace

V kapitole 3.6 věnované lokalizaci programu, je uvedeno jak je lokalizace programu vytvořena. Nyní se podíváme jak je implementována v programu a jaké používá metody. Jelikož je program umístěn na webových stránkách [18] které jsou pouze v angličtině, přišlo mi vhodné aby se program spustil také v anglickém jazyce, s tím že uživatel si může za chodu přepnout program do libovolného jazyka. Odkaz na výchozí jazykové nastavení je uloženo ve veřejném statickém atributu `currentLang`. Výchozí jazyk je angličtina, hodnota atributu je `currentLang = "en"`. Hodnota atributu se změní vždy, když se změní jazykové nastavení. Změna jazykového nastavení se provádí z horního menu editoru. Horní menu editoru má na starosti třída `EditorMenu`. Její instance, která nese název `editorMenu`, k tomuto atributu přistupuje a mění jeho hodnotu, podle aktuální volby jazyka viz metoda `setLanguage(String lang)` na obr.4.33.

První metoda která inicializuje jazykové nastavení programu je metoda `initBundle()`, která je na obr.4.19.

```
public void initBundle()
{
  this.setLocale(new Locale(currentLang));
  resBundle = ResourceBundle.getBundle("src.locale.Resources", this.getLocale());
}
```

Obr.4.19: Výpis metody initBundle() třídy Editor

Nejprve se nastaví lokalizační prostředí podle atributu `currentLang`. Pak se vytvoří odkaz na tzv. lokalizační balíček se kterým pracují všechny třídy nebo instance které se odkazují na nějaký lokalizovaný název nebo popisek. Jako parametr metody `getBundle(String baseName, Locale locale)` je předána nejprve cesta k adresáři kde jsou textové soubory s lokalizací a jako druhý parametr se předá aktuální jazykové nastavení programu (které bylo změněno na anglickou verzi). Adresář s lokalizovanými soubory je v podadresáři kořenového balíčku (`src`). Aby program věděl který z těchto souborů je určen pro daný jazyk, musí tyto soubory mít svůj přesný název. Anglická verze překladu musí mít název `Resources_en.properties`, česká verze pak `Resources_cs.properties`. Název souboru se liší v posledních dvou písmenech udávající lokalizační jazyk. Seznam všech lokalizačních zkratek je uveden zde [10].

Druhá metoda která zpracovává požadavek na změnu jazykového nastavení programu je metoda `localize()`. Její výpis je na obr.4.20.

```
public void localize()
 {
  initBundle();
  placeButton.setToolTipText(resBundle.getString("placeButtonToolTipText"));
  moveButton.setToolTipText(resBundle.getString("moveButtonToolTipText"));
  wireButton.setToolTipText(resBundle.getString("wireButtonToolTipText"));
  junctionButton.setToolTipText(resBundle.getString("junctionButtonToolTipText"));

   initItemList();

   if(simProfile != null)
    simProfile.localize();
 }
```

Obr.4.20: Výpis metody localize() třídy Editor

Nejdříve se znovu nastaví lokalizační prostředí. Nastaví se textový popisek tlačítek měnících stav kurzoru a znovu se inicializuje seznam součástek. Jestliže už bylo aktivováno okno se simulačním profilem, dojde také k jeho změně lokalizace. Tato metoda se spouští při změně jazykového nastavení programu, viz metoda `executeLocalization()` obr.4.36.

4.3.5 Netlist

Základní postup při generování netlistu je vysvětlen v kapitole 3.5. Nyní se podíváme jak se je generování netlistu implementováno v programu.

První metoda, která se zavolá při generování netlistu je metoda `initNodes()`. Tím dojde k počáteční inicializaci atributu součástek ukazující na číslo uzlu do kterého je součástka zapojena. Výpis metody je na obr.4.21.

```
private void initNodes()
{
   Iterator iter = componentList.iterator();
   Soucastka souc;
   Vodic vod;
   Messenger mess;
   IKresleny kres;

  // inicializace atributu ukazujici na to do kterého uzlu je součástka zapojena
  while(iter.hasNext())
  {
    kres = (IKresleny)iter.next();

    // inicializuje uzly soucastek
     if(kres instanceof Soucastka)
     {
       souc = (Soucastka)kres;
       for(int i=0; i<souc.uzly.length; i++)
       {
         souc.uzly[i] = -1;
       }
     }

     // separuje vodice, a vytvori zabali je do objektu messenger
     if(kres instanceof Vodic)
     {
       vod = (Vodic)kres;
       mess = new Messenger(vod, false);
       wiresList.add(mess);
     }
   }
}
```

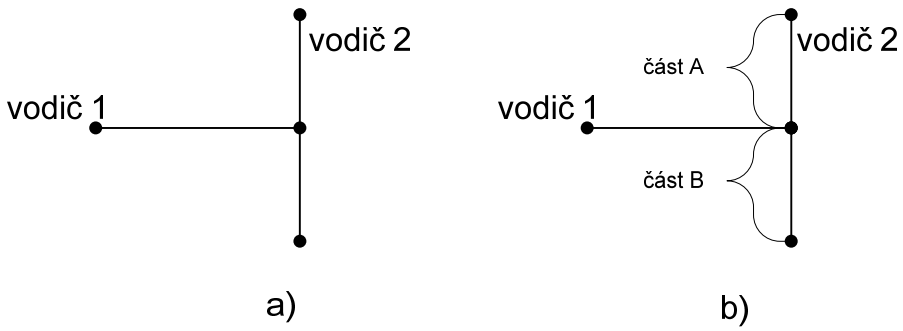
Obr.4.21: Výpis metody initNodes() třídy Editor

Lokální proměnná `iter` se odkazuje na iterátor kolekce obsahující objekty které jsou umístěny na kreslícím plátně. Cyklus `while` prochází kolekci součástek dokud neprojde celý její obsah. První podmíněný příkaz se provede, pokud je kreslený objekt součástka. Uvnitř `for` cyklu se inicializuje číselná hodnota uzlů. Hodnota -1 znamená, že součástka ještě není připojena do uzlu. Další podmíněný příkaz separuje vodiče, „zabalí“ je do instance třídy `Messenger` a přidá do kolekce pojmenované `wireList`. Kolekce `wireList` uchovává odkazy na všechny vodiče umístěné na kreslícím plátně, „zabalené“ do instancí messengeru. Třída `Messenger` je vnitřní třídou třídy `Editor`. Slouží jako pomocná třída. Instance třídy `Messenger` obsahuje dva veřejné atributy. Prvním atributem je odkaz na vodič, který se předá při vytváření instance a druhá je primitivní datový typ `boolean`, s názvem `marker`. `Marker` slouží pro označení „cesty“ při procházení cyklem, který má za úkol zjistit propojení mezi součástkami (viz. druhá část metody `DFS(Point startPoint, int nodeNumber)` na obr.4.28) Výchozí hodnota markeru je `false`, po označení se změní na `true`. Výpis vnitřní třídy `Messenger` je na obr.4.22.

```
private static class Messenger
{
  public boolean marker;
  public Vodic vodic;

  public Messenger(Vodic v, boolean marker)
  {
    this.marker = marker;
    this.vodic = v;
  }
}
```

Obr.4.22: Výpis vnitřní pomocné třídy Messenger třídy Editor

Ještě se vrátím ke kapitole 3.5. Na obr.3.6 jsou možné varianty propojení součástek a vodičů. Varianty *a, c* a *f* vpravo vyjadřují variantu spojení, kdy se přípojný bod jednoho vodiče, nebo součástky dotýká jiného vodiče. Nezasahuje ovšem do jeho přípojného bodu, ale zasahuje dovnitř vodiče. Algoritmus generování netlistu postupuje od jednoho přípojného bodu k dalšímu. Není tedy možné detekovat takovýto způsob spojení. Řešení tohoto problému spočívá v analýze přípojných bodů součástky, nebo vodiče a jejich případného zasahování dovnitř dalšího vodiče. Pokud tomu tak je, zasažený vodič je rozdělen na dva další (resp. vytvoří dva nové). Výsledek je na obr.4.23.



Obr.4.23: Rozdělení vodičů a) před rozdělením b) po rozdělení

Funkci rozdělení vodičů mají na starosti metody `splitWires` a `splitAtPoint(Point p)`. Výpis metody `splitAtPoint(Point p)` je na obr.4.24, metoda `splitWires()` je na obr.4.25.

```
private void splitAtPoint(Point p)
{
  Messenger mess;
  int size = wiresList.size();

  for(int i = 0 ; i < size; i++)
  {
    mess = (Messenger)wiresList.get(i);

// testuje jestli bod p nalezi do vodice ale zaroven neni soucasti pripojnych bodu

    if(mess.vodic.contains(p))
    {
      // rozdeli vodice na 2
      Vodic partA, partB;
      partA = new Vodic(mess.vodic.getJunctionPoint()[0], p);
      partB = new Vodic(p, mess.vodic.getJunctionPoint()[1]);
```

```
      // prida je do seznamu
      wiresList.add(new Messenger(partA, false));
      wiresList.add(new Messenger(partB, false));
    }
  }
}
```

Obr.4.24: Výpis metody splitAtPoint(Point p) třídy Editor

Vstupním parametrem metody je bod, podle kterého dojde k rozdělení vodiče v případě, že se tento bod nachází uvnitř. Cyklus `for` projede kolekci `wireList`. Pokud je bod ve vodiči obsažen, vytvoří se dva nové vodiče se souřadnicemi přípojných bodů dané původním vodičem a přípojným bodem uvnitř vodiče. Nové vodiče se přidají zpět do kolekce `wireList`.

```
private void splitWires()
{
  Messenger mess;
  IKresleny kres;
  Soucastka souc;
  Spoj spoj;
  int vodiceSize = wiresList.size();
  int komponentySize = componentList.size();

  // zjisti jestli konce vodicu zasahuji do jinych vodicu
  for(int i = 0; i < vodiceSize ; i++)
  {
    mess = (Messenger)wiresList.get(i);
    splitAtPoint(mess.vodic.getJunctionPoint()[0]);
    splitAtPoint(mess.vodic.getJunctionPoint()[1]);
  }

// zjisti jestli konec nejake soucastky zasahuje do vodice nebo jestli se jedna
// o krizeni spoju

  for(int j = 0; j < komponentySize; j++)
  {
    kres = (IKresleny)componentList.get(j);
    if(kres instanceof Soucastka)
    {
      souc = (Soucastka)kres;
      for(int k = 0; k < souc.getJunctionPoint().length ; k++)
      {
        splitAtPoint(souc.getJunctionPoint()[k]);
      }
    }
    if(kres instanceof Spoj)
    {
      spoj = (Spoj)kres;
      splitAtPoint(spoj.getJunctionPoint()[0]);
    }
  }
}
```

Obr.4.25: Výpis metody splitWires() třídy Editor

První `for` cyklus projde kolekci `wireList`. U každého vodiče v kolekci se testují oba dva jeho přípojné body, zda-li nezasahují dovnitř jiných vodičů a to metodou `splitAtPoint(Point p)`. Druhý cyklus `for` prochází kolekci všech kreslených komponent. První podmíněný příkaz testuje přítomnost přípojných bodů součástek, druhý příkaz testuje přítomnost spoje.

Nyní můžeme přikročit k popisu metody `assignNodes()`. Její úkol je přiřazení čísla uzlů součástkám. Výpis metody je na obr.4.26.

```
public void assignNodes()
{
  wiresList = new ArrayList<Messenger>();
  componentList = platno.getComponentList();
  nodeCounter = 0; // vynuluje citac uzlu
  initNodes();
  splitWires();

  Iterator iter = componentList.iterator();
  IKresleny kres;
  Soucastka souc;

  while(iter.hasNext())
  {
    kres = (IKresleny)iter.next();
    if(kres instanceof Soucastka)
    {
      souc = (Soucastka)kres;
      for(int i=0; i<souc.uzly.length; i++) // projde vsechny uzly soucastky
      {
        if(souc.uzly[i] == -1) // narazi na uzel ktery jeste nebyl oznacen
        {
          nodeCounter++;
          souc.uzly[i] = nodeCounter;
          DFS(souc.getJunctionPoint()[i], nodeCounter);
        }
      }
    }
  }

  // prejmenuje uzly spojene na zem
  iter = componentList.iterator();
  while(iter.hasNext())
  {
    kres = (IKresleny)iter.next();
    if(kres instanceof Gnd)   // jedna se o GND, prejmenuje vodice
    {
    // prejmenuje vsechny soucastky pripojene na zem na nulu
     souc = (Soucastka)kres;
     renameNode(souc.uzly[0], 0);
    }
  }
}
```

Obr.4.26: Výpis metody assignNodes() třídy Editor

Lokální proměnná `nodeCounter` odkazuje na číslo aktuálního uzlu. Při každém nalezení nového uzlu se jeho hodnota inkrementuje. Cyklus `while` projede obsah kolekce kreslených komponent. Jestliže narazí na součástku, cyklus `for` projede všechny její čísla uzlů. Podmíněný příkaz zjistí zda-li už je součástka připojena do uzlu (resp. její odkaz na číslo uzlu je různý od implicitní hodnoty, tj. -1). Pokud součástka není ještě připojena do uzlu, inkrementuje se čítač uzlů a na přípojný bod součástky se aplikuje algoritmus prohledávání do hloubky (viz metoda `DFS(Point startPoint, int nodeNumber)` na obr.4.28). Když jsou všem přípojným bodům součástek přiřazeny čísla uzlů, projede se kolekce kreslených komponent ještě jednou a všem součástkám, které jsou spojeny na zem (resp. součástkou `GND`) se přepíše

číslo uzlu na nulu metodou `renameNode(int oldNode, int newNode)`. Výpis metody je na obr.4.27.

```
private void renameNode(int oldNode, int newNode)
{
  Iterator iter = componentList.iterator();
  Soucastka s;

   while(iter.hasNext())
  {
     IKresleny kres = (IKresleny)iter.next();

     if(kres instanceof Soucastka)
     {
       s = (Soucastka)kres;
       for(int i=0; i<s.uzly.length; i++)
       {
         if(s.uzly[i] == oldNode)
          s.uzly[i] = newNode;
       }
     }
  }
}
```

Obr.4.27: Výpis metody renameNode(int oldNode, int newNode) třídy Editor

Cyklus `while` projde seznam kreslených komponent a přepíše všechny hodnoty čísel uzlů shodných s hodnotou předanou jako vstupní parametr `oldNode` hodnotou `newNode`.

Ještě zbývá uvést výpis metody implementující algoritmus procházení do hloubky. Výpis metody `DFS(Point startPoint, int nodeNumber)` je na obr.4.28.

```
private void DFS(Point startPoint, int nodeNumber)
{
   Iterator iterSeznam = componentList.iterator();
   Iterator iterVodice = wiresList.iterator();
   Point P1, P2;
   IKresleny kres;
   Soucastka souc;
   Vodic vod;
   Messenger mess;

   // prohleda seznam soucastek a podiva se jestli nejaka soucastka sdili
   // stejny pripojny bod, pokud ano priradi ji stejne cislo uzlu

   while(iterSeznam.hasNext())
   {
    kres = (IKresleny)iterSeznam.next();
    if(kres instanceof Soucastka)
    {
      souc = (Soucastka)kres;
      for(int i=0; i<souc.getJunctionPoint().length; i++)
      {
        if(souc.getJunctionPoint()[i].equals(startPoint))
        {
            souc.uzly[i] = nodeNumber;
        }
      }
    }
   }

   // hledame sousedy
   while(iterVodice.hasNext())
```

```
   {
     mess = (Messenger)iterVodice.next();

      if(mess.marker == false)
      {

         P1 = mess.vodic.getJunctionPoint()[0];
         P2 = mess.vodic.getJunctionPoint()[1];

         if(P1.equals(startPoint))
         {
           mess.marker = true;
           DFS(P2, nodeNumber);
         }

         if(P2.equals(startPoint))
         {
          mess.marker = true;
          DFS(P1, nodeNumber);
         }
       }
   }
}
```

Obr.4.28: Výpis metody DFS(Point startPoint, int nodeNumber) třídy Editor

Tato metoda má dva vstupní parametry. Prvním parametrem je počáteční bod, od kterého probíhá hledání s názvem `startPoint`. Ve druhém parametru je uložen odkaz na číslo uzlu. Cyklus `while` prohledá seznam kreslených komponent. Pokud narazí na součástku, cyklus `for` prohledá všechny její přípojné body. Jestliže je má nějaký přípojný bod shodné souřadnice se souřadnicemi bodu, který byl předán jako parametr, znamená to, že součástka je s tímto bodem spojena. Přepíše se tedy její číslo uzlu, hodnotou předanou jako druhý parametr. Takto se otestují přípojné body všech součástek. Druhá část algoritmu vyhledává další přípojné body součástek, které jsou spojené propojovacími vodiči. Testuje se, zda-li jsou souřadnice bodu vstupního parametru shodné s počátečním nebo koncovým bodem nějakého vodiče z kolekce `wireList`. Pokud tomu tak je, dojde k označení vodiče příkazem `mess.marker = true` a rekurzivně se zavolá opět stejná metoda. Vstupním parametrem jsou souřadnice buďto počátečního, nebo koncového bodu vodiče. Pokud jsou souřadnice bodu předané jako parametr shodné se souřadnicemi počátečního bodu vodiče, pak rekurzivním volání metody se předají jako parametr souřadnice koncového bodu vodiče a naopak.

Metoda, která je spuštěna při požadavku na generování netlistu je pojmenována `showNetlist()` a její výpis je na obr.4.29.

```
public void showNetlist()
{
  createOutputWindow();
  assignNodes();
  printNetlist();
  outputWindow.setVisible(true);
}
```

Obr.4.29: Výpis metody showNetlist() třídy Editor

Výpis netlistu probíhá do samostatně vytvořeného okna metodou `createOutputWindow()`. Proběhne přiřazení čísel uzlů součástkám a výpis netlistu do okna metodou `printNetlist()`. Výpis metody je na obr.4.30.

```
public void printNetlist()
{
  int size = componentList.size();
  IKresleny kres;
  Soucastka souc;
  String model;
  Vector<String> modelList = new Vector<String>();

  for(int i=0; i<size; i++)
  {
    kres = (IKresleny)componentList.get(i);
    if((kres instanceof Soucastka) && !(kres instanceof Gnd))
    {
      souc = (Soucastka)kres;
      textArea.append(souc.getNetlistText()+"\n");
    }
  }

  textArea.append("*\n");

  // vypise modely u soucastek ktere jej obsahuji
  for(int i=0; i<size; i++)
  {
    kres = (IKresleny)componentList.get(i);
    if(kres instanceof Soucastka)
    {
      souc = (Soucastka)kres;
      if(souc.hasModel)
      {  // zkontroluje duplicitu modelu
        if (!modelList.contains(souc.extractModelName()))
        {
          modelList.add(souc.extractModelName());
          textArea.append(souc.getModelText());
          textArea.append("\n*\n");
        }
      }
    }
  }

  if(simProfile != null)
  {
    textArea.append("*" + resBundle.getString("analysisDirectives") + "\n");
    textArea.append(this.simProfile.getSettings());
  }
  textArea.append(".PROBE/CSDF\n*\n");
  textArea.append(".END\n");

  // vymaze odkazy
  wiresList = null;
  modelList = null;
}
```

Obr.4.30: Výpis metody printNetlist() třídy Editor

Počáteční `for` cyklus projde kolekci kreslených komponent a u všech součástek kromě součástky `GND` zavolá metodu `getNetlistText()`. Metoda vrací textový zápis součástky podle konvence PSpice. Další `for` cyklus projde seznam kreslených komponent a vypíše modely

součástek, které jej obsahují. Kolekce s názvem `modelList` v sobě uchovává názvy použitých modelů ve schématu. Podle referenční příručky PSpice není možné uvést definici modelu součástky o stejném názvu více než jednou. Z tohoto důvodu je zde podmíněný příkaz, který testuje duplicitu názvu modelu. Pokud je ve schématu více než jedna součástka se stejným názvem modelu, bere se pouze její první výskyt, ostatní se ignorují. V případě, že byl simulační profil definován, výpis nastavení se provede příkazem `simProfile.getSettings()`.

Když už je řeč o simulačním profilu, editor sám obsahuje odkaz na instanci simulačního profilu, která nese název `simProfile`. Metody, které mají na starosti práci se simulačním oknem se nazývají `createSimulationProfileWindow()`, `destroySimulationProfileWindow()` a `showProfile()`. Myslím, že už z názvů metod je zřejmý jejich účel a netřeba jej vysvětlovat.

Třída `Editor` ještě obsahuje ještě některé další metody a to např. metodu `createSelectedItem(int x, int y)` kterou využívá instance třídy `Platno`. Metoda se volá před tím, než se vybraná součástka umístí na plátno. Metoda vrací instanci typu `Soucastka` na souřadnicích předaných jako parametr. Částečný výpis metody je na obr.4.31.

```
public Soucastka createSelectedItem(int x, int y)
{
  int volba = itemList.getSelectedIndex();

  if (volba == 0)
    return new Rezistor(x, y);
  else if (volba == 1)
     return new Kapacitor(x, y);
 ...
}
```

Obr.4.31: Částečný výpis metody createSelectedItem(int x, int y) třídy Editor

Příkaz `itemList.getSelectedIndex()` vrátí číselnou hodnotu řádku seznamu, který byl stisknut. Každému řádku je přiřazena hodnota. Hodnoty začínají od nuly a jsou číslovány od shora seznamu součástek směrem dolu. Nulu má rezistor, jedničku kapacitor atd. Jako návratový typ se vrací přímo instance součástky vytvořená na souřadnicích x, y. Další metodou je například metoda `clearCanvas()` která vynuluje počítadla jednotlivých součástek a vymaže obsah plátna. Počítadla součástek se vynulují tak, že se u každé třídy součástek zavolá metoda `resetCounter()`. Uvnitř třídy `Editor` se nacházejí ještě další jednořádkové přístupové a nastavovací metody které jsou natolik triviální a jejich význam nemá smysl vysvětlovat. Jako např. metody `setStatusText()`, `getStatusText()` apod.

4.4 Třída EditorMenu

Třída `EditorMenu` je potomkem třídy `javax.swing.JComponent`. Instance této třídy je využívána třídou `Editor`. Jak vypadá hlavní menu je vidět na obr.3.2. Objekty jako jsou názvy jednotlivých sloupců menu jsou potomkem třídy `javax.swing.JMenu`. Jednotlivé položky těchto sloupců jsou zase potomkem třídy `javax.swing.JMenuItem`. Instance těchto tříd se

vytvářejí v konstruktoru třídy. Každé položce menu je přiřazena reakce na událost, která se provede při spuštění. Reakce na události jsou uvedeny uvnitř metody `init()`, která se volá na konci konstruktoru třídy. Částečný výpis této metody je na obr.4.32.

```
public void init()
{
  // nabidka itemExit
  itemExit = new JMenuItem("", KeyEvent.VK_K);
  itemExit.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
        System.exit(0);}
  });

  // nabidka nove schema
  itemNew = new JMenuItem("", KeyEvent.VK_N);
 ...
        editor.clearCanvas();}
  });

  // nabidka Uzly
  itemShowNodes = new JMenuItem("", KeyEvent.VK_U);
 ...
        editor.assignNodes();
        editor.platno.showNodeNumber();}
  });

  // nabidka netlist
  itemShowNetlist = new JMenuItem("", KeyEvent.VK_Z);
 ...
        editor.showNetlist();}
  });

  // nabidka nastaveni simulace
  itemSettings = new JMenuItem("", KeyEvent.VK_O);
 ...
          editor.showProfile();}
  });

  // nabidka jazyk
  menuLang_CS.addActionListener(new ActionListener() {
 ...
        setLanguage("cs");
      }
  });

  menuLang_EN.addActionListener(new ActionListener() {
 ...
         setLanguage("en");
       }
  });

  // nabidka itemAuthor
  itemAuthor = new JMenuItem(getTxtValue("menuAuthor"));
  itemAuthor.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         JOptionPane.showMessageDialog(null,
          getTxtValue("menuAuthor"),
          getTxtValue("menuAbout"),
          JOptionPane.INFORMATION_MESSAGE);
        }
  });
...
```

Obr.4.32: Částečný výpis metody init() třídy EditorMenu

Ve výpisu metody je uveden posluchač na daný typ události každé položky menu. Pro zjednodušení textu je úplný výpis metody přidání posluchače a události uveden pouze u položky `itemExit`, která ukončí činnost programu. Položka `itemNew` vytvoří nové schéma, resp. zavolá metodu `clearCanvas()` třídy `Editor` vymaže všechny kreslené objekty z kreslícího plátna. Položka `itemShowNodes` zavolá metodu `assignNodes()` (viz obr.4.26), která přiřadí čísla uzlů součástkám a zobrazí je na plátně metodou `showNodeNumber()` třídy `Platno`. Položky `menuLang_CS` a `menuLang_EN` změní jazykové nastavení programu zavoláním metody `setLanguage(String lang)`. Poslední položka `itemAuthor` zobrazí informační okno třídy `javax.swing.JOptionPane`.

Na obr.4.32 je jako událost na změnu volby jazyka volána metoda `setLanguage(String lang)`, která mění veřejný atribut třídy `Editor` odkazující se na aktuální jazykové nastavení programu. Výpis metody je na obr.4.33.

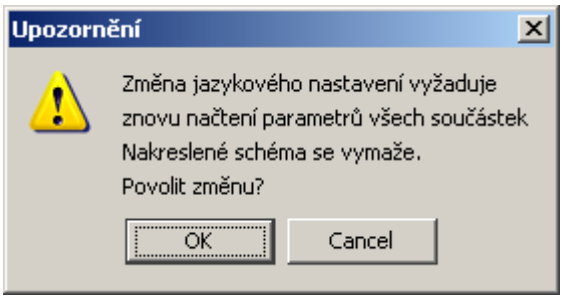
```
public void setLanguage(String lang)
{
   if(editor.currentLang != lang)
   {
     if(!editor.platno.isCanavasEmpty())
     {
       if(dialogConfirmed())
       {
        editor.currentLang = lang;
        executeLocalization();
       }
     }
     else
     {
      editor.currentLang = lang;
      executeLocalization();
     }
   }
}
```

Obr.4.33: Výpis metody setLanguage(String lang) třídy EditorMenu

První podmínka testuje, zda-li je volba jazykového nastavení zadaná jako vstupní parametr metody různá od aktuálně nastavené jazykové volby. Jestliže je volba různá zavolá se další podmínka, která testuje jestli je kreslící plátno prázdné. Jestliže je prázdné, otevře se okno s potvrzením o změně jazykové volby. S každou změnou jazyka se totiž musí znovu inicializovat všechny názvy a popisky součástek. Zobrazení potvrzovacího okna má na starosti metoda `dialogConfirmed()` jejíž návratový typ informuje o tom, jak se uživatel rozhodl. Výpis metody je na obr.4.34 a obrázek dialogového okna je na obr.4.35. V případě, že uživatel volbu potvrdí nebo je plátno prázdné, změní se atribut odkazující se na aktuální jazykové nastavení příkazem `editor.currentLang = lang` , který přepíše jeho hodnotu na hodnotu předanou jako parametr. A zavolá se metoda `executeLocalization()`, která provede lokalizaci programu.

```
private boolean dialogConfirmed()
 {
   boolean result = false;

     int i = JOptionPane.showConfirmDialog(editor,
            getTxtValue("warningDialog"),
            getTxtValue("warningDialogTitle"),
            JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.WARNING_MESSAGE);
     switch(i) {
       case JOptionPane.OK_OPTION:
           result = true;
           break;
       case JOptionPane.CANCEL_OPTION:
       case JOptionPane.CLOSED_OPTION:
           result = false;
           break;
     }

   setLangChoice();
   return result;
 }
```

Obr.4.34: Výpis metody `dialogConfirmed()` třídy EditorMenu



Obr.4.35: Dialogové okno o potvrzení změny jazykového nastavení

Metoda spuštějící lokalizační proces programu se nazývá `executeLocalization()`. Výpis metody je na obr.4.36.

```
public void executeLocalization()
 {
   editor.localize();
   setLangChoice();
   initMenuText();
   editor.clearCanvas();
 }
```

Obr.4.36: Výpis metody executeLocalization() třídy EditorMenu

Nejprve se zavolá metoda `localize()` třídy `Editor` (viz obr.4.20) nastaví se ukazatel položky v menu na právě zvolený jazyk metodou `setLangChoice()`, znovu se načtou lokalizované názvy jednotlivých položek menu metodou `initMenuText()` a pak se vymaže kreslící plátno voláním metody `clearCanvas()`.

Metoda která nastavující jako aktivní položku právě zvoleného jazyka se nazývá `setLangChoice()` a její výpis je na obr.4.37.

```
public void setLangChoice()
 {
   String lang = editor.currentLang;
   if(lang.equals("en"))
   {
     menuLang_EN.setSelected(true);
   }
   else
   {
     menuLang_CS.setSelected(true);
   }
 }
```

Obr.4.37: Výpis metody setLangChoice() třídy EditorMenu

Lokální proměnná `lang` odkazuje na jazykové nastavení editoru, podmíněný příkaz testuje jaký jazyk je právě nastaven a podle toho nastaví aktivní položku menu.

Pro přeložení programu do dalšího jazyka stačí přidat lokalizovaný textový soubor do složky mezi ostatní soubory s lokalizací. Dále pak přidat novou položku menu (obdobně jako jsou přidány položky s názvem `menuLang_CS`, `menuLang_EN`) a pozměnit metodu `setLangChoice()`, resp. přidat novou jazykovou volbu.

4.5 Rozhranní IKresleny

Všechny kreslené komponenty, které mají být nakresleny na plátně, musejí implementovat rozhranní `IKresleny`. Rozhranní deklaruje všechny potřebné metody potřebné k nakreslení komponenty. Implementují ho tedy všechny součástky, včetně tříd `Vodic` a `Spoj`. Rozhranní je součástí hlavního balíčku `main`. Výpis rozhranní je na obr.4.38.

```
public interface IKresleny
{
   public String getComponentName();
   public void paint(Graphics g);
   public void repaint(int x, int y, Graphics g);
   public Rectangle getShape();
   public Dimension getDimension();
   public int getX();
   public int getY();
   public void settings();
   public Point offset();
   public void rotateRight();
   public boolean changeStatus();
   public Point[] getJunctionPoint();
}
```

Obr.4.38: Výpis rozhranní IKresleny

Požadavky na kreslené komponenty jsou následující: komponenta se musí umět identifikovat svým názvem (metoda `getComponentName()`), umět se nakreslit (metody `paint(Graphics g)` a `repaint(int x, int y, Graphics g)`, vrátit informaci o rozměrech (metody `getShape()` a `getDimension()`), vrátit aktuální souřadnice (`getX()`, `getY()`), zobrazit

okno s nastavením (`settings()`), otočit se (`rotateRight()`), vrátit pole přípojných bodů (`getJunctionPoint()`). Ještě zde zbývají metody `offset()` a `changeStatus()`. První z nich vrací upravené souřadnice kreslené komponenty. Souřadnice se upravují v případě, kdy při tažení komponenty na plátně dojde k umístění součástky mimo kreslící plátno resp. mimo okno editoru. Souřadnice komponenty se upraví tak aby komponenta zůstala vždy na plátně. Druhá metoda se používá při práci s komponentou. Jejím voláním se komponenta „označí“ resp. změní svoji barvu na červenou.

4.6 Třída Platno

Třída `Platno` je odvozena od třídy `JPanel`. Implementuje rozhranní posluchačů událostí `MouseInputListener a KeyListener`. První rozhranní deklaruje metody reagující na události myši jako je stisk tlačítka nebo pohyb. Třída `Platno` překrývá pouze metody `mousePressed (MouseEvent e), mouseClicked (MouseEvent e)` a `mouseReleased(MouseEvent e)`. Metody `mouseEntered(MouseEvent e)` a `mouseExited(MouseEvent e)` zůstávají prázdné, nepoužívají se. Druhé rozhranní deklaruje metody reagující na události kláves. Jsou to metody, `keyPressed(KeyEvent e), keyReleased(KeyEvent e)` a `keyTyped (KeyEvent e)`. Z toho se využívá pouze metoda `keyReleased(KeyEvent e)`.

Vše co je na plátně nakresleno se ukládá do kolekce třídy `java.util.ArrayList` s názvem `componentList`. Kreslené komponenty se do kolekce ukládají v pořadí, ve kterém jsou vytvořeny. Obsah kolekce se na plátno nevykresluje postupně, ale jednorázově. Vykreslování neprobíhá přímo na instanci plátna, ale na pomocnou proměnnou nazvanou `drawingPane`. Pomocná proměnná je instancí vnořené třídy `DrawingPane`, která je taktéž odvozená od třídy `JPanel`. K vykreslování se ještě používá objekt třídy `java.awt.Image` s názvem `backBuffer`. Vykreslování komponent probíhá nejprve na „pozadí“ panelu a teprve když jsou všechny komponenty z kolekce vykresleny, vygeneruje se a zobrazí se obrázek (`backBuffer`) na plátně. Výpis vnořené třídy `DrawingPane` je na obr.4.39.

```
public class DrawingPane extends JPanel {

    public void paint(Graphics g){
      super.paint(g);
      refresh(g);
      update(g);
    }

    public void refresh(Graphics g)
    {
      Iterator iter = componentList.iterator();
      while(iter.hasNext())
      {
        IKresleny kres = (IKresleny)iter.next();
        kres.paint(g);
      }
    }

    public void update(Graphics g)
    {
      //prekresli pozadi jako obrazek
      g.drawImage( backbuffer, 0, 0, this );
    }

}
```

Obr.4.39: Výpis vnořené třídy DrawingPane

Povinně překrytá metoda `paint(Graphics g)` se zavolá poprvé při vytvoření instance a vždy když je se registruje změna kreslených komponent na plátně. Tato metoda se, ale volá nepřímo, pomocí zděděné metody `repaint()` z rodičovské třídy `java.awt.Component`. Jako parametr metody `paint` je předán objekt typu `java.awt.Graphics`, který funguje jako „kreslítko“. Objekt obsahuje metody umožňující prácí s grafickými objekty a jejich vykreslování. Metoda `refresh(Graphics g)` vykreslí postupně všechny kreslené komponenty na pozadí. Metodou `update(Graphics g)` dojde k vykreslení obrázku (`backBuffer`) na plátno. Důvod proč se kreslené komponenty nevykreslují postupně na plátno, ale najednou jako obrázek je ten, že při postupném vykreslování by docházelo k nepříjemnému „blikání“ při vykreslování. V odborné literatuře se pro potlačení blikání při vykreslování používá název tzv. double buffering.

Počáteční velikost plátna se závisí na velikosti rozlišení obrazovky. V případě, že je potřeba vytvořit schéma větších rozměrů než je standardní rozměr plátna, je možné zvětšit rozměry plátna. A to tak že stačí umístit kreslenou komponentu mimo plátno, směrem buď dolů nebo vpravo. Tím se automaticky vytvoří rolovací lišta v požadovaném směru. Funkce automatického zobrazení rolovací lišty je zprostředkována atributem instance s názvem `scrollPane` třídy `javax.swing.JScrollPane` do kterého je `drawingPane` umístěn. Metoda která pak realizuje funkci změny rozměru plátna se nazývá `resizeCanavas(IKresleny kres)`. Výpis metody je na obr.4.40.

```
private boolean resizeCanavas(IKresleny kres)
{
    // atribut ktery udava jestli je potreba zmenit velikost platna, nebo ne
    boolean changed = false;

    // souradnice a rozmer soucastky
    int x = kres.getX();
    int y = kres.getY();
    int sirkaSoucastky = kres.getShape().width;
    int vyskaSoucastky = kres.getShape().height;

   // Rozsireni platna v pripade ze se soucastka nevejde na platno

    int this_width = (x + sirkaSoucastky + 10); // 10pix je rezerva
    if (this_width > variableArea.width) {
      variableArea.width = this_width;
      changed=true;
    }

    int this_height = (y + vyskaSoucastky + 10);
    if (this_height > variableArea.height) {
      variableArea.height = this_height;
      changed=true;
    }

    // jestlize je soucastka vetsi tzn (changed == true) aktualizuje platno
    if (changed){
      drawingPane.setPreferredSize(variableArea);
      drawingPane.revalidate();
    }
  return changed;
}
```

Obr.4.40: Výpis metody resizeCanavas(IKresleny kres) třídy Platno

Jako parametr metody je zde předána kreslená komponenta. Nejprve se zjistí souřadnice umístěné součástky a její rozměry. Tyto rozměry se porovnají s aktuálními rozměry plátna a pokud se zjistí že kreslená komponenta by vybočovala z plátna, rozměr plátna se zvětší. Aktuální rozměry plátna jsou uloženy v proměnné `variableArea`. Rozměr plátna při vytvoření editoru je uložen v atributu s názvem `originalArea`. Odkaz na původní velikost plátna se využívá v případě vymazání jeho obsahu metodou `clearCanvas()`, viz obr.4.41.

```
public void clearCanvas()
{
  componentList.clear();
  variableArea = new Dimension(originalArea);
  drawingPane.setPreferredSize(variableArea);
  drawingPane.revalidate();
  drawingPane.repaint();
}
```

Obr.4.41: Výpis metody clearCanvas() třídy Platno

Nejprve se vymaže kolekce kreslených komponent. Odkaz na velikost plátna se přepíše původní hodnotou. Velikost plátna se změní na původní rozměr.

4.6.1 Reakce na události myši

Metody reagující na události myši jsou poměrně rozsáhlé a z toho důvodu jsou uvedeny v příloze A.

První metoda, která se volá při stisknutí tlačítka myši z implementovaného rozhranní `MouseInputListener` je metoda `mousePressed()`. Nejprve se zjistí jaké tlačítko myši bylo stisknuto. Jestliže bylo stisknuto levé tlačítko myši, zjistí se jaký je nastavený aktuální stav kurzoru. Podle toho jaký je nastaven režim kurzoru se provede požadovaná akce. Při aktivním režimu vytvoření součástky se kreslená komponenta umístí na souřadnicích kurzoru metodou `createSelectedItem(int x, int y)`. Režim vytvoření spoje vytvoří instanci spoje taktéž na souřadnicích kurzoru. Při vytváření součástky parametr se nejprve zkontroluje, jestli už taková součástka v kolekci existuje, pokud ne, součástka vytvoří. V případě součástky parametr se může vytvořit pouze jediná instance. Je to z toho důvodu, aby se v netlistu neobjevilo více součástek nazvaných parametr (viz kapitola 4.20). V případě kreslení vodičů se najednou vytvoří vodiče dva. Jako první se vytvoří vodorovný vodič, jako druhý se vytvoří vodič svislý. V případě, že je kurzor nastaven do režimu přesunu komponenty, zavolá se metoda `getKresleny(int x, int y)`, která vrací odkaz na instanci objektu nacházejícího se „pod“ kurzorem. Metoda projede celou kolekci kreslených komponent a zjistí, jestli aktuální souřadnice kurzoru zasahují do aktivní oblasti nějaké komponenty. Pokud ano, nalezená komponenta se vrátí jako návratová hodnota. Výpis metody je na obr.4.42.

```
private IKresleny getKresleny(int x, int y)
{
  Iterator iter = componentList.iterator();
  while(iter.hasNext())
  {
    IKresleny kres = (IKresleny)iter.next();
    if (kres.getShape().contains(x, y))
     return kres;
  }
   return null;
}
```

Obr.4.42: Výpis metody getKreleny(int x, int y) třídy Platno

Nalezená komponenta se pak buď označí, nebo odznačí metodou s názvem `updateLastComponent`. Na konci metody `mousePressed()` se ještě zjišťuje varianta, zda kromě levého tlačítka myši bylo stisknuto tlačítko pravé. Pokud ano, je kreslená komponenta otočena vpravo (metodou `rotateRight()`).

Druhá metoda, která se volá v případě pohybu kurzoru myši se současným držením tlačítka myši. Je to metoda `mouseDragged()`. Nejdříve se zjišťuje jaké je současně drženo tlačítko myši. Pokud je to levé tlačítko, zjišťuje se, jestli při předcházející metodě (`mousePressed()`) byl kurzor nad nějakou komponentou. Jestliže tomu tak je, aktuální souřadnice komponenty se při tažení společně mění se souřadnicemi kurzoru. V případě

kreslení vodičů se při pohybu kurzoru mění jejich délka podle toho, ve kterém směru se kurzor pohybuje.

Metoda `mouseReleased()` se volá při uvolnění tlačítka myši. Opět se zjišťuje, jaké tlačítko bylo uvolněno a jestli přitom bylo pracováno s nějakou kreslenou komponentou. Pokud ano, kreslená komponenta se umístí na plátno a přichytí k tzv. mřížce metodou `snapToGrid(int z)`. Jestliže bylo pracováno s vodiči, zjistí se, jakou délku tyto vodiče mají. Často se totiž stává, že je potřeba kreslit propojovací vodič pouze v jednom směru, tedy většinou druhý vodič má nulovou délku. V tom případě se vodič, který má nulovou délku odstraní z kolekce.

Metoda `mouseClicked()` je volána při dokonaném stisku tlačítka myši. Tedy hned po metodě `mouseReleased()`. Jedná se tak o událost na vykonaný stisk tlačítka. V našem případě se obsah metody soustředí na dvojité stisknutí levého tlačítka myši. Opět se za pomocí souřadnic kurzoru se testuje, zda se kurzor nachází nad nějakou kreslenou komponentou. V případě že nachází nad nějakou kreslenou komponentou z kolekce, aktivuje její okno s nastavením parametrů (metodou `settings()`).

4.6.2 Reakce na události stisknutí klávesy

Každý stisk klávesy způsobí zavolání metody `keyPressed(KeyEvent e)`. Celkem se rozlišuje události na stisk devíti kláves. Pro změnu stavu kurzoru to jsou klávesy: *P* pro vytvoření komponenty, *S* pro označení komponenty, *W* pro kreslení vodičů a *J* pro vytvoření spoje. Jako klávesové zkratky pro rychlé volání funkcí z horního menu to jsou klávesy: *F7* která ukáže čísla uzlů, *F8* vygeneruje netlist a *F9* zobrazí okno s nastavením simulačního profilu. Ještě jsou zde uvedeny reakce na klávesy *Esc* která odznačí zvolenou komponentu a *Del* která označnou komponentu vymaže z plátna (resp. z kolekce).

4.6.3 Zobrazení čísel uzlů

K zobrazení čísel uzlů je určena metoda `showNodeNumber()`. Metoda projede celou kolekci, přitom pracuje pouze se součástkami. Na souřadnicích přípojných bodů zobrazí čísla uzlů, do kterých náleží. Čísla uzlů se zobrazují v modrém rámečku. Rozměr rámečku se mění současně s počtem míst uvedeným v uzlu. Rámeček se kreslí metodou `paintBox(Graphics g, int x, int y, int width, int height)`. Jako vstupní parametry se předávají „kreslítko“ souřadnice kde se má rámeček umístit, jeho šířka a výška. Jak vypadá takové zobrazení uzlů je na obr.3.3.

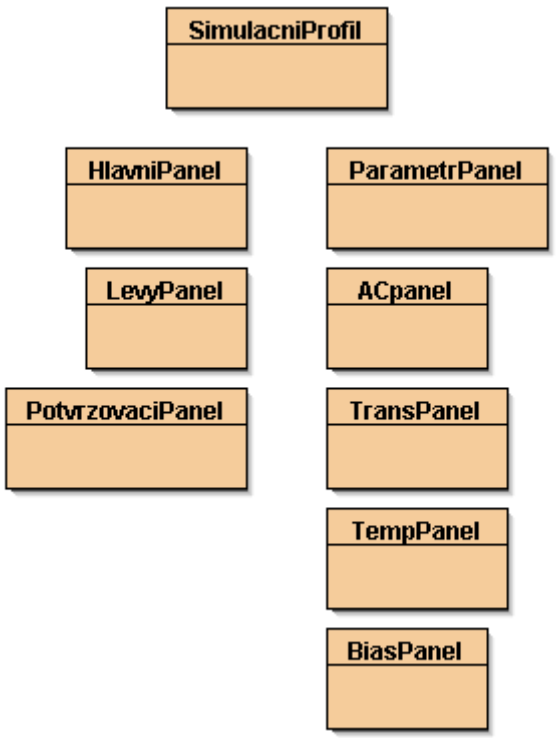
4.6.4 Přichycení k mřížce

Na konci kapitoly 3.2. je zmínka o rastru pozadí ke kterému jsou kreslené komponenty přichyceny. Implementace přichycení komponent k mřížce je jednoduchá, před jejím umístěním na plátno se souřadnice zaokrouhlí na celočíselný násobek desíti. Metoda vracející

celočíselný násobek se nazývá `snapToGrid(int z)`, kde vstupním parametrem je číslo, které má být zaokrouhleno.

4.7 Balíček NastaveniGUI

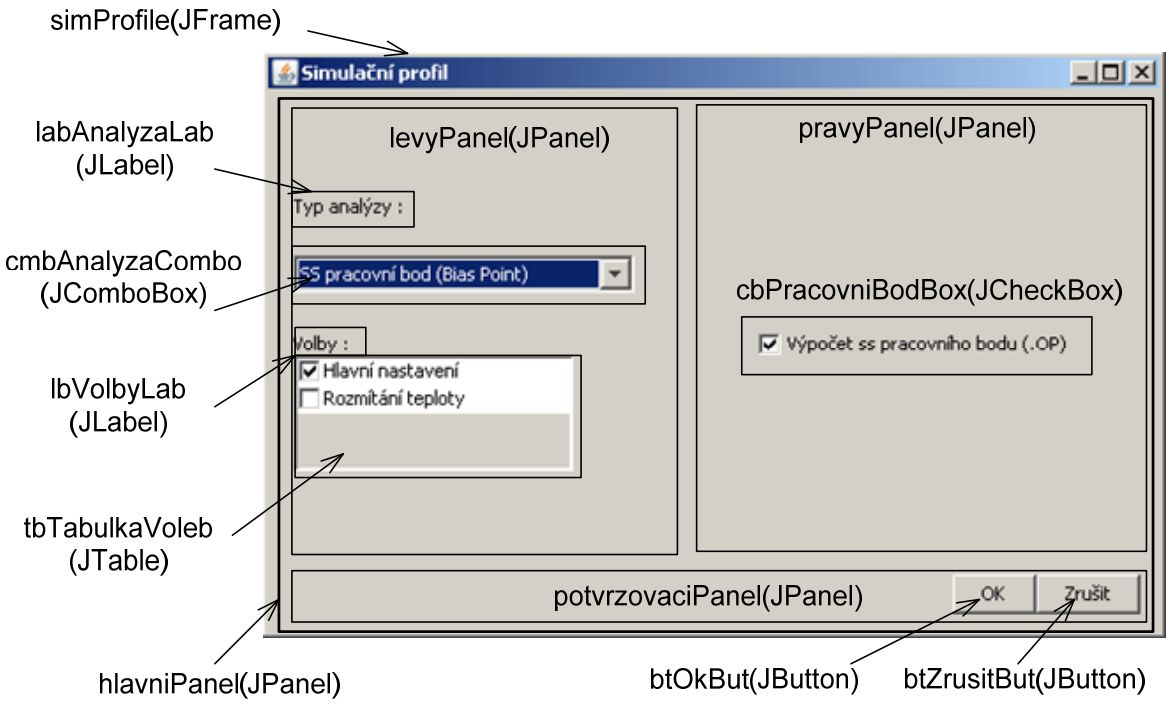
Okno simulačního profilu je na obr.3.4. K vytvoření a sestavení komponent okna byl pro usnadnění práce použit program určený přímo pro tvorbu GUI v Javě s názvem Jvider v.1.8 [14]. Zmínění program pracuje s layout managerem typu `GridBagLayout`. Zmíněný layout manager se používá hlavně u složitějších, programově generovaných rozmístění komponent, nepoužívá se pro ruční zadávání z důvodu složitosti a špatné čitelnosti. Okno simulačního profilu se zkládá z několika částí. Tyto části jsou rozděleny do jednotlivých tříd a všechno co se týká nastavení okna nastavení simulačního profilu je umístno do samostatného balíčku s názvem `nastaveniGUI`. Obsah balíčku `nastaveniGUI` je na obr.4.43.



Obr.4.43: Obsah balíčku nastaveniGUI

Třída `SimuacniProfil` používá všechny ostatní třídy v balíčku. Hlavní panel obsahuje levý panel a pravý panel. V závislosti na volbě analýzy (`cmbAnalyzaCombo`) se změní obsah pravého panelu a současně se změní výběr v tabulce dodatečného nastavení simulace (`tbTabulkaVoleb`). Nastavení simulačního profilu se pak uloží stiskem tlačítka `butOkBut`, nebo zruší tlačítkem `btZrusitBut`. Ke každému typu analýzy je předem definované dodatečné nastavení simulace. U stejnosměrné analýzy je na výběr primární a sekundární rozmítání, rozmítání parametru a rozmítání teploty. Střídavá a analýza v časové oblasti nabízí kromě

hlavního nastavení, také rozmítání parametru a teploty. Analýza pracovního bodu nabízí navíc pouze rozmítání teploty. Rozvržení a nastavení voleb jednotlivých komponent je převzato ze simulačního programu PSpice OrCAD Capture. Jak jsou komponenty rozloženy je vidět na obr.4.44. Názvy komponent odpovídají názvům uvedeným v programovém kódu. Jako dodatek upřesním, že všechny třídy obsahují metody pro změnu lokalizace programu s názvem `localize()`. Všechny třídy nesoucí názvy jednotlivých typů panelů pak mají metody vracející textovou hodnotu právě nastaveného profilu s názvem `getSettings()`.



Obr.4.44: Rozložení komponent okna simulačního profilu

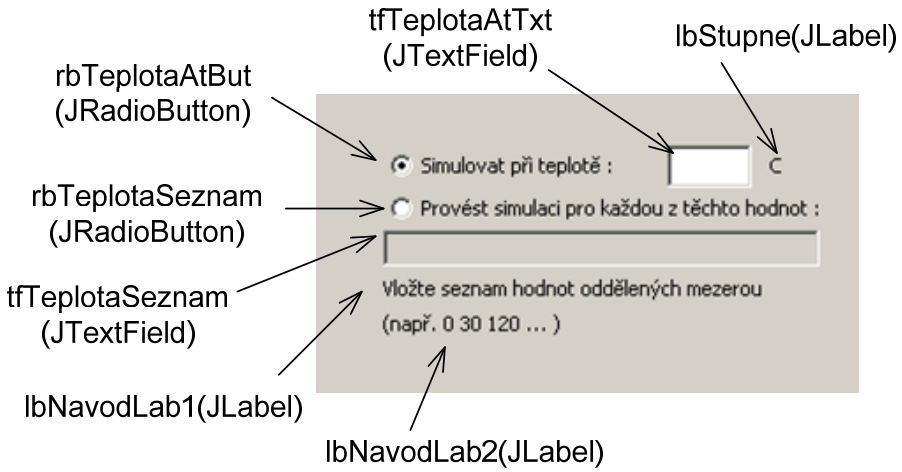
4.7.1 Třída BiasPanel

Analýza výpočtu pracovního bodu je nejjednodušším typem analýzy. Analýza provádí výpočet malosignálových parametrů nelineárních prvků. Zobrazení `BiasPanelu` je rovněž na obr.4.44. Obsahuje pouze jedno zaškrtávací políčko typu `javax.swing.JCheckBox`. Zaškrtnutím políčka se potvrdí volba analýzy. Metoda `getSettings()` pak vrátí textový řetězec `".OP"`.

4.7.2 Třída TempPanel

Rozmítání teploty se využívá u všech typů analýz. Teplotní panel je kombinací dvou přepínacích tlačítek, třech popisků a dvou textových polí. Výřez okna panelu je na obr.4.45. V závislosti na tom jaký typ teplotního rozmítání je vyžadován, podle toho jsou aktivní textová pole pro vložení textu. Buď lze zadat jednu teplotu, pro kterou bude simulace probíhat, pak je aktivní textové pole `tfTeplotaAtTxt`, nebo zadat seznam hodnot, pak je

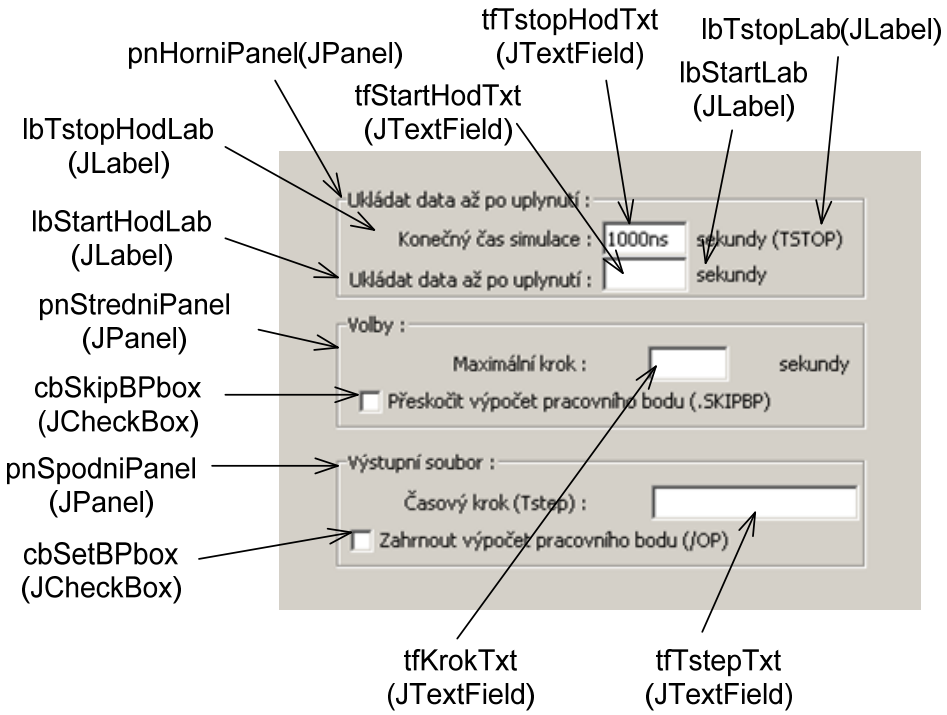
aktivní pole `tfTeplotaSeznam`. Metoda `getSettings()` pak vrací textový řetězec `".TEMP "` plus obsah jednoho nebo druhého textového pole.



Obr.4.45: Částečný výřez okna panelu třídy TempPanel

4.7.3 Třída TransPanel

Základem panelu analýzy v časové oblasti jsou čtyři textové pole a dvě zaškrtávací políčka. Komponenty jsou rozdělené do tří panelů. Částečný výřez okna je na obr.4.46.



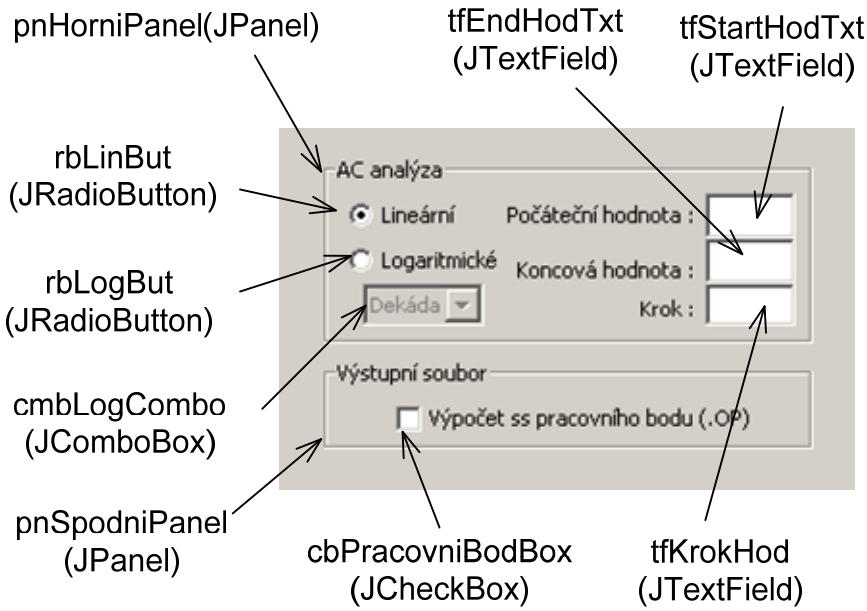
Obr.4.46: Částečný výřez okna panelu TransPanel

Horní panel obsahuje nastavení konce nebo počátku simulace, koncová hodnota je standardně nastavena na 1000ms, počáteční hodnota se musí zvolit. Nastavení maximální

délky kroku při simulaci je na prostředním panelu. Součástí středního panelu je i možnost volby pro potlačení výpočtu pracovního bodu (`cbSkipBPbox`). Naopak zaškrtávací políčko `cbSetBPbox` ve spodním panelu, aktivuje detailní výpočet pracovního bodu. Textové políčko `tfTstepTxt` slouží k vyplnění hodnoty udávající rozmezí při vykreslování bodů. Metoda `getSettings()` textový řetězec v pořadí znaků podle referenční příručky PSpice. Řetězec začíná jako vždy výpisem informujícím o jaký typu simulace, tedy `".TRANS "`, ke kterému jsou postupně přidány hodnoty z textových polí (pokud nejsou ponechány prázdné), nebo podle zaškrtnutých voleb. Jako první se vypisuje volba `cbSetBPbox`, pokud je zaškrtnuta tak se přidá parametr `"/OP"`. Druhý parametr je obsah textového pole `tfTstepTxt`. Pokud je textové pole prázdné nahradí se jeho obsah nulou. Dalšími parametry jsou hodnoty `tfTstopHodTxt, tfStartHodTxt, tfKrokTxt` a nakonec volba `cbSkipBPbox`.

4.7.4 Třída ACpanel

V nastavovacím okně střídavé analýzy se definuje počáteční kmitočet, koncový kmitočet a krok. Pro vložení těchto hodnot jsou zde textová pole `tfStartHodTxt, tfEndHodTxt` a `tfKrokHodTxt` .Dalším parametrem analýzy je nastavení typu rozmítání. Na výběr je lineární a logaritmické rozmítání. Volbu typu rozmítání zajišťují tlačítka `rbLinBut` a `rbLogBut`. V případě logaritmického rozmítání je zde ještě na výběr počet bodů na dekádu resp. oktávu. Částečný výřez okna je na obr.4.47.

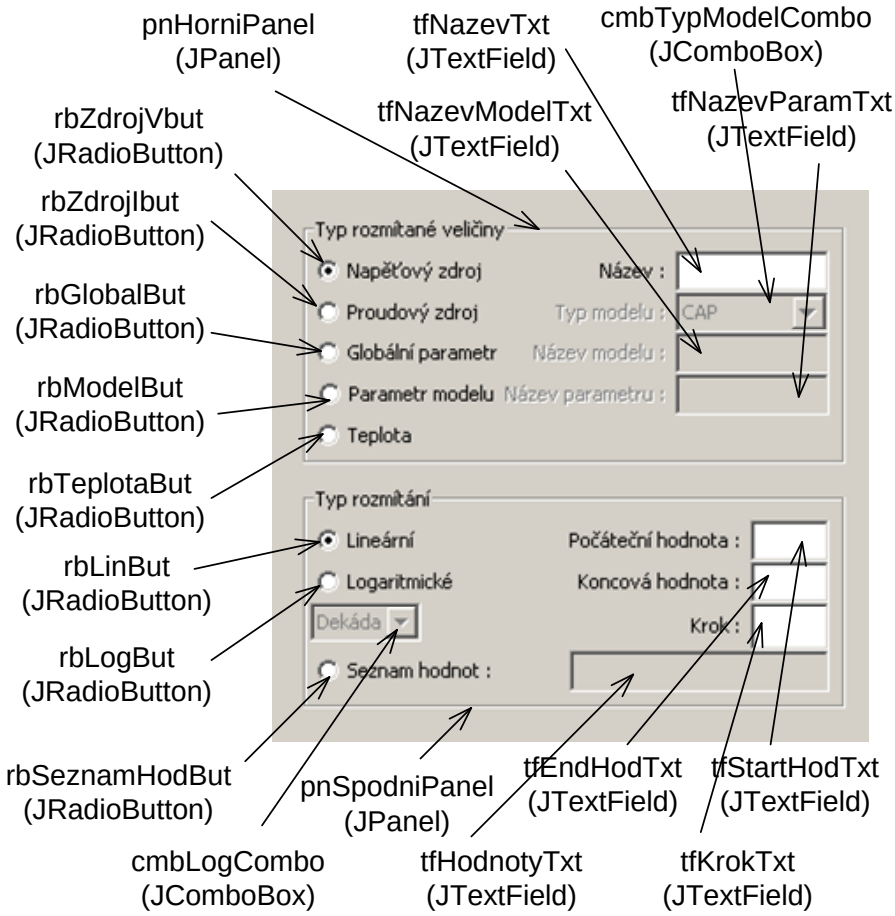


Obr.4.47: Částečný výřez okna panelu ACpanel

4.7.5 Třída ParametrPanel

Panel pro rozmítání parametru je základem pro stejnosměrnou analýzu. Stejnosměrná analýza využívá tento panel hned třikrát. Používá se jak u primárního a sekundárního

rozmítání, tak i u rozmítání parametru. Jako rozmítání parametru je tento panel použit také ve střídavé analýze a u rozmítání v časové oblasti. Zobrazení panelu je na obr.4.48.



Obr.4.48: Částečný výřez okna panelu ParametrPanel

Panel pojmenovaný jako `pnHorniPanel` slouží k volbě rozmítané veličiny. Vybrat lze pouze vždy jen jeden typ rozmítané veličiny. Ke každé volbě náleží jiný počet zpřístupněných textových polí. U napěťového nebo proudového zdroje je aktivní pouze textové pole `tfNazevTxt`, u globálního parametru pouze pole `tfNazevParamTxt` atd. Spodní panel nabízí volbu, jak bude veličina rozmítána. Na výběr je lineární, logaritmické rozmítání, nebo zadání seznamu hodnot. Generování výsledného textového řetězce je to složitější než u předešlých panelů. V první řadě je nutné rozlišit, o jaký panel se jedná, jestli to je panel s primárním, sekundárním rozmítáním nebo rozmítáním parametru. O tom o jaký druh panelu se jedná, se rozlišuje hned v konstruktoru panelu, kde je jako vstupní parametr uveden typ panelu. Např. pokud se jedná o primární rozmítání, metoda `getSettings()` vrací na počátku textového řetězce `".DC "`, sekundární rozmítání vrací pouhé `"+ "` což naznačuje že kromě rozmítání jedné veličiny se současně rozmítá veličina druhá. Pokud se jedná o panel s rozmítání parametru, výstupní řetězec začíná posloupností znaků `".STEP"`. Za úvodním řetězcem jsou pak výpisy textových polí stejné bez ohledu na typ panelu.

4.8 Balíček Soucastky

Balíček obsahuje třídy reprezentující všechny použité součástky, včetně dvou pomocných tříd. Kompletní obsah balíčku je v příloze B. Všechny součástky jsou potomkem abstraktní třídy `Soucastka`. Abstraktní třída tak deklaruje metody a atributy společné pro všechny její potomky. Výčet atributů instancí třídy `Soucastka` je následující : Souřadnice levého horního rohu součástky `int x` a `int y`. Aktivní rozměr oblasti součástky, na kterou reaguje pozice kurzoru myši (objekt třídy `java.awt.Rectangle` s názvem `shape`). Název součástky uvedené ve schématu, `nazev_sch`. Úplný název součástky `nazev_souc`. Atribut uchovávající stav součástky (informující o tom jestli je součástka označená či nikoliv) typu `boolean` s názvem `selected`. Obrázek součástky s názvem `obrazek` a jeho označenou verzi `obrazekSelected`. Objekt, který přepočítává souřadnice zobrazení obrázku (využívá se zde hlavně funkce otáčení a zrcadlení) třídy `java.awt.geom.AffineTransform` s názvem `transform`. Řetězec uchovávající čísla uzlů, do kterých je součástka připojena, s názvem `uzly`. Atribut typu `boolean` s názvem `hasModel`, informující o tom jestli součástka obsahuje model či nikoliv. Atribut ukazující do jakého směru je právě součástka natočena třídy `Smer` s názvem `smer`. Jako poslední je zde uveden objekt obsahující parametry, které se předávají do nastavovacího okna součástky s názvem `aktualniParametry`.

Metody třídy `Soucastka` lze rozdělit do tří skupin. První skupina metod má stejný obsah pro všechny potomky třídy (nehledě na tom o jakého potomka se jedná). Druhá skupina metod má také stejný obsah pro všechny potomky třídy, jen s tím rozdílem, že potomci v případě potřeby tyto metody překryjí. Poslední skupina metod jsou tzv. abstraktní metody. Rodičovská třída pomocí abstraktních metod deklaruje, které metody musejí být povinně překryty potomky. Obsah překrytých abstraktních metod je pak u každé součástky odlišný. Součástky se od sebe liší například: ve velikosti obrázku, umístění popisky názvu a hodnotě součástky, rozdílných parametrech, počtu přípojných bodů apod.

V první skupině jsou metody : metoda `repaint(int x, int y, Graphics g)`. Zavoláním metody dojde k aktualizaci souřadnic součástky a vykreslení součástky na aktuálních souřadnicích, viz obr.4.49. Metoda `paintImage(Graphics2D g2)` , která použije k vykreslení součástky neoznačený obrázek, nebo označený obrázek, v závislosti na stavu součástky. Výpis je na obr.4.50. Metoda `changeStatus()` , která se volá při změně stavu součástky. Volání metody způsobí označení nebo odznačení součástky. Každé volání způsobí změnu atributu `selected` na opačný viz obr.4.51. Metoda `updateShape()`. Tato metoda se volá při otočení součástky. Uvnitř metody dojde k záměně rozměrů obrázku tj. šířky a výšky obrázku, viz obr.4.52. Do první skupiny metod ještě patří přístupové metody `getX()`, `getY()`, `getComponentName()`, `getShape()`, `getDimension()`, a `getTxtValue(String val)`.

```
public void repaint(int x, int y, Graphics g)
{
  this.x = x;
  this.y = y;
  paint(g);
}
```

Obr.4.49: Výpis metody repaint(int x, int y, Graphics g) třídy Soucastka

```
protected void paintImage(Graphics2D g2)
{
  if(selected)
    g2.drawImage(obrazekSelected, transform, null);
  else
    g2.drawImage(obrazek, transform, null);
}
```

Obr.4.50: Výpis metody paintImage(Graphics2D g2)

```
public boolean changeStatus()
 {
  if(selected)
    selected = false;
  else
    selected = true;

  return selected;
 }
```

Obr.4.51: Výpis metody changeStatus() třídy Soucastka

```
protected void updateShape()
{
  int temp = shapeWidth;
  shapeWidth = shapeHeight;
  shapeHeight = temp;
}
```

Obr.4.52: Výpis metody updateShape() třídy Soucastka

Druhá skupina obsahuje pouze metody `getModelText()` a `extractModelName()`. Tyto metody se překryjí jenom u součástek, které mají definovaný svůj vlastní model. Metody vrací textový řetězec výpisu modelu. U součástek, které model nemají, vrací `null`.

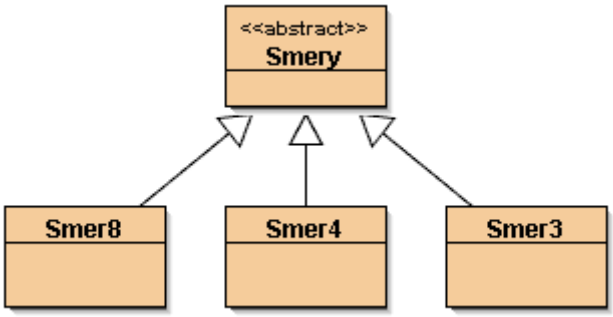
Do poslední skupiny abstraktních metod patří : metoda `paint(Graphics g)`, která vykreslí součástku na plátno. Metoda `settings()`, která zobrazí okno s aktuálním nastavením parametrů součástky. Metoda realizující otočení součástky vpravo s názvem `rotateRight()`. Dále pak metodu `getJunctionPoint()` vracející pole přípojných bodů součástky. Metodu `getNetlistText()` vracející textový zápis netlistu a nakonec metodu vracející upravené souřadnice součástky s názvem `offset()` (viz kapitola 4.5).

Obecně lze součástky rozdělit do skupin podle společných znaků. Například podle : toho jestli obsahují model, podle počtu směrů do kterých je možné součástku otočit, počtu přípojných bodů, počet parametrů součástky aj. Podle kombinace těchto kritérií jsou také třídy

součástek rozděleny. Než dojdu k samotnému rozdělení tříd, vysvětlím k čemu jsou zde třídy směrů (viz. následující kapitola) a také jakým způsobem je uchováno rozložení přípojných bodů součástek (kapitola 4.10).

4.9 Třídy Smer8, Smer4, Smer3

Všechny součástky použité v editoru se mohou otáčet do různého počtu směrů. Součástky se otáčejí po směru hodinových ručiček (po 90°). Standardně existují čtyři směry. Názvy směrů jsou pojmenovány podle světových stran v angličtině, tedy : *north, east, south a west*. Ne všechny součástky mají shodný počet směrů do kterých je možné je otočit. Součástky jako tranzistory mají navíc možnost otáčet se zrcadlově, celkem mají 8 směrů. Naproti tomu součástka GND má směry jen tři. Podle počtu směrů, do kterých je možné součástky otočit jsou pojmenovány i názvy tříd. Směry mají společného rodiče `Smery` jehož instance je využívána v abstraktní třídě součástka. Vzájemná provázanost tříd je na obr.4.53.



Obr.4.53: Třída Smer a její potomci

Abstraktní třída `Smery` obsahuje dva atributy instance. Atribut `pointer` a `pocetSmeru`. První atribut je ukazatel. Využívá se při procházení polem směrů. S každým nastavením směru se číslo uložené v ukazateli inkrementuje. Druhý atribut se odkazuje na hodnotu vyjadřující počet směrů. Tato hodnota se inicializuje v konstruktoru při vytváření potomka abstraktní třídy `Smery`. Každý potomek obsahuje výčet směrů, kterých může nabývat. Metoda, která nastavuje další směr se nazývá `setNextSmer()` a je na obr.4.54.

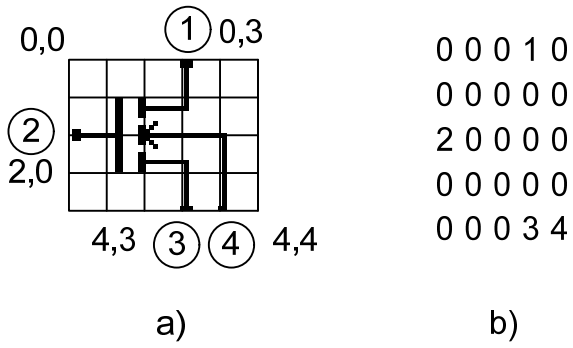
```
public void setNextSmer()
{
  if(pointer < (pocetSmeru - 1))
    pointer++;
  else
    pointer = 0;
}
```

Obr.4.54: Výpis metody setNextSmer abstraktní třídy Smery

Ukazatel se inkrementuje pokud nebylo dosaženo všech stavů. V případě že je dosažen poslední stav, ukazatel při příštím zavolání metody vynuluje.

4.10 Třída Matrix

Při generování textového zápisu netlistu, musí být dodržena posloupnost výpisu přípojných bodů součástek. Jako příklad uvedu tranzistor MOSfet, který má čtyři přípojné body (vývody). Konvence PSpice definuje posloupnost přípojných bodů v pořadí : *drain, gate, source* a *bulk*. Z hlediska uchování informace je tedy potřeba si pamatovat pořadí přípojných bodů a jejich rozmístění. K tomu slouží právě instance třídy `Matrix`. Současně v sobě uchovává informace o pořadí a rozmístění přípojných budů. Informace jsou uloženy ve dvojrozměrném poli primitivního datového typu `integer` s názvem `mat`. Princip rozmístění přípojných bodů je jednoduchý. V závislosti na velikosti obrázku reprezentující schématickou značku součástky se vytvoří rozměr pole. Jelikož jsou rozměry obrázků součástek shodné s násobkem rozměru mřížky (10 pixelů), tak je i vytvořené pole shodné s násobkem rozměru mřížky. Jeden řádek nebo sloupec pole odpovídá jedné mřížkové konstantě. Jak vypadá grafická reprezentace mřížky a obrázku součástky tranzistoru NMOS je vidět na obr.4.55.



Obr.4.55: Schématická značka tranzistoru NMOS a) rozložená do mřížky b) zjednodušený textový zápis

Obrázek tranzistoru NMOS zabírá čtyři násobky mřížky ve vertikálním směru a pět násobků mřížky ve směru horizontálním jak je vidět v části a). Čísla v kroužku vyjadřují posloupnost přípojných bodů. Čísla uvedená u vývodů znázorňují souřadnice přípojných bodů v horizontálním nebo vertikálním směru. mřížky v horizontálním a vertikálním směru. Pole přípojných bodů má rozměr 5 x 5, jak je vidět v části b).

Rozměr dvojrozměrného pole se zadává jako parametr hned v konstruktoru třídy. Konstruktor vytvoří prázdné pole o definovaných rozměrech. Metoda, která rozmístí čísla přípojných bodů na definovaných pozicích se nazývá `setPointAt(int row, int column, int numer)` a je na obr.4.56.

```
public void setPointAt(int row, int column, int number)
{
  mat[row][column] = number;
}
```

Obr.4.56: Výpis metody setPointAt(int row, int column, int numer) třídy Matrix

Metody vracející rozměr pole se nazývají `numOfRows()` a `numOfColumns()`. Při otáčení součástky současně dochází k otáčení dvojrozměrného pole. Otáčení pole provádí metoda `turnMatrix()` a využívá k tomu metodu `transpose()`. Výpisy metod jsou na obr.4.58 a obr.4.57.

```
public void transpose()
{
  int [][] temp = new int[numOfColumns()][numOfRows()];
  for(int i = 0; i<numOfColumns(); i++)
  {
    for(int j = 0; j<numOfRows(); j++)
    {
      temp[i][j] = mat[j][i];
    }
  }
  mat = temp;
}
```

Obr.4.57: Výpis metody transpose() třídy Matrix

```
public void turnMatrix()
{
  int [][] result;
  int counter = 0;

  transpose();
  result = new int[numOfRows()][numOfColumns()];
  counter = numOfColumns()-1;

  for(int k = 0; k<numOfColumns(); k++)
  {
     for(int m = 0; m<numOfRows(); m++)
     {
        result[m][k] = mat[m][counter];
     }
     counter--;
  }
  mat = result;
}
```

Obr.4.58: Výpis metody turnMatrix() třídy Matrix

Metoda `transpose()` transponuje pole, tj. prohodí řádky za sloupce. Nejprve se vytvoří lokální dvojrozměrné pole s názvem `temp` s počtem řádků shodným s počtem sloupců pole `mat` a stejně tak shodným počtem sloupců jako je počet řádků pole `mat`. Vnější `for` cyklus provádí záměnu sloupců a vnitřní cyklus záměnu řádků. Uvnitř metody `turnMatrix()` se opět vytvoří lokální pole s názvem `result` o stejném rozměru jako transponované pole. Uvnitř dvou `for` cyklů dojde k postupnému přesunutí sloupcových vektorů zprava do leva.

Otočení pole tranzitoru NMOS je na obr.4.59. Část a) vyjadřuje stav před otočením. Stav pole po zavolání metody `transpose()` je v části b). Výsledek přehození sloupcových vektorů je v části c).

0 0 0 1 0	0 0 2 0 0	0 0 2 0 0
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
2 0 0 0 0	0 0 0 0 0	0 0 0 0 0
0 0 0 0 0	1 0 0 0 3	3 0 0 0 1
0 0 0 3 4	0 0 0 0 4	4 0 0 0 0
a)	b)	c)

Obr.4.59: Stavy pole uchovávající rozložení přípojných bodů tranzistoru NMOS
a) před otočením b) stav mezi otáčením c) po otočení o 90º vpravo

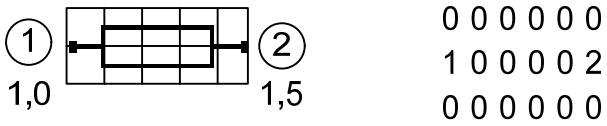
Třída `Matrix` obsahuje ještě metodu na vyhledávání čísel přípojných bodů s názvem `findNumber(int numer)`. Metoda vrací instanci třídy Point vyjadřující souřadnice bodu s daným číslem. Výpis metody je na obr.4.60.

```
public Point findNumber(int number)
{
  int row = 0;
  int column = 0;
  for(int i = 0; i<numOfRows(); i++)
  {
    for(int j = 0; j<numOfColumns(); j++)
    {
      if(mat[i][j] == number)
      {
        row = i;
        column = j;
      }
    }
  }
  return new Point(column, row);
}
```

Obr.4.60: Výpis metody findNumber(int number) třídy Matrix

4.11 Pasivní součástky R, L, C

Společným rodičem součástek rezistor, induktor a kapacitor je abstraktní třída `RLC`. Definuje metody společné těmto třem součástkám. Součástky mají stejný počet přípojných bodů včetně rozložení, stejné rozměry obrázků reprezentující součástky, umístění popisků a stejný počet parametrů. Přípojné body jsou dva, označené jako `p1` a `p2`. Čísla přípojných bodů vyjadřují jejich pořadí. Pole přípojných bodů je o rozměru 3 x 6. Jako ukázku ze tří součástek uvedu jednu a to schématickou značku rezistoru, která je na obr.4.61.

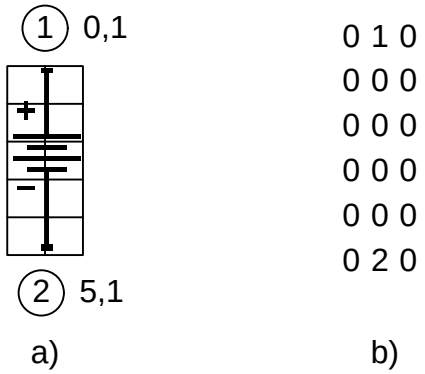


Obr.4.61: Schématická značka rezistoru a) rozložená do mřížky b) zjednodušený textový zápis

Okno s nastavením parametrů rezistoru je na obr.3.5 v části b). Mezi parametry které lze zadat je kromě názvu součástky i hodnota (odpor, kapacita, indukčnost). Jako dodatečné parametry, které se vloží na konec generovaného výpisu, je možné vložit do textového pole.

4.12 Nezávislé zdroje U/I

Nezávislé zdroje napětí a proudu (U/I) mají společného abstraktního předka s název `ZdrojVI`. Narozdíl od všech ostatních abstraktních předků součástek v sobě abstraktní třída zdrojů zahrnuje počítadla instancí napěťový zdrojů (`pocetV`) a zdrojů proudových (`pocetI`). Zdroje mají dva přípojné body, s názvem `plus` a `minus`. Standardně jsou zdroje umístěny ve svislé poloze s kladným pólem nahoře a záporným pólem dole. Pole přípojných bodů je o rozměru 6 x 3. Jako příklad schématické značky uvedu stejnosměrný zdroj, který je na obr.4.62.



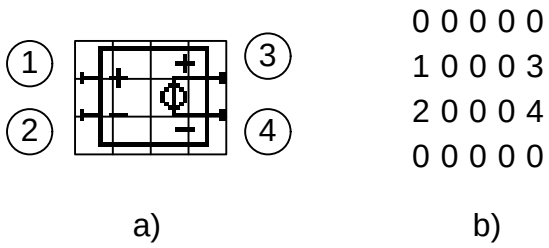
Obr.4.62: Schématická značka stejnosměrného zdroje a) rozložená do mřížky b) zjednodušený textový zápis

Skupina potomků abstraktní třídy `ZdrojVI` v sobě zahrnuje opět abstraktní třídy definující společné znaky napěťových a proudových zdrojů už konkrétních funkcí. Jsou to : stejnosměrný zdroj (`ZdrojDC`), zdroj určený pro střídavou analýzu (`ZdrojAC`) a zdroje se změnou průběhu v čase (`ZdrojSIN`, `ZdrojPULSE`, `ZdrojEXP`, `ZdrojPWL`, a `ZdrojSFFM`). Jednotlivé abstraktní třídy se liší pouze v počtu parametrů zdrojů které lze měnit a teprve konkrétní potomci těchto tříd mohou vytvořit konkrétní instance zdrojů. U stejnosměrného zdroje lze jako parametr nastavit pouze napětí na svorkách nebo dodávaný proud. Zdroj určený pro střídavou analýzu má parametry: stejnosměrnou složku (DC), amplitudu (AC) a

fázi. Zdroje se změnou průběhu v čase mají stejný počet parametrů jaký je uveden v referenční příručce PSpice, je tedy zbytečné tyto parametry opisovat. Výjimkou je pouze zdroj vlastního tvaru signálu v čase (PWL), který umožňuje zadat maximálně 6 hodnot v čase.

4.13 Závislé zdroje U/I

Závislé zdroje jsou čtyři a mají společného předka, abstraktní třídu pojmenovanou `ZdrojEFGH`. Zdroje mají jednu vstupní bránu a jednu výstupní bránu. Přípojné body jsou pojmenovány : `vstupPlus, vstupMinus, vystupPlus a vystupMinus`. Zdroje jsou standardně umístěny tak že vstupní brána je vlevo a výstupní vpravo. Pole přípojných bodů je o rozměru 4 x 5. Jako příklad schématické značky uvedu zdroj napětí řízený napětím, který je na obr.4.63.



Obr.4.63: Schématická značka zdroje napětí řízeného napětím a) rozložená do mřížky b) zjednodušený textový zápis

Konvence PSPice dovoluje zapsat závislost výstupního napětí nebo proudu v různých formách jako např. v polynomiální formě, tabulkou, vzorcem. V našem případě je na výběr pouze lineární závislost. Jako parametr lineární závislosti slouží atribut s názvem `prenos`. Z hlediska generování netlistu, mají podobný zápis napětím řízený zdroj napětí (`ZdrojE`) a napětím řízený zdroj proudu (`ZdrojG`). Také proudem řízený zdroj proudu (`ZdrojF`) a proudem řízený zdroj napětí (`ZdrojH`) mají podobnou syntaxi zápisu netlistu. Poslední dva zdroje jsou vytvořeny jako pod obvod. Pod obvod je složen ze dvou vstupních svorek a dvou výstupních svorek. Na vstupní svorky zdroje je připojen zdroj o nulovém napětí a na výstupních svorkách pod obvodu je pak připojen závislý zdroj proudu nebo napětí. Zápis podobvodu je na obr.4.64.

```
.subckt Source_F_(CCCS) 1 2 3 4
F_F1  3 4 VF_F1 1
VF_F1 1 2 0V
.ends Source_F_(CCCS)
```

Obr.4.64: Výpis netlistu podobvodu zdroje proudu řízeného proudem

4.14 Tranzistory

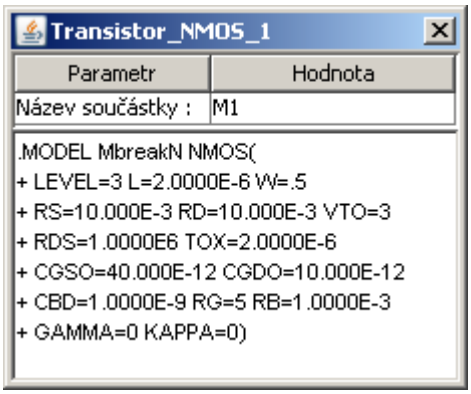
Skupina tranzistorů nemá žádného společného předka jako tomu je u většiny součástek se společnými znaky. Od ostatních součástek se odlišují tím, mají 8 směrů. Velice často totiž bývá zapotřebí otočit součástku i zrcadlově. Kromě nezrcadlených směrů existují i směry

zrcadlené. Tranzistory tak musí definovat o jedno pole přípojných bodů navíc. Při otáčení součástky se v prvních čtyřech směrech pracuje s prvním polem, v dalších čtyřech pak s druhým polem. Příklad zrcadleného a nezrcadleného pole přípojných bodů tranzistoru MOSfet je na obr.4.65.

0 0 0 1 0	0 1 0 0 0
0 0 0 0 0	0 0 0 0 0
2 0 0 0 0	0 0 0 0 2
0 0 0 0 0	0 0 0 0 0
0 0 0 3 4	3 4 0 0 0
a)	b)

Obr.4.65: Textový zápis pole přípojných bodů a) nezrcadlené pole b) zrcadlené pole

Chování tranzistoru je definováno modelem. Každý tranzistor má svůj předem definovaný model, který lze zaměnit vlastním, nebo pouze změnit některé parametry modelu. Modelu jsou převzaty z knihovny simulačního programu OrCAD PSpice. Vychází se ze standartně pojmenovaných názvů modelů. Tranzistoru MOSfet jsou přiřazeny modely `MBreakN` a `MBreakP`. Bipolárnímu tranzistoru model `QBreakP` resp. `QBreakN`. Modely `JBreakN` a `JBreakP` patří tranzistoru s trvalým kanálem, model `Bbreak` tranzistoru GaAsFET a model `ZBreakN` patří tranzistoru IGBT. Okno s nastavením parametrů tranzistoru MOSfet je na obr.4.66.



Obr.4.66: Okno s nastavením parametrů tranzistoru MOSfet

4.15 Dioda

Rozměry obrázku diody, počet směrů natočení a stejně tak rozmístění přípojných bodů je u diody shodné se součástkami R, L, C. Jediný rozdíl je ten, že dioda má svůj vlastní model.

Stejně tak jako u tranzistorů, je možné změnit buď jednotlivé parametry modelu, nebo změnit celý model.

4.16 Řízené spínače U/I

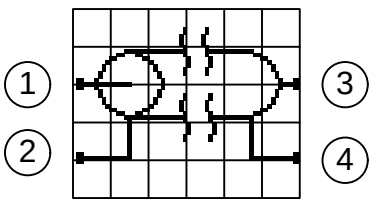
Existují dva druhy řízených spínačů, jsou to spínače řízené napětím nebo proudem. Mají shodné rozmístění přípojných bodů a stejný rozměr schématické značky jako řízené zdroje. Společným předkem řízených spínačů je abstraktní třída `Switch`. Spínač je v netlistu vytvořen jako podobvod s vlastním modelem. Každému spínači náleží jeho vlastní podobvod. Model spínače je vytvořen podle uživatelem zadaných parametrů v nastavovacím okně obvodu. Ukázka výpisu podobvodu spínače řízeného napětím je na obr.4.67.

```
.subckt SwitchS_1 1 2 3 4
S_S1   3 4 1 2 _S1
RS_S1   1 2 1G
.MODEL    _S1 VSWITCH Roff=1.0 Ron=1.0 Voff=0V Von=1V
.ends SwitchS_1
```

Obr.4.67: Výpis podobvodu spínače řízeného napětím

4.17 Přenosové vedení

Přenosové vedení reprezentuje model ideálního vedení (změnou parametrů i ztrátové vedení). Třída nemá žádného přímého předchůdce kromě třídy `Soucastka`. Rozměr schématické značky, potažmo rozměr pole přípojných bodů je největší ze všech použitých součástek. Názvy přípojných bodů jsou uvedeny v pořadí v jakém jsou vyjmenovány na obr.4.68, tedy : `coreIn, coatIn, coreOut` a `coatOut`.



0 0 0 0 0 0 0
0 0 0 0 0 0 0
1 0 0 0 0 0 3
0 0 0 0 0 0 0
2 0 0 0 0 0 4
0 0 0 0 0 0 0

a) b)

Obr.4.68: Schématická značka přenosového vedení a) rozložená do mřížky b) zjednodušený textový zápis

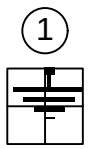
4.18 Indukční vazba

Součástka slouží buď k přiřazení magnetické vazby mezi induktory nebo magnetického jádra induktorům. O tom jaký typ vazby bude použit rozhoduje uživatel, podle toho jaké vstupní parametry vyplní. Součástka nabízí k zápisu až 6 indukčních vazeb, včetně textového okna pro zápis dodatečných parametrů. Součástka nemá žádný přípojný bod (tedy ani pole

přípojných bodů) a ani svoji schématickou značku. Je reprezentován symbolem čtverce s označením čísla vazby uvnitř.

4.19 Třída Gnd

Instance třídy `Gnd` reprezentuje „uzemnění“ – tzv. společný potenciál. Umístěním uzemnění do obvodu se změní číslo uzlu, do kterého je připojen na nulu. Podle konvence PSpice se tak označuje uzel s nulovým, referenčním potenciálem. Uzemnění má pouze jeden přípojný bod a 3 směry natočení. Neotáčí se schématická značka jak je u ostatních součástek zvykem, ale otáčí se pouze text okolo značky. Kvůli jednomu přípojnému bodu není zapotřebí vytvářet žádné pole přípojných bodů, zvlášť když zde nedochází k rotaci součástky, resp. přípojného bodu. Souřadnice přípojného bodu jsou vůči souřadnicím součástky posunuty vždy o stejnou vzdálenost. Vzdálenost je rovna jednomu násobku mřížky v horizontálním směru. Schématická značka uzemnění spolu s umístěním přípojného bodu je na obr.4.69.



Obr.4.69: Schématická značka uzemnění

4.20 Třída Parametr

Poměrně často bývá potřeba vytvořit proměnnou přímo uvnitř netlistu namísto konkrétní hodnoty a na tuto proměnnou se odkazovat, popřípadě s ní pracovat, provádět výpočty atd.. Syntaxe jazyka PSPICE vytvoření takovéto proměnné umožňuje. K tomu slouží příkaz .PARAM . Konkrétní implementace tohoto příkazu v editoru zajišťuje právě třída Parametr, resp. její instance. Třída Parametr se oproti ostatním třídám součástek liší hned v několika věcech. Ve své podstatě by se dalo říct, že to vlastně ani není součástka, jelikož instance této třídy není reprezentována žádným grafickým symbolem, resp. schématickou značkou, není možné ji otáčet a nemá ani žádný přípojný bod. Třída by klidně mohla být umístěna v balíčku `Main` s třídami jako třídy `Vodic` nebo `Spoj`. Ale důvodem pro začlenění této třídy do balíčku společně s ostatními součástkami je fakt, že při generování netlistu se chová jako součástka. Obsahuje totiž metodu, která je společná pro všechny součástky a tou je metoda `getNetlistText()`. Další zvláštností třídy je, že dovoluje vytvořit pouze jedinou instanci třídy `(SINGLETON)`, stejně jako třída `Editor`. Je to tak vytvořeno kvůli přehlednosti. Přehlednost a orientaci ve schématu by určitě nezlepšilo, kdyby se pro každý definovaný parametr vytvořila nová instance parametru, tzn. co nový parametr, to nový nadpis „PARAMETR“ ve schématu a pod ním jeho jednořádkový popis. V případě vytvoření více parametrů by se tak schéma stalo nepřehledné. Namísto toho, je při každém dalším volání o umístění instance na plátno je předán odkaz na již existující instanci třídy `Parametr`, která se přesune na pozici kurzoru o

vstupních souřadnicích `x, y`. Tuto funkci zajišťuje statická metoda `getInstanceAt(int x, int y)`. Implementace metody je na obr.4.70.

```
public static Parametr getInstanceAt(int x, int y)
{
  SINGLETON.x = x;
  SINGLETON.y = y;
  return SINGLETON;
    }
```

Obr.4.70: Metoda getInstanceAt(int x, int y)

Uvnitř metody se aktualizují staré hodnoty souřadnic za nové a jako návratový typ se vrací odkaz na jednu a tu samou instanci.

Instance třídy Parametr má jako každá součástka, nastavovací okno, kde co řádek to parametr a jemu odpovídající hodnota. Jako dostačující se jeví použití pěti řádků, čemuž odpovídá nastavení statického atributu třídy `pocetRadku` na hodnotu 5. Parametry jsou rozděleny do dvou jednorozměrných polí s názvy `nazev` a `hodnota`. Implicitně jsou všechny prvky polí nastaveny jako prázdný řetězec metodou `stringInit()`. Pokud by tomu tak nebylo, prvky by ukazovali na hodnotu `null`. Výpis části kódu soukromého konstruktoru třídy je na obrázku obr.4.71, metoda `stringInit()` pak na obrázku obr.4.72.

```
private Parametr()
{
 ...
 shapeWidth = GRID*nazev_sch.length();
 shapeHeight = GRID;
 pocetRadku = 5;
 uzly = new int[0]; // nema zadny pripojny bod
 nazev = new String[pocetRadku];
 hodnota = new String[pocetRadku];
 stringInit();
 ...
}
```

Obr.4.71: Výpis části kódu soukromého konstruktoru třídy Parametr

```
public void stringInit()
{
 for(int i = 0; i < pocetRadku; i++)
 {
  nazev[i] = "";
  hodnota[i] = "";
 }
}
```

Obr.4.72: Metoda stringInit()

Cyklus proběhne právě tolikrát, kolikrát je uvedeno v atributu `pocetRadku`, v našem případě tedy pětkrát. Atributy se nastaví na prázdný řetězec. Metoda `stringInit()` se také volá při změně jazyka, nebo v případě vytvoření nového plátna, kdy je potřeba znovu zajistit inicializaci polí.

Třída také definuje veřejnou statickou konstantu `ROZESTUP_TEXTU`, která udává odstup mezi řádky výpisu jednotlivých parametrů na plátno. Vyjadřuje hodnotu v pixelech a je nastaveno na hodnotu 3. Instance třídy `Parametr` je vytvořena hned v její inicializaci. Jako soukromé atributy instance jsou zde dvě jednorozměrné pole primitivního datového typu `String` a to `nazev` a `hodnota`.

Jako aktivní oblast instance je vymezena textová oblast názvu „PARAMETR“, s výškou danou konstantou mřížky (10 pixelů) a šířkou danou počtem písmen uložených v atributu názvu schématické značky (`nazev_sch`).

Výpis parametrů na plátno se provádí obdobně jako u jiných součástek, ovšem s tím rozdílem, že zde se porovnává, jestli uživatel vyplnil nějaký řádek určený pro zadávání parametrů. Cyklus projede obsahy jednorozměrných polí a zjistí jestli je v nich něco zapsáno. resp. jestli obsahují alespoň jeden znak. Pokud je tedy ve stejném řádku v obou z těchto polí (`název` a `hodnota`) obsažen alespoň jeden znak, zobrazí se tento řádek na plátně. Mezi tyto řetězce se automaticky přidá rovnítko. Vertikální souřadnice textu je vypočtena z výšky textu, konstanty udávající odstup mezi řádky (`ROZESTUP_TEXTU`) a konstantní hodnotou mřížky (`GRID`). Celé je to pak násobeno počítadlem cyklu. Výpis částečného kódu je na obrázku obr.4.73.

```
public void paint(Graphics g)
{
...
  for(int i = 0; i < nazev.length; i++)
  {
   // jestlize je radek vyplneny, vykresli jej
   if(!nazev[i].equals("") && !hodnota[i].equals(""))
   {
    g2.drawString(nazev[i] + " = " + hodnota[i],
         x, y+2*shapeHeight+ROZESTUP_TEXTU+(ROZESTUP_TEXTU+GRID)*i);
   }
  }
}
```

Obr.4.73: Výpis části kódu metody paint(Graphics g)

Jelikož parametr nemá žádný grafický symbol, ale pouze textovou reprezentaci, zděděné atributy jako `obrazek` a `obrazekSelected` zůstávají prázdné (`null`). Parametrem také nelze nijak otáčet, z tohoto důvodu je tělo metody `rotateRight()` prázdné. Metoda `getNetListText()` obsahuje podobnou část kódu jako metoda na obr.4.73. Ovšem nepracuje s objektem typu `Graphics` ale s primitivním typem `String`, kde k vytvořenému textu přidává parametry, pokud jsou přítomny. Parametry jsou přidávány metodou `concat(String str)` třídy `String`. Výpis metody `getNetListText()` je na obr.4.74.

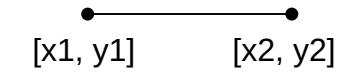
```
public String getNetlistText()
{
  String temp = ".PARAM";
  for(int i = 0; i < nazev.length; i++)
  {
   if(!nazev[i].equals("") && !hodnota[i].equals(""))
    temp = temp.concat(" " + nazev[i] + "=" + hodnota[i] + " ");
  }

  return temp;
}
```

Obr.4.74: Výpis metody getNetlistText()

4.21 Třída Vodic

Podle názvu třídy je zřejmé, že instance této třídy reprezentuje objekt sloužící k propojení součástek na plátně. Třída se nenachází v balíčku společně se součástkami. S těmito instancemi se zachází jinak než se součástkami a to jak při kreslení tak hlavně při generování netlistu. Pro tuto třídu je vymezeno místo v balíčku `main` společně s rozhranním `IKresleny`, které implementuje.

Mezi důležité atributy instance třídy `Vodic` patří: souřadnice počátečního a koncového bodu, obdélník vymezující aktivní oblast vodiče (na tuto oblast reaguje pozice kurzoru myši) a odkaz objekt reprezentující jeho grafickou podobu. Souřadnice bodů jsou typu `integer` s názvy `x1, x2, y1, y2`. Aktivní oblast vodiče je objekt typu `java.awt.Rectangle` a grafickou reprezentaci vodiče zastupuje objekt typu `java.awt.geom.Line2D`. Grafické znázornění vodiče je obr.4.75.



Obr.4.75: Grafické znázornění vodiče včetně přípojných bodů

K vytvoření vodiče je nutné v konstruktoru uvést souřadnice počátečního a koncového bodu. Uvnitř konstruktoru, ale hlavně při práci s vodičem, se používá metoda `updateCoordinate(int x1, int y1, int x2, int y2)`. Metoda aktualizuje atributy, ve kterých jsou uloženy souřadnice počátečního a koncového bodu. Výpis metody je na obr.4.76.

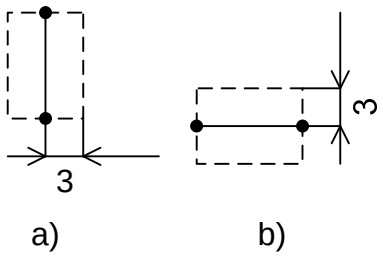
```
private void updateCoordinate(int x1, int y1, int x2, int y2)
 {
  // kresli se zprava do leva? jestli ano, prehod souradnice
  // (kvuli zachovani LH horniho rohu soucastku ktery je vychozi)

   if((x2 - x1) < 0)
   {
     int temp = x1;
     x1 = x2;
     x2 = temp;
   }

   // kresli se sdola nahoru? jestli ano, prehod souradnice
```

```
   // (kvuli zachovani LH horniho rohu soucastku ktery je vychozi)

   if((y2 - y1) < 0)
   {
     int temp = y1;
     y1 = y2;
     y2 = temp;
   }

   // aktualizuje souradnice
  this.x1 = x1;
  this.y1 = y1;
  this.x2 = x2;
  this.y2 = y2;

  vodic.setLine(x1, y1, x2, y2);
  updateShape();
 }
```

Obr.4.76: Výpis metody `updateCoordinate(int x1, int y1, int x2, int y2)` třídy Vodic

Souřadný systém, který pracuje s kreslenými komponentami předpokládá, že se souřadnice počítají od levého horního rohu komponenty. Uvnitř metody se proto zjišťuje, kde se nachází počáteční a koncový bod vodiče. Vždy platí že počáteční bod je buď vlevo, nebo nahoře.

Aktivní oblast vodiče se je vymezena na 3 pixely od středu vodiče na obě strany, tak jak je vidět na obr.4.77.



Obr.4.77: Aktivní oblast vodiče a) svilý směr b) vodorovný směr

Jak je tomu u součástek, tak i zde je metoda zajišťující aktualizaci aktivní oblasti pojmenována `updateShape()`. Výpis metody je na obr.4.78.

```
private void updateShape()
{
  int width = (int)vodic.getBounds().getWidth();
  int height = (int)vodic.getBounds().getHeight();

  // aktualizuje souradnice obrysu
  if (width > height)
     // otoceno horizontalne
    shape = new Rectangle(x1-1, y1-1, width + 1, height + 3);
  else // otoceno vertikalne
    shape = new Rectangle(x1-1, y1-1, width + 3, height + 1);
}
```

Obr.4.78: Výpis metody updateShape() třídy Vodic

V kapitole 4.3.5 je uvedena metoda zajišťující rozdělení vodičů s názvem `splitAtPoint(Point p)`, která využívá metodu vodiče nazvanou `contains(Point p)`. Metoda zjistí zda se bod předaný jako parametr nachází uvnitř vodiče. Výpis metody je na obr.4.79.

```
public boolean contains(Point p)
{
   boolean vysledek = false;
   int pX = (int)p.getX();
   int pY = (int)p.getY();

   if ((pX == x1) && (x1 == x2)) //vodic vertikalne
    vysledek = (y1 < pY && pY < y2) ;

   if ((pY == y1) && (y1 == y2)) //vodic horizontalne
    vysledek = (x1 < pX && pX < x2);

   return vysledek;
}
```

Obr.4.79: Výpis metody contains(Point p) třídy Vodic

První podmiňovací příkaz vymezuje případ vertikálně položeného vodiče (souřadnice v ose x jsou stejné). Porovnává se jestli souřadnice bodu náleží do intervalu v ose y. Druhý podmiňovací příkaz vymezuje případ horizontálně položeného vodiče. Testuje se interval v ose x.

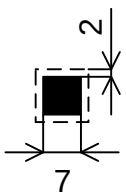
Metoda ověřující nulovou délku vodiče se nazývá `isNullLenght()` a využívá se v případě uvolnění tlačítka myši v režimu kreslení vodičů (`mouseReleased(MouseEvent e)`). Výpis metody je na obr.4.80.

```
public boolean isNullLength()
{
   return (x1 == x2 && y1 == y2);
}
```

Obr.4.80: Výpis metody isNullLenght() třídy Vodic

4.22 Třída Spoj

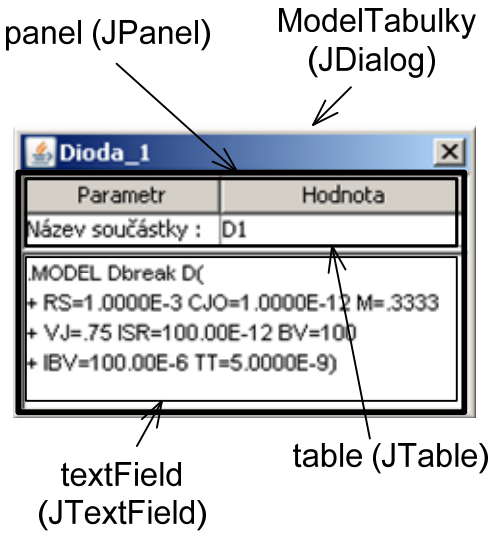
Třída `Spoj` se stejně jako třída `Vodic` nachází v hlavním balíčku. Instance této třídy reprezentuje vodivé spojení vodičů. Jako označení pro spojení vodičů se používá čtverec. Délka strany je 7 pixelů, aktivní oblast komponenty je o 2 pixely vyšší. Spoj má jeden přípojný bod nacházející se přímo ve středu čtverce. Značka spoje je vidět na obr.4.81.



Obr.4.81: Značka vodivého spojení vodičů včetně označení aktivní oblasti

4.23 Třídy oken s nastavením parametrů

Třídy s oknem nastavením parametrů mají na konci svého názvu „GUI“, např. `RLCGUI`, `ZdrojVIGUI` apod. Třídy oken jsou společné pro součástky, které mají společného předka. Okno s nastavením parametrů je potomkem třídy `javax.swing.JDialog`. Jedná se o modální dialogové okno, které má tu vlastnost, že v době kdy je aktivní, uživatel nemá možnost přepnout do jiného okna dokud okno neuzavře. Při konstrukci modálního okna se uvádí odkaz na okno které je modálnímu oknu nadřazené. V našem případě to je vždy okno editoru. Nastavovací okno je buď kombinací tabulky s textovým polem nebo jen samotné tabulky. Příklad kombinace tabulky a textového pole je například u nastavovacího okna diody, viz obr.4.82.



Obr.4.82: Nastavovací okno diody

Každé okno s nastavením parametrů implementuje rozhranní `windowListener` reagující na události okna. Překrytá je ale pouze metoda ošetřující událost při zavírání okna (`windowClosing(WindowEvent e)`) uvnitř které se volá metoda `dispose()`, zavírající okno. Součástí každé třídy s nastavením parametrů je také vnořená třída `ModelTabulky`. Třída je odvozená od abstraktní třídy `javax.swing.table.AbstractTableModel` definující počet a obsah řádků tabulky. Každá typ součástky obsahuje různý počet řádků. Sloupce jsou pouze dva. První slouží jako název parametru a druhý pro zadání hodnoty. Názvy řádků jsou uloženy v atributu `columnNames`. Obsahy polí jsou zase ve dvojrozměrném atributu `parametry`. Povinně překryté metody abstraktního předka jsou: `getColumnCount()` vracející počet sloupců, `getRowCount()` vracející počet řádků tabulky, `getValueAt(int row, int col)` vracející objekt nacházející se na určitém řádku a sloupci tabulky, `getColumnName(int col)` vracející název určitého řádku a metody `isCellEditable(int row, int col)` a `setValueAt(Object aValue,`

`int rowIndex, int columnIndex)`. První metoda definuje, které řádky nebo sloupce u kterých může uživatel měnit jejich obsah a druhá metoda nastavuje obsah dané buňky tabulky.

Součástky, které mají vlastní model obsahují metody pro nastavení modelu a jeho extrakci. Metody se nazývají `setModel()` a `getModel()`. Metoda, kterou využívají součástky k předání parametrů se nazývá `Object[] aktivujOkno(Object[] pravySloupec)`. Výpis této metody pro diodu je uveden obr.4.83.

```
public synchronized Object[] aktivujOkno(Object[] pravySloupec)
{
 int i; // promenna na prochazeni cyklem
 for(i=0; i < table.getRowCount(); i++)
 {
   table.setValueAt(pravySloupec[i], i, 1);
 }

// posledni dve hodnoty v pravemSloupci udavaji souradnice soucastky
setPoziciOkna(new Point(Integer.parseInt((String)pravySloupec[i++]),
            Integer.parseInt((String)pravySloupec[i])));

pack();
setVisible(true);

// zmenene hodnoty tabulky
Object [] noveParametry = new Object [] {table.getValueAt(0,1)};

return noveParametry;
}
```

Obr.4.83: Výpis metody aktivujOkno(Object[] pravySloupec) třídy Dioda

Cyklus nastaví hodnoty parametrů součástky uvnitř tabulky. Jako poslední dvě hodnoty jednorozměrného pole jsou vždy zasílány souřadnice součástky. Nastavovací okno se totiž zobrazí vždy na pozici součástky. To zajišťuje metoda `setPoziciOkna(Point souradniceSouc)`. Jako parametr se předává právě souřadnice součástky. Z rozměru nastavovacího okna a ze souřadnic bodu se vypočítá pozice nastavovacího okna. Výpis metody je na obr.4.84.

```
private void setPoziciOkna(Point souradniceSouc)
{
 Editor ed = Editor.getEditor();
 Point pozicePlatna = ed.getCanvasPosition();
 int x = (int)souradniceSouc.getX() + (int)pozicePlatna.getX();
 int y = (int)souradniceSouc.getY() + (int)pozicePlatna.getY();
 Dimension rozmerOkna = this.getSize();
 GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
 Rectangle rect = ge.getMaximumWindowBounds();

 //vycentrovani okna, aby bylo nad soucastkou
 x -= rozmerOkna.width / 2;
 y -= rozmerOkna.height / 2;

 // osetreni aby okno nebylo zobrazeno mimo obrazovku
 if (x > rect.width-rozmerOkna.width)
    x = rect.width/2;
 else if (x < rozmerOkna.width/2)
    x = 0;

 if (y > rect.height-rozmerOkna.height)
    y = rect.height/2;
 else if (y < rozmerOkna.height/2)
    y = 0;

 // nastaveni pozice okna
 setLocation(x, y);
}
```

Obr.4.84: Výpis metody setPoziciOkna(Point souradniceSouc)

5 Závěr

Hlavním cílem diplomové práce bylo vytvořit internetové uživatelské prostředí pro tvorbu elektronických schémat. Veškeré cíle kladené v zadání byly náležitě splněny. Výsledkem práce je program, který umožňuje tvorbu schémat pod webovým rozhraním s pomocí předem definované knihovny součástek. Program je vytvořen v Javě a je spustitelný z jakéhokoliv internetového prohlížeče s podporou apletů. Výstupem programu je generovaný netlist elektronického schématu ve formátu PSpice.

Kapitoly diplomové práce jsou účelně rozděleny od výběru vhodného programového prostředí, přes návrhu programu až po jeho realizaci. Součástí diplomové práce jsou i části zdrojových kódů včetně popisu. Kompletní zdrojový kód programu je přiložen na CD. Přiložené CD obsahuje spolu s programovým kódem také použité vývojového prostředí a instalační balíček Javy nutného ke spuštění programu.

Samotný programový kód doznal během vývoje řady změn. Z počátku nebylo hned jasné jaké nároky by měl editor přesně splňovat a co bylo nutné zajistit pro správnou funkci programu jako celku. Jakákoliv větší změna v programu (např. dodatečná lokalizace) měla za následek nepoměrně větší časové nároky, než kdyby se změnou bylo uvažováno už od počátku vývoje programu. Z toho plyne, že vždy před zahájením tvorby, a to ať už se jedná o programu, nebo jakéhokoliv projektu vůbec, je zapotřebí důkladná analýza problému a definice všech vlastností před započetím práce. Veškeré změny prováděné v programu, vedly k zefektivnění práce, nebo větší flexibilitě při dodatečných změnách programového kódu. Navzdory některým problémům vzniklým v průběhu tvorby programu, se podařilo vyvinout plnohodnotný nástroj, ve kterém je vidět souvislost mezi kresleným schématem v grafické podobě a generovaným netlistem v textové podobě.

Při testování programu nebyly zaznamenány žádné chyby, při kterých by program nepracoval správně nebo neplnil svoji funkci. Jeden drobný nedostatek, ale program obsahuje. Když se program poprvé inicializuje a uživatel vyvolá požadavek na umístění součástky na kreslící plátno, dojde k alokaci paměti. Protože program potřebuje ke své funkci více paměti než je standartně přiděleno. Přidělování paměti probíhá automaticky podle potřeby a zabere určitý čas. Než dojde k alokaci potřebné paměti, je vidět prodleva mezi stiskem tlačítka myši a umístěním první součástky na kreslící plátno. K tomuto jevu dochází pouze při prvním spuštění programu a umístěním součástky na plátno. Umístěním dalších součástek, nebo po uzavření programu a dalšímu znovu spuštění samozřejmě k žádné prodlevě nedochází. Prodlevu je možné snížit u samostatně spustitelného souboru tím, že se spustí s parametrem definujícím výchozí a maximální velikost alokované paměti (viz kapitola 4). U apletu není možné možné ovlivnit počáteční velikost vyrovnávací paměti, protože JVM se automaticky spouští před spuštěním apletu.

Co se týká dalšího vývoje programu, z uživatelského hlediska by si editor zasloužil např. implementaci funkce manipulace s více součástkami najednou, přidání kontextového menu apod. Z programátorského hlediska by zase bylo výhodné např. vytvořit rozhranní součástkám, které mají stejný počet vývodů. Nebo upravit třídy oken s nastavením parametrů tak, aby zde byla jedna společná třída spolu s tovární metodou, která by vytvářela modifikované instance podle požadavků konkrétní třídy součástek, viz [17].

Na konec bych chtěl uvést, že vytvořený editor je využíván studenty ústavu mikroelektroniky a je umístěn na webových stránkách předmětu Modelování a simulace v mikroelektronice, zde [18]. Na těchto stránkách je také umístěn simulační nástroj, který provádí simulaci generovaného netlistu. Program byl také publikován na konferenci Electronic Device and Systems EDS 2008 [19].

6 Seznam použitých zdrojů

[1] KOLKA, Zdeněk, BIOLEK, Dalibor. *Snap : symbolická, semisymbolická a numerická analýza elektronických obvodů* [online]. Verze 2.61. [2004] , 1.7.2004 [cit. 2009-04-20]. Text v češtině. Dostupný z WWW: http://snap.webpark.cz/.

[2] KÖLLING, Michael, et al. *BlueJ : The interactive Java enviroment* [online]. Verze 2.5.0. Kent (UK) : 7.11.2008 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://www.bluej.org/index.html.

[3] Přispěvatelé Wikipedie, *Unified Modeling Language* [online], Wikipedie: Otevřená encyklopedie, c2009, Datum poslední revize 9. 02. 2009, 23:35 UTC, [citováno 20. 04. 2009] http://cs.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=3605882.

[4] PECINOVSKÝ, R. *Myslíme objektově v Jazyku Java 5.0.* Grada Publishing. 2004. 601 stran. ISBN 80-247-0941-4

[5] KEOGH, J. *Java bez předchozích znalostí.* Computer press. 2005. 275 stran. ISBN 80-251-0839-2.

[6] FOTR, J., SCHNEIDER, Z. *Flash pro grafiky a tvůrce webů.* Computer press. 2000. 199 stran. ISBN 80-7226-415-X

[7] VIRIUS, M. *Java pro zelenáče.* Neocortex. 2001. 240 stran. ISBN: 80-902230-9-5

[8] Přispěvatelé Wikipedie, *Prohledávání do hloubky* [online], Wikipedie: Otevřená encyklopedie, c2009, Datum poslední revize 24. 02. 2009, [citováno 20. 04. 2009] http://cs.wikipedia.org/w/index.php?title=Prohled%C3%A1v%C3%A1n%C3%AD_do_hloubky&oldid=3659377.

[9] Sun Microsystems. *Java reference guide* [online]. c2003 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://java.sun.com/j2se/1.4.2/docs/api/.

[10] SIMONSEN, Keld. *Code for the representation of names of languages* [online]. 1990-11-30 , last updated 1996-07-28 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt.

[11] Sun Microsystems. *Creating a GUI with JFC/Swing* [online]. c1995-2008 , last updated 2/14/2008 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://java.sun.com/docs/books/tutorial/uiswing/index.html.

[12] Unicode. *The Unicode® Standard : A Technical Introduction* [online]. c1991-2009 , last updated 17. 9. 2008 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://www.unicode.org/standard/principles.html#Assigning_Codes.

[13] ITPro CZ. *Unicode Characters to Java Entities Converter : A online utility to convert Unicode characters to Java entities and back* [online]. c2003-2009 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://itpro.cz/juniconv/

[14] Midpoint systems. *Jvider : Visual Java GUI builder* [online]. c2003-2007 , last updated 5 July 2007 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://www.jvider.com/.

[15] Sun Microsystems. *Laying Out Components Within a Container* [online]. c1995-2008 , last updated 2/14/2008 [cit. 2009-04-20]. Text v angličtině. Dostupný z WWW: http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html.

[16] Microsim. *Reference Manual Microsim PSpice A/D* [online]. Verze 8. [1997] [cit. 2009-04-21]. Text v angličtině. Dostupný z WWW: http://www.berks.psu.edu/faculty/litwhiler/pspcref.pdf.

[17] PECINOVSKÝ, R. *Návrhové vzory - 33 vzorových postupů pro objektové programování*. Computer press 2007. První vydání. 527 stran. ISBN 978-80-251-1582-4

[18] KADLEC, Jaroslav, POPELKA, Lukáš. *Internet Graphical Simulation Tool* [online]. [2009] [cit. 2009-04-21]. Text v angličtině. Dostupný z WWW: http://www.umel.feec.vutbr.cz/mmsi/igst/.

[19] KADLEC, J.; POPELKA, L. Internet graphical simulation tool for computer analysis of electrical circuits. In Electronic Devices and Systems EDS´08. Brno: Ing. Zdeněk Novotný CSc., 2008. s. 1-5. ISBN: 978802143717-3.

7 Přílohy

7.1 Příloha A – výpis metod reagujících na události myši

```
public void mousePressed(MouseEvent e)
{
  reqFocus(); // vrati focus na okno (aby fungoval KeyListener)

  // aktualni souradnice kurzoru
  last_x = e.getX();
  last_y = e.getY();

  // Vytvori novou komponentu :
  if(SwingUtilities.isLeftMouseButton(e))
  {
     IKresleny s = null; // komponenta ktera se bude pridavat do seznamu
     if(getCursorState() == "place") // stav vytvoreni soucastky
      s = createSelectedItem(last_x, last_y);

     if(getCursorState() == "junction") // stav vytvoreni noveho spoje
      s = new Spoj(last_x, last_y);

    // jestlize je vybrano ze seznamu, nebo je vytvoren novy spoj, pridej
    // komponentu na platno
     if (s!= null)
     {
     // zjisti rozmer vybrane komponenty a vycentruje ji na stred kurzoru a
     // znovu ji prekresli
        Dimension dim = s.getDimension();

     // jestlize se jedna o spoj, nevycentruje souradnice (uz jsou)
        if(s instanceof Spoj)
         s.repaint(last_x, last_y, drawingPane.getGraphics());
        else
         s.repaint(last_x-dim.width/2,                        last_y-dim.height/2,
drawingPane.getGraphics());

     // Jestlize se jedna o soucastku Parametr, zkontoruluje jestli
     // uz nahodou v seznamu existuje. Jako jedina soucastka Parametr splnuje
     // kriterium pro metodu contains(Object) tridy arraylist, protoze
     // tato metoda pouziva k porovnani 2 objektu metodu equals()
     // Jako jedina trida ze soucastek ukazuje vraci trida Parametr odkaz
     // na stejnou instanci.

        if(!componentList.contains(s))
        {
          // parametr neni v seznamu, pokracuje dal ve vykreslovani
          componentList.add(s); // prida do seznamu
          setEditorStatusText("pridan "+ s.getComponentName());
          drawingPane.scrollRectToVisible(s.getShape());

          // rozsiri platno podle posledni pridane soucastky
          resizeCanavas(s);
        }
        updateLastComponent(s); // aktualizuje posledni odkaz

        // vraci pozici soucastky v seznamu
        lastComponentIndex = componentList.indexOf(s);
        drawingPane.repaint();
     }

     // nakresleni vodice
     if(getCursorState() == "line")
```

```
    {
      IKresleny s1,s2; // komponenty ktera se bude pridavat do seznamu

      s1 = new Vodic(last_x, last_y, last_x, last_y);
      s2 = new Vodic(last_x, last_y, last_x, last_y);
      s1.rotateRight();

      componentList.add(s1); // prida do seznamu
      componentList.add(s2); // prida do seznamu

      //vraci pozici soucastky v seznamu
      lastComponentIndex = componentList.indexOf(s2);
     }

    if(getCursorState() == "move")
    {
      IKresleny kres = getKresleny(last_x,last_y);

      if (kres != null){
       lastComponentIndex = componentList.indexOf(kres);
       updateLastComponent(kres);

       if((kres instanceof Soucastka) || (DEBUG))
          setEditorStatusText("Soucastka " + kres.getComponentName() + " oznacena
");

      }
      else{
         lastComponentIndex = -1;
         updateLastComponent(null);
      }
    }
  }
    // zmacknuty soucasne LMB + RMB, muze otacet soucastku a vodic
    if(e.getModifiersEx() == (BOTH_PRESSED))
    {
      if(lastComponentIndex != -1) // je vybrano ze seznamu
      {
        IKresleny kres = (IKresleny)componentList.get(lastComponentIndex);
        kres.rotateRight();

      // zjisti rozmer vybrane komponenty a vycentruje ji
      // na stred kurzoru a znovu prekresli
        Dimension dim = kres.getDimension();

        if(kres instanceof Spoj)
          kres.repaint(last_x, last_y, drawingPane.getGraphics());
        else
          kres.repaint(last_x-dim.width/2,                    last_y-dim.height/2,
drawingPane.getGraphics());

        drawingPane.repaint();
      }

    }

}

// ****************************************************************************
public void mouseDragged(MouseEvent e)
{
  // rezim nasleduje po akci MousePressed, nezalezi na stavu kurzoru,
  // protoze neplati podminka lastComponentIndex != -1
  // stav kurzoru je rozhodujici v metode MousePressed
```

```
 if((SwingUtilities.isLeftMouseButton(e) && lastComponentIndex != -1))
 {
   if(getCursorState() != "line")
   {
   // odkaz na soucastku ktera byla naposled vybrana v metode MousePressed
   IKresleny kres = (IKresleny)componentList.get(lastComponentIndex);

   // aktualni souradnice kurzoru
   int x = e.getX();
   int y = e.getY();

   // vypocet rozdilu souradnic aktualne oznacene soucastky
   int new_x = kres.getX() + x - last_x;
   int new_y = kres.getY() + y - last_y;

   // Zobrazi status tex pri presunu soucastky
   if ((kres instanceof Soucastka) || (DEBUG))
     setEditorStatusText("Soucastka " + kres.getComponentName() + " presouvana " +
new_x + ":" + new_y );
   else
     setEditorStatusText(e.getX()+":"+e.getY());

   kres.repaint(new_x, new_y, drawingPane.getGraphics());
   //update souradnic
   last_x = x;
   last_y = y;
   drawingPane.repaint();
  }

  // v pripade kresleni vodice
  if(getCursorState() == "line")
  {
   // odkaz na soucastku ktera byla naposled vybrana v metode MousePressed
   // v1 je vertikalni vodic, v2 je horizontalni vodic
   Vodic v1 = (Vodic)componentList.get(lastComponentIndex - 1);
   Vodic v2 = (Vodic)componentList.get(lastComponentIndex);

   setEditorStatusText(e.getX()+":"+e.getY());

   // aktualni souradnice kurzoru
   int x = snapToGrid(e.getX());
   int y = snapToGrid(e.getY());

   last_x = snapToGrid(last_x);
   last_y = snapToGrid(last_y);

   v2.prekresliVodic(last_x, last_y, x, last_y, drawingPane.getGraphics()); // H
   v1.prekresliVodic(   x,   y, x, last_y, drawingPane.getGraphics()); // V

   drawingPane.repaint();
  }
 }
}
// ********************************************************************
public void mouseReleased(MouseEvent e)
{
 if(SwingUtilities.isLeftMouseButton(e) && lastComponentIndex != -1)
 {
  // byla prave upustena soucastka? Jestli ano, prichyt ji k mrizce
  if((getCursorState() != "line"))
  {
    // Vrati soucastku se kterou probihal presun
    IKresleny kres = (IKresleny)componentList.get(lastComponentIndex);
    Dimension dim = kres.getDimension(); // rozmery aktualni soucastky
```

```
    int x = e.getX() - dim.width/2;
    int y = e.getY() - dim.height/2;

    // odsadi soucastu v pripade potreby, vraci novou souradnici nebo stavajici
    Point bod = kres.offset();

    x = snapToGrid((int)bod.getX());
    y = snapToGrid((int)bod.getY());

    kres.repaint(x, y, drawingPane.getGraphics());
    setEditorStatusText("prichyceno k mrizce ");
    resizeCanavas(kres);
    drawingPane.repaint();
    lastComponentIndex = -1; // neni nic vybrano
  }
  else
 // byly kresleny vodice, odstran vodice s nulovou delkou, pokud zde nejake vznikli
  {
    IKresleny kres1 = (IKresleny)componentList.get(lastComponentIndex);
    IKresleny kres2 = (IKresleny)componentList.get(lastComponentIndex - 1);
    Vodic v1 = (Vodic)kres1;
    Vodic v2 = (Vodic)kres2;

    if(v1.isNullLength()){
      componentList.remove(lastComponentIndex);
    }

    if(v2.isNullLength()){
      componentList.remove(lastComponentIndex - 1);
    }

    lastComponentIndex = -1; // vynuluje posledni odkaz
  }
 }
}
// ************************************************************************
public void mouseClicked(MouseEvent e)
{
  // dvojity stisk LMB
  if (SwingUtilities.isLeftMouseButton(e) && e.getClickCount() == 2)
  {
    int x = e.getX();
    int y = e.getY();
    IKresleny kres = getKresleny(x, y);

  // zobrazi okno v pripade ze se jedna o soucastku nebo je zapnuty debug rezim
    if (((DEBUG) || (kres instanceof Soucastka)) && kres != null){
       kres.settings();
       drawingPane.repaint();
    }
  }
}
```

7.2 Příloha B - obsah balíčku Součástky