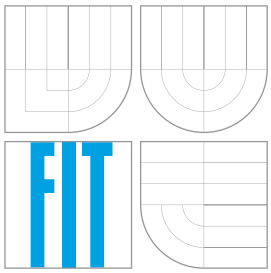
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO MODELOVÁNÍ A SIMULACI TECHNOLOGICKÝCH PROCESŮ

TOOL FOR MODELLING AND SIMULATION OF TECHNOLOGICAL PROCESSES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LIBOR SKYDÁNEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. RADEK KOČÍ, Ph.D.
SUPERVISOR

BRNO 2009

ZDE VLOŽIT ZADÁNÍ DIPLOMOVÉ PRÁCE

Abstrakt

Tato práce se zabývá problematikou modelování a simulace technologických procesů. Vychází z reálných potřeb Fakulty strojního inženýrství při VUT, kde chyběl dostatečně rozšiřitelný a otevřený simulační nástroj pro modelování procesů zpracování biomasy. V práci jsou vysvětleny základní pojmy z oboru (fyzikální) chemie, systémový přístup k modelování a simulaci procesů a řeší se zde problematika různých simulačních metod. Je popsán základní rozdíl mezi návrhovou a simulační úlohou a oddůvodněna volba sekvenčně modulárního přístupu k simulaci. V práci je detailně rozveden návrh architektury nástroje, jež klade důraz především na rozšiřitelnost budoucí aplikace. Nástroj byl implementován a v současnosti je úspěšně využíván na Fakultě strojního inženýrství.

Abstract

This thesis focuses on the problem of technological process modelling and simulation. It is based on the realistic requirements of the Faculty of Mechanical Engineering, where they lacked an extensible and open simulation tool with the ability to model biomass processing. In this work we explain the main concepts in the field of (physical) chemistry, the systematic approach to process modelling and simulation and we discuss various problems of simulation methods. We also describe the differences between simulation and design specifications and give reasons for choosing the sequential modular approach. An important part of the work is aimed at the design of the simulation tool's architecture, where the main emphasis is laid on the extensibility of future application. The tool has been implemented and it is now sucessfully being used at the Faculty of Mechanical Engineering.

Klíčová slova

bilanční rovnice, hmotnostní bilance, recykl, proces, chemicko-technologický proces, procesní inženýrství, modelování, simulace, sekvenčně modulární, simulační nástroj, simulační systém

Keywords

balance equations, material balances, recycle, process, chemical-technological process, process engineering, modelling, simulation, sequential modular, simulation tool, simulation system

Citace

Libor Skydánek: Nástroj pro modelování a simulaci technologických procesů, diplomová práce, Brno, FIT VUT v Brně, 2009

Nástroj pro modelování a simulaci technologických procesů

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Radka Kočího, Ph.D.
Další informace mi poskytli Ing. Martin Pavlas, Ph.D. a Ing. Michal Touš.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Libor Skydánek
21. května 2009

Poděkování

Děkuji tímto vedoucímu práce Ing. Radkovi Kočímu, Ph.D. za jeho trpělivost a odborné vedení, dále pak Ing. Martinu Pavlasovi, Ph.D. a Ing. Michalu Toušovi z FSI za poskytnutí materiálů a množství cenných informací, bez kterých by tato práce nikdy nemohla vzniknout. Také děkuji kolegovi Ing. Jiřímu Krajíčkovi, který se se mnou zpočátku podílel na vývoji aplikace W2E, na níž je tato práce založena.

© Libor Skydánek, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**
1.1 Použité konvence pro styly písma 4

2 Modelování technologických procesů **5**
2.1 Chemicko-technologické procesy 5
2.1.1 Klasifikace procesů 5
2.1.2 Bilance v procesech 6
2.2 Modelování technologických procesů, systémový přístup 7

3 Simulace technologických procesů **10**
3.1 Fáze řešení simulační úlohy 11
3.1.1 Analýza reálného systému 11
3.1.2 Sestavení modelu 12
3.1.3 Simulační výpočet 12
3.1.4 Vyhodnocení a kontrola výsledků simulace 13
3.2 Simulační modely 13
3.2.1 Modely proudů 13
3.2.2 Modely jednotkových operací 14
3.3 Simulační metody 14
3.3.1 Sekvenčně modulární metody 15
3.3.2 Rovnicově orientované metody 16

4 Návrh architektury simulačního nástroje **18**
4.1 Architektura Model-view-controller 18
4.2 Struktura simulačního nástroje, architektura řadiče 19
4.3 Architektura modelu 20
4.3.1 Třídy vztahující se obecně k modelu 21
4.3.2 Bloky 22
4.3.3 Proudy 22
4.4 Architektura rozhraní 24

5 Implementace modelu procesů pro zpracování biomasy **28**
5.1 Výčtové třídy pro definici vlastností modelů používaných v simulátoru . . . 28
5.1.1 Výčtové třídy pro deklaraci modelů a jejich vztahů 29
5.1.2 Výčtové třídy pro definici parametrů modelů 32
5.2 Implementace modelů bloků a proudů 34
5.2.1 Třída AbstractBlock 36
5.2.2 Třídy implementující konkrétní bloky 37

5.2.3 Třída AbstractFlow . . . 38
5.2.4 Třídy implementující konkrétní proudy . . . 39
5.3 Správa modelu a simulace . . . 40
5.3.1 Třída ModelExecution pro řízení simulace . . . 40
5.3.2 Třída ModelController, správce modelu . . . 41

6 Grafické uživatelské rozhraní simulátoru 44
6.1 Hlavní okno aplikace, třída MainFrame . . . 45
6.2 Plátno . . . 47
6.2.1 Grafické objekty plátna . . . 49
6.2.2 Správce plátna . . . 52
6.3 Postranní panely . . . 54
6.3.1 Panel pro vkládání . . . 54
6.3.2 Panely pro úpravu modelů . . . 54
6.4 Správa konfigurace rozhraní . . . 55
6.5 Další komponenty rozhraní . . . 56
6.5.1 Výpočetní formulář pro tuhá paliva . . . 57
6.5.2 Třída ViewController . . . 57

7 Závěr 60

Seznam použitých zkratek a symbolů 62

Kapitola 1

Úvod

Současný trend prudkého růstu výpočetního výkonu pracovních stanic i domácích počítačů má mimo jiné za důsledek neustálý vývoj složitějších a realističtějších modelovacích a simulačních nástrojů. Velkou výzvou v tomto oboru bylo již od jeho počátků modelování různých aspektů reálného světa. V mnoha odvětvích lidské činnosti je totiž z rozličných důvodů (finance, čas, bezpečnost) nemožné dělat pokusy přímo v reálném světě, tudíž je potřeba si tento svět (resp. jeho část) namodelovat a pokusy provádět ve fiktivním prostředí. Bohužel je fyzikální svět velmi komplexní a i velmi zjednodušené a specializované modely vyžadují výpočetně velmi silné stroje, proto nebylo dříve vůbec možné některé simulace provádět.

Na Fakultě strojního inženýrství (FSI) při VUT v Brně se před několika lety zrodila potřeba nástroje pro modelování a simulaci chemicko-technologických procesů v odvětví zpracování biomasy. Přestože moderní doba vývoji simulačních nástrojů přeje a v současnosti jsou na trhu dostupné desítky kvalitních (zpravidla komerčních) produktů, ukázalo se, že žádný z nich nesplňuje požadavky pracovníků FSI. Uzavřenost kódu a univerzalita komerčních simulačních nástrojů totiž značně omezuje (ne-li přímo vylučuje) jejich rozšiřitelnost a tím i možnost experimentování s netradičními přístupy. Proto byl v rámci grantu Ministerstva školství, mládeže a tělovýchovy České republiky zahájen vývoj vlastního simulačního nástroje s názvem Waste to Energy (W2E).

Tato práce se zabývá návrhem a implementací zmíněného simulačního nástroje. V první kapitole za úvodem (kap. 2) je čtenář uveden do problematiky chemicko-technologických procesů a seznámí se s základními principy modelování těchto procesů. Další kapitola (kap. 3) se zabývá obecnou teorií simulace technologických procesů a výběrem vhodných metod simulace pro navrhovaný software. Následující kapitola (kap. 4) již přistupuje k návrhu architektury pro modelování a simulaci technologických procesů. Hlavním cílem je dostatečná obecnost návrhu, aby jej bylo možné implementovat v libovolném odvětví procesního inženýrství. V dalších dvou kapitolách (kap. 5 a 6) je nastíněna tvorba implementace nástroje W2E, jež je praktickým výsledkem této práce.

Vývoj nástroje W2E probíhá ve skutečnosti již necelé dva roky a kromě autora této práce se na něm podíleli tito pracovníci: Ing. Jiří Krajíček (v současnosti student doktorského studia na FIT), Ing. Michal Touš (student doktorského studia na FSI), Ing. Martin Pavlas, Ph.D. (pracovník FSI, iniciátor celého projektu) a Ing. Radek Kočí, Ph.D. (pracovník FIT, vedoucí mé diplomové práce). Tato diplomová práce navazuje na můj semestrální projekt, který řešil tematiku prvních čtyř kapitol této práce, z těch byly dvě převzaty pouze s malými korekčními úpravami (kap. 2 a 3) a jedna byla upravena a rozšířena (kap. 4).

1.1 Použité konvence pro styly písma

Pro lepší orientaci v textu uvedu některé základní konvence, které jsou v této práci dodržovány:

- většina textu je psána základním proporciálním písmem
- *kurzívou* jsou uváděny důležité pojmy v textu a používá se též k zvýraznění dlouhých citací
- **tučné písmo** obvykle zvýrazňuje hlavní pojmy v různých výčtech
- `strojové písmo` je používáno pro úseky zdrojového kódu a pro názvy tříd a metod v jazyce Java
- ***tučná kurzíva*** označuje veškeré odkazy na balíčky jazyku Java

Kapitola 2

Modelování technologických procesů

Při návrhu simulačního nástroje je velmi důležité dokonalé pochopení aplikační domény tohoto nástroje. Bez zevrubné znalosti aplikační oblasti je prakticky nemožné sestrojit simulátor, který by daný problém namodeloval skutečně realisticky a jehož výstupy by odborníci mohli brát vážně. Proto se v této kapitole budeme nejprve zabývat obecnou teorií chemicko-technologických procesů a následně metodologií modelování těchto dějů.

2.1 Chemicko-technologické procesy

Proces [5] je obecně jakákoliv činnost nebo posloupnost činností, jejichž vykonáním je možné splnit určitý cíl. Pro potřeby tohoto textu budeme vždy mluvit pouze o *chemicko-technologických procesech*, což jsou takové činnosti, které způsobují chemické nebo fyzikální změny v látkách či směsích látek. Hmota, která vstupuje do procesu, se nazývá *vstup (vstupní látka, v originále input, feed)*, naopak hmota, která z procesu vystupuje, je označována jako *výstup (výstupní látka, produkt, v originále output, product)*. Procesy se obvykle skládají z několika kroků, každý krok je vykonán v tzv. *jednotkové operaci (process unit)*, přičemž každá jednotková operace má k sobě přiřazenu množinu vstupních a výstupních *proudů (process streams)*. Každý proces je možné úplně charakterizovat kombinací technologického schématu (flowsheet) a specifikací jednotlivých jednotkových operací (což mohou být reaktory, separátory, výměníky tepla) a souvisejících řídících proměnných.

2.1.1 Klasifikace procesů

Chemické procesy je možné rozdělit do několika kategorií (viz [5]). Podle přívodu vstupních látek se dělí na *vsádkové (batch)*, *kontinuální (continuous)* a *semikontinuální (semi-batch)*, podle průběhu reakce pak na procesy *ustálené (steady state)* a *neustálené (transient, unsteady-state)*.

- **Vsádkový proces**: vstupní látky se na počátku procesu zavedou do uzavřeného prostředí a po určité době (na konci procesu) je všechen obsah vyňat. Mezi vložením a vyjmutím látek nepřekročí žádná hmota hranice systému. Příkladem by mohlo být rychlé nahromadění reaktantů v nádrži, kde dojde k (bouřlivé) reakci, po jejímž ustálení (když nastane rovnovážný stav) vyjmeme produkty a nespotřebované reaktanty.

- **Kontinuální proces**: vstupní i výstupní látky průběžně proudí systémem po celou dobu procesu. Příkladem je proces destilace (např. alkoholu), kdy přivádíme konstantní rychlostí směs tekutin do destilační kolony a zároveň neustále odebíráme proudy produktů svrchu i zespodu kolony.

- **Semikontinuální proces** je takový proces, který nelze zařadit do ani jedné z předchozích kategorií. Příkladem tohoto typu procesu je např. proražení kontejneru se stlačeným vzduchem, jehož obsah začne následně unikat do atmosféry.

Pokud během procesu nedochází k žádným změnám hodnot proměnných (jako jsou teploty, tlaky, objemy, průtoky) s výjimkou drobného kolísání kolem konstantní průměrné hodnoty, říkáme o procesu, že je *ustálený*. Pokud se s časem mění libovolná proměnná, mluvíme o *neustáleném* procesu. Svojí povahou patří vsádkové a semikontinuální procesy do kategorie neustálených procesů, kdežto kontinuální procesy mohou být jak ustálené, tak neustálené.

Vsádkové zpracování se obvykle používá v případě, kdy je potřeba vytvořit malé množství produktu během jediné příležitosti, kontinuální procesy jsou naopak vhodné při potřebě vysoké produkce. Kontinuální procesy obvykle provozujeme co nejblíže k ustálenému stavu; podmínky neustáleného stavu zpravidla existují v průběhu startovní fáze procesu a také vznikají důsledkem (úmyslných) změn v operačních podmínkách procesu.

2.1.2 Bilance v procesech

Pokud chceme navrhnout nový proces, je důležité brát v úvahu některá omezení vycházející z přírodních zákonů. Žádný chemicko-technologický proces nemůže nikdy porušovat zákony zachování hmoty a energie. Z tohoto důvodu hrají tyto zákony zásadní roli při výpočtech potřebných pro simulaci procesů. Jak si později ukážeme, téma rovnováhy procesů velmi těsně souvisí s dříve uvedenou klasifikací procesů [5].

Zákon zachování hmoty jasně říká, že hmota nemůže být vytvořena ani zničena. (Neuvažujeme téměř infinitesimální přeměny hmoty na energii spojené s chemickými reakcemi.) Tvrzení založená na zákonu zachování hmoty, jako například „celková hmota na vstupu = celková hmota na výstupu“, jsou příklady tzv. *hmotnostních bilancí (mass balances, material balances)*. Návrh procesu není úplný, dokud není ověřeno, že všechny vstupy a výstupy celého procesu a všechny vstupy a výstupy jeho jednotkových operací splňují bilanční rovnice.

Obecná bilanční rovnice, která platí pro libovolnou ohraničenou veličinu (celková hmota, hmota určitého druhu, energie, hybnost) v nějakém systému (jednotková operace, množina jednotkových operací, nebo celý proces) může být zapsána takto:

vstup	+	*produkce*	−	*výstup*	−	*spotřeba*	=	*akumulace*
(vstupuje do systému)		(vyprodukováno systémem)		(opouští systém)		(spotřebováno systémem)		(nahromaděno v systému)

Rozlišujeme dva druhy bilancí:

1. **Diferenciální bilance** vyjadřují, co se děje v systému v určitém časovém okamžiku. Hodnoty bilancí jsou v tomto případě míry bilancovaných veličin udávané jako množství za jednotku času. Tyto bilance se obvykle používají ve spojení s kontinuálními procesy.

2. **Integrální bilance** popisují, co se děje v systému mezi dvěma časovými okamžiky. Hodnoty bilancí odpovídají množství bilancovaných veličin a zapisují se tedy přímo s jednotkou dané veličiny. Tento typ bilancí je běžně užíván u vsádkových procesů, počáteční čas se nastaví na moment těsně po vložení vstupů a koncový čas na okamžik těsně před vyjmutím produktů.

Pro potřeby simulace technologických procesů jsou zajímavé hlavně diferenciální bilance aplikované na kontinuální ustálené systémy. Existuje několik pravidel, které je možné použít ke zjednodušení látkových bilančních rovnic:

- *Pokud zapisujeme bilanční rovnici pro celkovou hmotu, položíme produkci i spotřebu rovnu nule.* S výjimkou jaderných reakcí není možné hmotu vytvářet ani ničit.
- *Pokud zapisujeme bilanční rovnici pro nereaktivní látku (nejde ani o reaktant ani produkt reakce), položíme produkci i spotřebu rovnu nule.*
- *Pokud je systém v ustáleném stavu, můžeme položit akumulaci rovnu nule nezávisle na tom, co bilancujeme.* Přímo z definice ustáleného systému vychází, že se žádné množství nemění.

Předchozí pravidla nám umožňují v některých případech značně zjednodušit uvedené bilanční rovnice. V typickém případě, kdy budeme simulovat kontinuální ustálené procesy, můžeme z rovnice bez obav odstranit akumulaci, tedy:

$$vstup + produkce = výstup + spotřeba$$

Při zápisu bilance nereaktivních látek nebo celkové hmoty můžeme navíc položit produkci a spotřebu rovnu nule, tedy vznikne velmi jednoduchý vztah $vstup = výstup$.

Druhým velmi důležitým zákonem, který hraje v bilančních výpočtech velkou roli, je *zákon zachování energie*, ze kterého vyplývá, že energii nelze vyrobit ani zničit, lze ji pouze přeměnit na jiný druh energie. Celková energie systému má tři složky:

1. **Kinetická energie**: Energie v důsledku translačního pohybu systému jako celku vzhledem k jiné vztažné soustavě (obvykle k povrchu země), nebo rotace systému kolem nějaké osy.
2. **Potenciální (polohová) energie**: Energie v důsledku polohy systému v nějakém potenciálovém poli.
3. **Vnitřní energie**: Energie všech částic, z nichž se těleso skládá.

Pokud uvažujeme uzavřený systém, tak mezi takovým systémem a okolím může být energie přenesena dvěma způsoby: v podobě tepla nebo práce. Přeměny energií a jejich přechody mezi jednotlivými systémy, stejně tak jako energetické bilanční rovnice, se řídí termodynamickými zákony. Detaily si může zájemce vyhledat v příslušné literatuře, pro potřeby této publikace jsou zmíněné základy postačující.

2.2 Modelování technologických procesů, systémový přístup

Při řešení technologických problémů se velmi často setkáváme s faktem, že není možné zabývat se pouze dílčím dějem (procesem, aparátem), ale je nutné brát v úvahu součinnost větších celků, jakými jsou skupiny aparátů, vzájemně se ovlivňujících procesů apod.

Přímým důsledkem je vzrůstající složitost problému. Metody vhodné pro modelování a predikci chování jednotlivých aparátů nebo procesů nejsou vhodné pro aplikaci na rozsáhlejší technologie, a je proto nutné používat odlišnou metodiku. Fakta uvedená v této části jsou čerpána z [9] s výjimkou míst, kde jsou uvedeny jiné citace.

Nejprve si definujeme pojem *systém*, abychom měli v dalším textu na čem stavět. Předem podotýkám, že se zde nebudu pokoušet o sestavení obecné definice systému, nýbrž o co nejvhodnější definici v rámci tématu technologických procesů, jak je uvedeno v [9]. Z běžného každodenního pozorování jsme schopni vyvodit, že svět kolem nás není homogenní, lze v něm rozpoznat různé vzájemně odlišitelné *objekty*, jejich vzájemné vztahy, interakce a závislosti. Objektem rozumíme nejen hmatatelné předměty kolem nás, ale i abstraktní celky. Každý objekt reálného světa je nutné nějak charakterizovat, abychom s ním mohli dále pracovat. Jednak musíme být schopni odlišit od sebe dva různé objekty, na druhé straně potřebujeme rozeznat, když mají dva nebo více objektů řadu vlastností shodných a lze je zařadit do stejné třídy objektů. Od pojmu objekt se dostaneme k pojmu systém tak, že při studiu objektu se neomezíme jen na jeho vnější znaky, ale budeme jej chápat jako složený, s určitou vnitřní strukturou a rozpoznatelnými částmi, které jsou také objekty ale na nižší hierarchické úrovni. Vhodná definice systému (podle [9]) zní: „*Systém je komplexní objekt skládající se z jedné nebo více úrovní objektů, které se mohou vzájemně ovlivňovat.*“

Objekty, ze kterých se systém skládá, je možné označit jako *podsystémy*. Mohou to být také systémy nebo jen určité elementární objekty, jejichž vnitřní struktura pro nás není podstatná. To, zda budeme objekt chápat jako systém nebo elementární objekt, je závislé hlavně na cíli našeho zkoumání objektu. V určitém kontextu mohou být detaily objektu důležité, tudíž jej namodelujeme jako samostatný systém (či podsystém), jindy (zpravidla při pohledu na větší celek) však ponecháme strukturu objektu schovanou a budeme se k němu chovat jako k elementárnímu stavebnímu prvku. (V počítačovém odvětví používáme pro takovéto skrývání detailů pojmu *zapouzdření*, často se také o objektu, jehož struktura je pro nás neznámá nebo nezajímavá, hovoří jako o *černé schránce*.) Strukturu složeného systému můžeme typicky popsat grafem (uzly jsou elementární podsystémy, hrany představují vzájemné interakce).

Zavedení pojmů systém a podsystém je nutné pro vytváření modelů jako náhrady nebo obrazu reálných systémů, kterým pak říkáme *originál*. Při tvorbě modelů se používají kroky nazývané *formalizace*, *abstrakce* a *generalizace* (*zobecňování*). Existuje mnoho různých forem modelů, pro naše účely budou podstatné modely matematické a modely v podobě programového kódu.

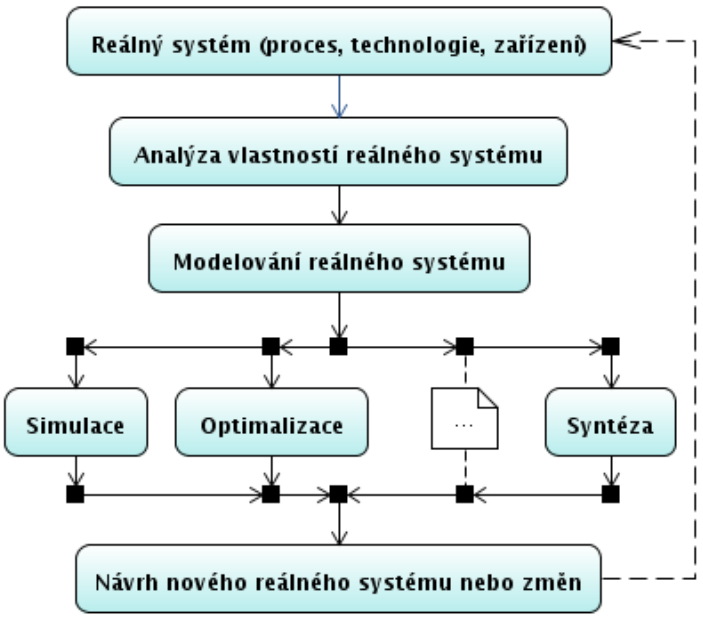
Systémové inženýrství dělí matematické modely podle různých hledisek souvisejících s použitým aparátem pro tvorbu modelu i s oblastí jejich použití. Při výběru modelu musíme mít nejprve jasno, jaké vlastnosti má originál, a které z nich jsou pro nás podstatné.

- Podle předvídatelnosti modelovaného děje jsou modely
  - ▷ deterministické
  - ▷ nedeterministické
- Podle chování v čase jsou modely
  - ▷ stacionární (statické)
  - ▷ dynamické (tranzientní)
- Podle matematického aparátu jsou modely reprezentované

- ▷ systémem lineárních algebraických rovnic (LAR, angl. LAE)
- ▷ systémem nelineárních algebraických (transcendentních) rovnic (NAR, angl. NE nebo NAE)
- ▷ systémem obyčejných diferenciálních rovnic (ODR, angl. ODE), resp. kombinovaným systémem diferenciálních a algebraických rovnic (DAR, angl. DAE)
- ▷ systémem parciálních diferenciálních rovnic (PDR, angl. PDE), resp. kombinovaným systémem společně s NAR nebo ODR

- Podle stupně detailnosti modelování rozlišujeme modely
  - ▷ rigorózní
  - ▷ přibližné, aproximativní

Obecně lze říci, že všechny modely, které studujeme v základních předmětech, jakými jsou fyzika, fyzikální chemie a chemické inženýrství, jsou formálně chápány jako deterministické, tedy za shodně definovaných podmínek se sledovaný systém chová vždy stejně. Pro potřeby modelování technologických procesů budeme uvažovat pouze systémy, které se v čase nijak nemění, tedy v celé práci budeme mluvit o stacionárních modelech. K jejich popisu se typicky používá systém NAR.



Obrázek 2.1: Fáze modelování technologického procesu (převzato z [9])

Systémový přístup je charakterizován tím, že se snažíme chápat realitu nebo její vymezenou část jako systém a získané poznatky uplatníme ve fázi modelování, na kterou pak navazují fáze vytváření a zpracování modelu. Celý cyklus je vidět na obr. 2.1. Z tohoto obrázku je jasně patrné, že modelování a simulace technologických procesů jsou koncipovány jako cyklus: s pomocí analýzy stávajícího reálného systému vytvoříme model, jehož simulací a úpravami dojdeme nakonec k návrhu nové technologie, která má požadované konstrukční i operační parametry.

Kapitola 3

Simulace technologických procesů

Jak bylo zmíněno na konci předchozí kapitoly, budeme se zabývat pouze stacionární simulací. Proces simulace je obvykle definován [6] jako metoda získávání nových znalostí o systému prostřednictvím experimentování s jeho modelem. Jinou definici, která do simulace zahrnuje i pojem modelování, najdeme v [11]: „*Simulace je reprezentace reálného děje matematickým modelem, který po vyřešení dává informaci o vnitřním chodu tohoto děje.*“

Z pohledu technologických procesů můžeme na simulaci také pohlížet jako na zobecnění bilančních výpočtů (kap. 2.1.2), jež jsou považovány za jeden z pilířů chemického inženýrství. Do simulace se promítlo i používání bilančních schémat, kde jsou hlavními stavebními prvky *uzly*, resp. *jednotkové operace* a *proudy*. Na rozdíl od bilančních modelů, které umožňují výpočet neznámých (tj. neměřených) veličin pouze tehdy, když je k dispozici dostatečné množství měřených dat, jsou však simulační modely doplněny o popisy fázových rovnováh, transportní rovnice a další vztahy. Podobně modely proudů chápané jako vektory jejich parametrů jsou doplněny o parametry, jako je tlak, teplota, podíl parní fáze apod. V důsledku toho vyžaduje simulace podstatně méně vstupních údajů. Vliv bilančních výpočtů na simulaci se projevuje i v terminologii, kde se rozlišují dva typy výpočtů (viz [9] a [5]):

- **Simulační výpočet** – cílem výpočtu je získat neznámé parametry vnitřních a výstupních proudů
- **Návrhový výpočet** – cílem výpočtů mohou být kromě vybraných parametrů vnitřních a výstupních proudů také některé parametry zařízení v procesu a neznámé parametry vstupních proudů (naopak jsou však známé parametry výstupních proudů)

Různé metody simulace mají odlišné schopnosti provádění simulačních a návrhových výpočtů, detailně si tuto problematiku rozebereme v závěru kapitoly, nyní si charakterizujeme podobu *obecné simulační úlohy* chemicko-technologického procesu:

- Jsou zadány povinné a volitelné vstupní údaje. Mezi povinné patří například topologie (struktura) procesu, systém uvažovaných chemických položek, fyzikálně chemický popis. Mezi volitelnými údaji jsou hodnoty vybraných parametrů proudů a jednotkových operací a podle potřeby různé dodatečné podmínky.
- Je dán cíl výpočtu – všechny nebo vybrané parametry jednotkových operací či proudů, které nebyly zadány.

3.1 Fáze řešení simulační úlohy

Na obrázku 2.1 byl uveden diagram, který zobrazuje obecný postup při modelování technologických procesů. Rozeberme si v této části podstatu jednotlivých fází (podle [9]).

3.1.1 Analýza reálného systému

Ideálním výsledkem simulace je dokonalá shoda vypočtených parametrů procesu s realitou. Je jasné, že tento stav není dosažitelný, ale právě proces analýzy reálného systému by měl zabránit hrubému zkreslení skutečnosti špatně definovaným modelem. Důležité je shromáždění maxima informací různého charakteru o simulovaném systému, i kdyby se tyto údaje neměly do simulační úlohy vůbec promítnout, jelikož právě nevyužité údaje můžeme následně použít ke kontrole výsledků simulace.

Prodívejme se nyní, které aspekty systému jsou pro nás při analýze a modelování systému podstatné.

- **Jednotkové operace:** Pojem jednotkové operace je relativní (jak bylo zmíněno již několikrát v předchozí kapitole). Vymezení jednotkových operací v systému je velmi důležité, ale podřízené účelu simulace. V simulačních programech existuje obvykle množina jednotkových modulů, které představují jednotlivé jednotkové operace. Ty nemusí nutně vyjadřovat jen modely různých jednotkových operací, často může jít i o stejné modely na různých vrstvách abstrakce. Některá zařízení jsou obvykle pro konkrétní simulační účel zbytečná, například ve stacionární simulaci je naprosto zbytečné uvažovat zásobníky, proto tento typ zařízení nebudeme v simulačním schématu vůbec uvažovat. Také různá paralelní a záložní zařízení, regulační prvky a pomocné komponenty mají v simulaci jen málokdy nějaký význam.

- **Topologie systému:** Dalším krokem je propojení jednotkových operací vymezených v předchozím kroku. Základem je prozkoumání potrubní sítě ve stávající technologii nebo představa o součinnosti jednotlivých jednotkových operací v plánovaném provozu. V prvním případě je opět nutná redukce, v praxi se totiž potrubí obvykle zdvojují a existují různá speciální potrubí pro najíždění, odstavování a nouzové operace, jež nejsou pro simulaci podstatná. Naopak je nutné započítat veškeré energetické a tepelné proudy v systému, stejně jako konanou práci. Prostup tepla se označuje jako tepelný proud.

- **Chemické složky:** Každá chemická výroba je spojena s určitým spektrem chemických složek vystupujících v technologii jako suroviny, produkty, meziprodukty nebo pomocné látky. Při analýze je klíčová detekce prvků, které jsou pro simulaci podstatné. Nezahrnutí těchto prvků a následné hledání chyby bývá nesmírně nepříjemné. Pokud se v systému vyskytují směsi po chemicko-fyzikální stránce velmi příbuzných složek, bývá výhodné je chápat jako jednu složku - tzv. *pseudosložku*. Toto zjednodušení má však svá pravidla a simulační programy by měly být schopny umožnit pracovat s těmito směsmi jako se zvláštním případem.

- **Fyzikálně-chemický popis:** Bez řádného a realistického popisu fyzikálně-chemického chování jednotlivých složek i jejich směsí nemá simulační výpočet valnou cenu. Simulační programy mají obvykle zabudovanou rozsáhlou knihovnu fyzikálně-chemických metod, i přesto však není vždy jednoduché vybrat ten správný popis, který by

zajistil shodu s realitou. Při analýze je zejména potřeba věnovat pozornost neideálnímu chování v různých fázích, výskytu nemísitelných nebo částečně mísitelných kapalných fází, přítomnosti polymeračních procesů nebo spontánních chemických reakcí apod. Pro řadu metod, i když regulérně zařazených v knihovně fyzikálně-chemických metod simulačního programu, je nutno dodat sadu interakčních koeficientů. To je typické u rovnic pro výpočet aktivitních koeficientů a stavových rovnic. Ve fázi analýzy je často žádoucí se zaměřit na vyhledávání nebo naměření experimentálních dat týkajících se fyzikálně-chemického chování jednotlivých složek a jejich směsí.

- **Data:** Bylo již zmíněno, že simulační výpočet vyžaduje ve srovnání s bilančními výpočty mnohem méně údajů, přesto je nutno ve fázi analýzy věnovat datům a jejich zdrojům patřičnou pozornost. Veškerá data, která nevyužijeme při simulaci, je možné využít pro kontrolu výsledků simulace. Základními zdroji dat jsou:
  - ▷ měření na reálných provozních zařízeních
  - ▷ technologické specifikace
  - ▷ konstrukční údaje o provozních zařízeních – technická dokumentace
  - ▷ odborná literatura

3.1.2 Sestavení modelu

Moderní simulační programy tuto fázi v maximální míře ulehčují a přejímají velkou část zodpovědnosti z beder uživatele. Simulační program by měl obsahovat dostatečně velkou knihovnu jednotkových modulů, jež modelují určité jednotkové operace. Je také vhodné (ale u současných simulačních programů velmi vzácné), aby bylo možné snadno vytvářet a přidávat nové jednotkové moduly. Úkolem uživatele je vybrat co nejvhodnějšího reprezentanta pro každý uzel v simulovaném procesu, což není vždy jednoduchý úkol.

Dále je nutné uzly (= jednotkové moduly) správně propojit pomocí technologických proudů, simulační program by měl být při tomto úkolu také maximálně nápomocen. Za správné pořadí provádění operací pak zpravidla přebírá veškerou odpovědnost simulační program, resp. jeho část pověřená řízením simulačního výpočtu. V této fázi je také velmi klíčová detekce recyklů, které budou šířeji rozebrány v části o sekvenčně modulárních metodách (kap. 3.3.1).

Simulační model je dále ovlivněn volbou fyzikálně chemického popisu. Simulační program obvykle poskytuje dostatečně širokou paletu modelů, ze kterých si pak uživatel vhodný popis zvolí. Je nutné brát v úvahu všechna fakta zjištěná ve fázi analýzy, aby se modelovaný systém blížil co nejvíce skutečnosti.

3.1.3 Simulační výpočet

Vlastní výpočet provádí simulační program zpravidla autonomně, aniž by uživatel do zpracování nějak výrazně zasahoval (tak je možné provádět velké množství simulačních kroků v okamžitém sledu). Výpočet musí být schopen určit správné pořadí zpracování jednotlivých uzlů, musí se umět vypořádat s recyklovými proudy, pokud jsou tyto detekovány.

Výstupem simulačního výpočtu bývá informace o zpracovaných uzlech, počtu iterací, hodnotách konvergenčních kritérií apod. Většina programů nějakým způsobem animuje prováděnou simulaci, aby měl uživatel přehled, ve které části topologie se výpočet právě nalézá (toto je ovšem závislé na rychlosti simulace, často můžou jednotlivé kroky simulace probíhat příliš rychle, než aby to bylo možné zobrazit).

3.1.4 Vyhodnocení a kontrola výsledků simulace

Program by měl poskytovat prostředky k přehlednému zobrazení výsledků simulace. Základním výstupem jsou tabulky s přehledem parametrů proudů a formuláře s parametry jednotkových operací, které zahrnují zadané parametry i některé vypočtené (většinou se jedná o parametry závislé na vypočtených parametrech proudů). Moderní simulační programy dále poskytují grafické služby, které se používají například pro zobrazení profilů průtoků, teploty, tlaku a koncentrací.

3.2 Simulační modely

V průběhu kapitoly bylo opakovaně zdůrazněno, že rozklad složitého procesu na systém jednotkových operací je prakticky nezbytným předpokladem pro uskutečnitelnost simulace. Modely jednotkových operací a modely proudů tvoří základní stavební kameny každého simulačního modelu. V této části je hlavním literárním zdrojem [9].

Při modelování používáme obecné pojmy *modelové proměnné* a *modelové rovnice* (případně další typy vztahů – nerovnice apod.). Model se tedy skládá z jednotlivých rovnic, ve kterých vystupují různé modelové proměnné. V simulačních modelech představují modelové rovnice zpravidla vztahy mezi vstupními a výstupními proudy jednotkových operací a vztahy popisující vnitřní děje v jednotkových operacích. Modelové proměnné simulačního modelu můžeme rozdělit do dvou základních kategorií:

- parametry proudů
- parametry jednotkových operací

Pro každý proud v procesu a pro každou jednotkovou operaci je definován vektor těchto parametrů, který může být pro některé jednotkové operace (v jednodušších modelech) i prázdný. Vektor parametrů proudu nazýváme také *modelem proudu*.

3.2.1 Modely proudů

V simulačních programech obvykle podle potřeby používáme a rozlišujeme následující typy proudů (a tím i strukturu jejich parametrů):

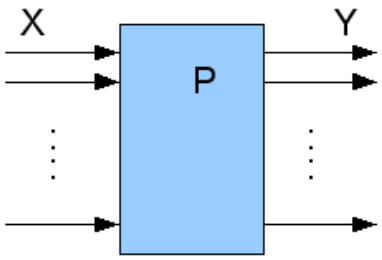
- **Materiálové proudy:** Představují hmotnostní, materiálové toky mezi jednotkovými operacemi. Zpravidla se používá popis systémem základních parametrů proudu, přičemž kromě povinných parametrů (typicky teplota, tlak, celkový průtok, složení, podíly různých fází) bývá možné zadat i další volitelné parametry vhodné pro danou simulační úlohu. Často je vhodné zabudovat do programu různé šablony (typy) proudů, které rozdělují proudy do několika skupin podle složení a podílů fází. Simulační program by měl pak sám spravovat povolené hodnoty atributů v závislosti na šabloně.
- **Tepelné nebo čistě energetické proudy:** Když potřebujeme v simulačním modelu popsat tok tepla (zprostředkovaný např. prostupem tepla), je možno použít tento druh proudu. Jediným parametrem tohoto proudu je obvykle teplo, celková entalpie[1] nebo jejich tok.

[1] Entalpie je fyzikální veličina označovaná písmenem H udávaná v joulech, která vyjadřuje tepelnou energii uloženou v jednotkovém množství látky. Jedná se o jeden ze čtyřech základních termodynamických potenciálů [2].

- **Proudy představující konanou práci:** Vztah mezi dvěma jednotkovými operacemi může být reprezentován i konanou prací, která pak obvykle vystupuje jako aditivní člen v entalpických (energetických) bilancích v obou uzlech.

3.2.2 Modely jednotkových operací

Jednoduché schéma modelu jednotkové operace je znázorněno na obrázku 3.1. Vektor X představuje souhrn všech parametrů všech proudů do uzlu vstupujících, vektor Y pak souhrn všech parametrů všech proudů z uzlu vystupujících a vektor P souhrn všech vnitřních parametrů jednotkové operace.



Obrázek 3.1: Schéma jednotkové operace

Z matematického hlediska je obvykle jednotková operace chápána jako transformace vstupních proudů na výstupní, což odpovídá základnímu typu modelu:

$$Y = F(X, P) \tag{3.1}$$

Vzhledem k tomu, že by měl být vektor Y dopočítáván neiteračně (alespoň z vnějšího pohledu), je tento typ modelů nazýván *explicitním modelem jednotkové operace*. Bohužel většina jednotkových operací má podstatně složitější strukturu a pouze v ojedinělých případech lze sestavit modelové rovnice analyticky ve tvaru 3.1. Obecnějším typem modelu je model ve tvaru

$$R(X, Y, P) = 0 \tag{3.2}$$

kterému říkáme *implicitní model jednotkové operace*. Znamená to, že v tomto případě je model reprezentován obecně systémem NAR, která je prakticky vždy řešen numerickými iteračními metodami. Jestliže pro určitou jednotkovou operaci nelze sestavit explicitní model, ale máme pouze model implicitní, je možné použít tzv. *semiexplicitní model*, který se navenek chová stejně jako explicitní model, ale výpočet výstupu Y je uskutečněn numericky. Mluvíme v tomto případě o *vnitřní iteraci*.

3.3 Simulační metody

Simulace technologických procesů je zdánlivě jednoduchou záležitostí. Z množiny jednotkových operací vybereme vhodné reprezentanty pro namodelování daného procesu, spojením dílčích modelů do jediné soustavy rovnic vytvoříme simulační model a soustavu vyřešíme patřičnou numerickou metodou. Ve skutečnosti již pro jednoduché procesy mohou dimenze takovýchto soustav narůstat na tisíce až deseti tisíce rovnic i více a ani současná špičková výpočetní technika neposkytuje možnosti v plné podobě tyto soustavy řešit (viz [9]). Proto

již v od 60. letech vznikly dva přístupy automatizovaného řešení technologických procesů: jsou to *sekvenčně modulární* a *rovnicově orientované metody*.

3.3.1 Sekvenčně modulární metody

Tyto metody jsou založené na explicitních modelech a jsou tou nejjednodušší odpovědí na otázku počítačového řešení simulace procesů. Používají osvědčenou filozofii: Pokud je problém příliš složitý, dekomponuj ho i za cenu vzniku iterační smyčky. Základní princip je následující (podle [9] a [5]):

- ▷ Máme schéma jednotkových operací propojených proudy znázorněné pomocí slabě souvislého orientovaného grafu. Operační jednotky vystupují v grafu jako uzly, proudy jako hrany. Každý proud se bere jako orientovaná hrana od výstupu (výstupního konektoru) jednoho uzlu do vstupu (vstupního konektoru) jiného uzlu.

- ▷ Hrany slouží pouze k propojení uzlů, při výpočtu se hrana chová tak, že zobrazuje hodnoty z výstupu zdrojového uzlu přímo na vstup cílového. Každý uzel, který má na vstupu hodnoty z jiného uzlu, který ještě nebyl zpracován, má vstupní hodnoty nedostupné.

- ▷ V uzlech probíhají výpočty. Každý uzel, který má dostupné všechny vstupní hodnoty, může být zpracován, což spočívá ve vyčíslení parametrů všech jeho výstupních proudů na základě okamžitých hodnot parametrů jeho vstupních proudů a hodnot jeho vnitřních parametrů (vždy pevně zadaných).

- ▷ Během simulace mluvíme o *krocích* a *iteracích*. Jedním krokem simulace je zpracování právě jednoho uzlu (tedy právě jedné jednotkové operace), jedna iterace odpovídá právě jednomu zpracování každého uzlu grafu. Další iterace nemůže začít, dokud aktuální neskončila.

- ▷ Po spuštění simulace dochází k postupnému zpracování uzlů (provádění simulačních kroků) tak, že z množiny v dané iteraci nezpracovaných uzlů vybereme jeden z těch, jež mají dostupné všechny vstupní hodnoty, a zpracujeme tento uzel. V praxi se tedy začíná zpravidla od uzlů, jejichž všechny vstupní proudy přicházejí z okolí procesu (tzv. *nástřiky do procesu*).

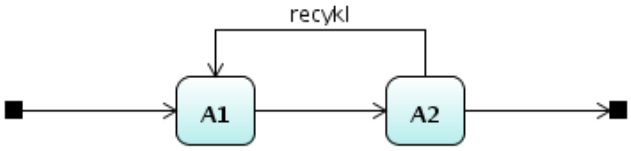
Celý výpočet pak proběhne v jediné iteraci, pokud v daném orientovaném grafu neexistuje cyklus. Bohužel, většina složitějších průmyslových procesů cykly běžně obsahuje, tudíž je vhodné, aby se s nimi simulační metoda dovedla vyrovnat. Simulační program by měl ve fázi modelování detekovat cyklus a dovolit uživateli vybrat hranu, která bude označená za tzv. *recykl* (obr. 3.2), což je pojem používaný v procesním inženýrství pro proud způsobující cyklus ve zpracování. U každého recyklu pak postupujeme následně (viz [9]):

1. Parametrům recyklového proudu přiřadíme počáteční hodnoty. Tyto hodnoty by měly být založené na odhadu skutečných hodnot tohoto proudu během procesu, jelikož na kvalitě odhadu přímo závisí úspěšnost simulace.

2. Zpočátku předstíráme rozpojení recyklového proudu, do cílového uzlu hrany zavedeme imaginární nástřikový proud, jež má hodnoty rovny počátečním hodnotám recyklu.

3. Když proběhne zpracování cílového uzlu recyklu, odstraníme imaginární nástřikový proud a obnovíme původní zapojení recyklu.

Pokud jsou v procesu přítomné recykly, nepostačí k výpočtu jedna iterace, hodnoty recyklů by měly postupně konvergovat ke správným hodnotám, ale tato konvergence není zaručena, a jak bylo zmíněno, je závislá na správnosti odhadu recyklových hodnot.



Obrázek 3.2: Příklad recyklového proudu

Sekvenčně modulární metody jsou primárně určeny k řešení tzv. *základní simulační úlohy*. Její definice zahrnuje tyto předpoklady o struktuře zadaných / vypočítávaných modelových proměnných (viz [9]):

- Jsou zadány právě všechny parametry všech proudů vstupujících do procesu z okolí (nástřiky do procesu) a všechny (nezávislé) parametry všech jednotkových operací v procesu.
- Cílem výpočtu jsou právě všechny parametry všech vnitřních a výstupních proudů z procesu.

Často se stává, že je však potřeba řešit spíše *návrhovou úlohu* (pojmy *základní simulační úloha* a *návrhová úloha* přímo souvisejí s pojmy *simulační výpočet* a *návrhový výpočet*, viz kap. 3), tedy úlohu, ve které máme zadány některé hodnoty parametrů vnitřních anebo výstupních proudů a cílem je vypočítat nejen chybějící hodnoty parametrů vnitřních a výstupních proudů, ale i požadované (chybějící) hodnoty parametrů vstupních proudů. Obecně tento problém sekvenčně modulární metody nedokáží řešit, dá se to však obejít podobným způsobem, jakým v těchto metodách řešíme recykly. Je nutné opět provést odhad chybějících hodnot parametrů vstupních proudů a tyto hodnoty použít pro inicializaci těchto proudů. Simulace je úspěšná, pokud vypočtené hodnoty konvergují k požadovaným hodnotám parametrů vnitřních a výstupních proudů. Stejně jako v případě recyklů však není konvergence zaručena a je závislá na kvalitě odhadu.

Se stoupajícím počtem recyklů a nezadaných nástřiků stoupá pravděpodobnost, že nedojde ke konvergenci simulační metody. Proto jsou sekvenčně modulární metody vhodné hlavně pro řešení základní simulační úlohy s malým počtem recyklových proudů. Simulační programy vyžadující výpočet *obecné simulační úlohy* (tj. základní simulační úlohy i návrhové úlohy) by měly používat rovnicově orientované metody.

3.3.2 Rovnicově orientované metody

Rovnicově orientované metody se od sekvenčně modulárních liší hlavně tím, že simulace neprobíhá v krocích odpovídajících zpracovávání jednotlivých jednotkových operací. V rovnicově orientovaných metodách se shromáždí všechny rovnice pro všechny jednotkové operace a řeší se najednou. To v sobě skrývá několik základních nevýhod těchto metod (z nichž mnohé již byly zmíněny v úvodu do simulačních metod):

- Soustavy rovnic mohou nabývat obrovských rozměrů, což vede k ohromným požadavkům na paměťový prostor, často nesplnitelným i pro relativně malé procesy.

- Na jednotkové operace se díváme jako na množiny jim příslušných rovnic, při výpočtech pak ovšem zanikají hranice jednotlivých jednotkových operací, což není procesním inženýrům právě příjemné. Procesní inženýři jsou zvyklí pohlížet na proces jako na diagram proudů mezi jednotkovými operacemi, což odpovídá sekvenčně modulárnímu modelu.
- Metody pro řešení vzniklých soustav jsou často velmi náročné, kladou si velké nároky na programátory i na výpočetní výkon.

Všechny tyto nevýhody zpravidla převažují hlavní výhodu rovnicově orientovaných metod, jíž je přirozenost řešení obecné simulační úlohy. Při použití rovnicově orientovaných metod totiž nezáleží na tom, jaké parametry zadáme a které se počítají, pokud je úloha dobře definovaná a daný stroj ji výpočetně zvládne, proběhne simulace vždy správně a získá správné výsledky. V budoucnosti začnou tyto metody pravděpodobně převládat, v současnosti však mají sekvenčně modulární metody navrch a i my se budeme z uvedených důvodů v této práci dále zabývat pouze jimi.

Kapitola 4

Návrh architektury simulačního nástroje

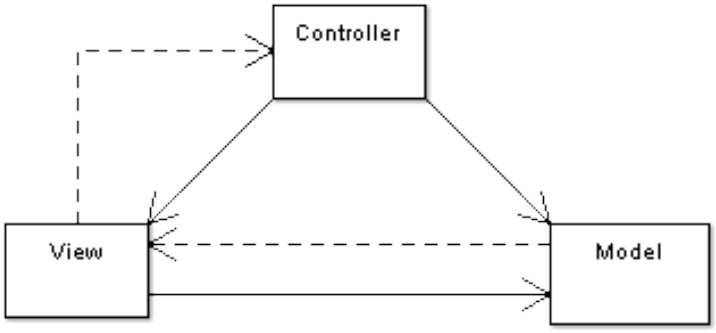
V předchozích dvou kapitolách jsme si zavedli důležité pojmy z oboru procesního inženýrství a simulace chemicko-technologických procesů a nyní se konečně dostáváme k návrhu vlastního simulátoru. Již ve fázi návrhu je vhodné brát v úvahu cílovou platformu aplikace. V zadání stojí, že implementace by měla být provedena v jazyce Java. Důvodem pro tento požadavek je výborná přenositelnost tohoto jazyka a navíc jeho snadná integrace např. do webových stránek. Navíc jde o programovací jazyk vyšší úrovně (na vysoké úrovni abstrakce), tudíž je programování v něm velmi efektivní a kvalitní návrh by měl zaručit dobrou rozšiřitelnost aplikace.

Ze zmíněných důvodu bude návrh strukturován po vzoru Javy. Většina moderních aplikací má obvykle dvě části: model a uživatelské rozhraní. První část implementuje funkčnost aplikace (modeluje implementovanou skutečnost), druhá slouží k interakci s uživatelem. Knihovna pro tvorbu grafických uživatelských rozhraní v Javě, Swing, je založena na architektonickém vzoru Model-view-controller (viz. [1]), který celkem jasně definuje způsob oddělení uživatelského rozhraní od funkčnosti. Při vývoji aplikací v Javě je vhodné se tohoto vzoru držet, proto mu bude věnována část této kapitoly.

4.1 Architektura Model-view-controller

Model-view-controller, zkráceně MVC, je softwarová architektura, která nahlíží na aplikaci jako na tři nezávislé komponenty: *model*, *view* (pohled, nebo lépe rozhraní) a *controller* (řadič) (viz [4]). Cílem je, aby modifikace jedné komponenty neměla velký vliv na ostatní. Jednotlivé komponenty hrají tyto úlohy:

- **Model** je doménově specifická reprezentace informací, s nimiž aplikace pracuje. Tato vrstva implementuje hlavně funkčnost aplikace (výpočty, přístup k datům atd.).
- **View (rozhraní)** převádí data reprezentovaná modelem do podoby vhodné k interaktivní prezentaci uživateli. Tato vrstva odpovídá grafickému uživatelskému rozhraní, reaguje na vstupní zařízení (myš, klávesnice) a využívá různé grafické metody k zobrazení výsledků na monitor nebo jiné výstupní zařízení.
- **Controller (řadič)** je pomocná vrstva, která zajišťuje komunikaci zbylých dvou vrstev a umožňuje provádění změn v modelu nebo rozhraní.



Obrázek 4.1: Schéma architektury MVC

Koncept MVC lze realizovat různě, nejde o pevně danou sadu pravidel, spíše o určitá vodítka, jak postupovat a čeho se vyvarovat. Základním pravidlem je, že model by nikdy neměl přistupovat přímo ke komponentě rozhraní, aby se aplikace nestala závislou na uživatelském rozhraní. Na obr. 4.1 jsou znázorněny obvyklé vztahy mezi komponentami: plné čáry značí přímá spojení, přerušované nepřímá (vztah pozorovatele). Vyplývá z toho, že řadič smí přímo ovlivňovat ostatní komponenty, přijatelný je také přímý přístup rozhraní k modelu, naopak model by měl mít pouze omezenou možnost informování rozhraní o změnách, stejně jako rozhraní by mělo být schopno informovat řadič. Často se komunikace modelu a rozhraní také řeší prostřednictvím řadiče, jež je pak rozdělován na část, která se orientuje na rozhraní, a na část modelovou.

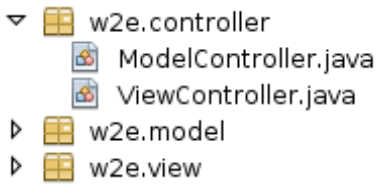
4.2 Struktura simulačního nástroje, architektura řadiče

Když máme nyní k dispozici architekturu MVC, můžeme na ní založit návrh vlastního nástroje. Nejčastěji se budu při návrhu architektury opírat o diagramy tříd, případně o diagramy balíčků, neboť jsou v této fázi návrhu nejužitečnější a hierarchie uspořádání zdrojových textů v Javě odpovídá stromové struktuře balíčků a tříd.

Na nejvyšší vrstvě abstrakce tedy vytvoříme tři balíčky odpovídající třem základním celkům aplikace: ***model***, ***view*** a ***controller***. Klíčové třídy aplikace, tedy reprezentaci dat a samotný simulační systém, umístíme do balíčku ***model***, veškeré prvky grafického rozhraní do balíčku ***view***, ***controller*** pak bude obsahovat dvě třídy: `ViewController` a `ModelController`. `ViewController` je třída odpovědná za správu grafického rozhraní, nejčastěji bude volána z modelu a jejím hlavním úkolem by mělo být volání překreslování a aktualizace různých prvků rozhraní. `ModelController` bude provádět úpravy v modelu vyvolané uživatelem z rozhraní. Schéma této základní hierarchie (bez prozatím neurčených detailů balíčků ***model*** a ***view***) je znázorněno na obr. 4.2.

Obrázek 4.2: Diagram tříd simulátoru, detail pro komponentu řadiče

Tímto máme definovanou základní strukturu simulátoru, ještě než se však pustím do návrhu modelu a rozhraní, ukáži zde diagram balíčků pro zatím navrženou část architektury. Základní balíček, který se stane kořenem stromové hierarchie balíčků, pojmenuji ***w2e*** (tento název je libovolný, zpravidla se jedná o zjednodušenou formu názvu implementace, jak je tomu i v tomto případě). Do kořenového adresáře umístíme strukturu balíčků podle MVC, tedy balíčky `controller`, `model` a `view` (viz obr. 4.3). Nyní můžeme pokračovat v detailnějším návrhu modelu a rozhraní.

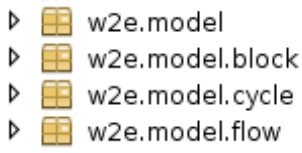


Obrázek 4.3: Diagram balíčků, základní architektura

4.3 Architektura modelu

Architektura modelové části simulátoru musí nutně vycházet ze simulačních technik rozebraných v předchozích kapitolách. V kapitole o simulaci jsem odůvodnil použití sekvenčně modulárních metod simulace (kap. 3.3), tudíž si budeme libovolný proces modelovat jako slabě souvislý orientovaný graf. Základními stavebními prvky grafu jsou hrany a uzly, v procesním inženýrství tyto odpovídají *technologickým proudům* a jednotkovým operacím. V terminologii simulátoru budu jednotkové operace uvádět jako *bloky*, neboť jde o vhodnější abstrakci – jednotkové operace lze označit za samostatné výpočetní bloky procesu a pojem blok se navíc často používá pro části velkých zařízení.

Začněme nyní s analýzou modelovaného systému s cílem navrhnout základní hierarchii modelu včetně struktury hlavních tříd. Nejprve budeme potřebovat třídu, která zapouzdří model simulačního systému jako celek, nazvěme tuto třídu jednoduše `Model` a umístíme ji přímo do balíčku ***w2e.model***. Ústřední částí systému jsou bloky a proudy, proto pro ně vytvoříme samostatné balíčky ***w2e.model.block*** a ***w2e.model.flow***, do kterých pak umístíme související třídy. Velmi charakteristickou součástí modelu jsou také cykly v grafu. Bude vhodné je identifikovat a označovat již během tvorby modelu procesu a později se tato informace bude hodit při samotné simulaci. Pro přehlednost tedy oddělíme třídy pro cykly do balíčku ***w2e.model.cycle***. Timto získáváme základní podobu diagramu balíčků pro model (viz obr. 4.4).



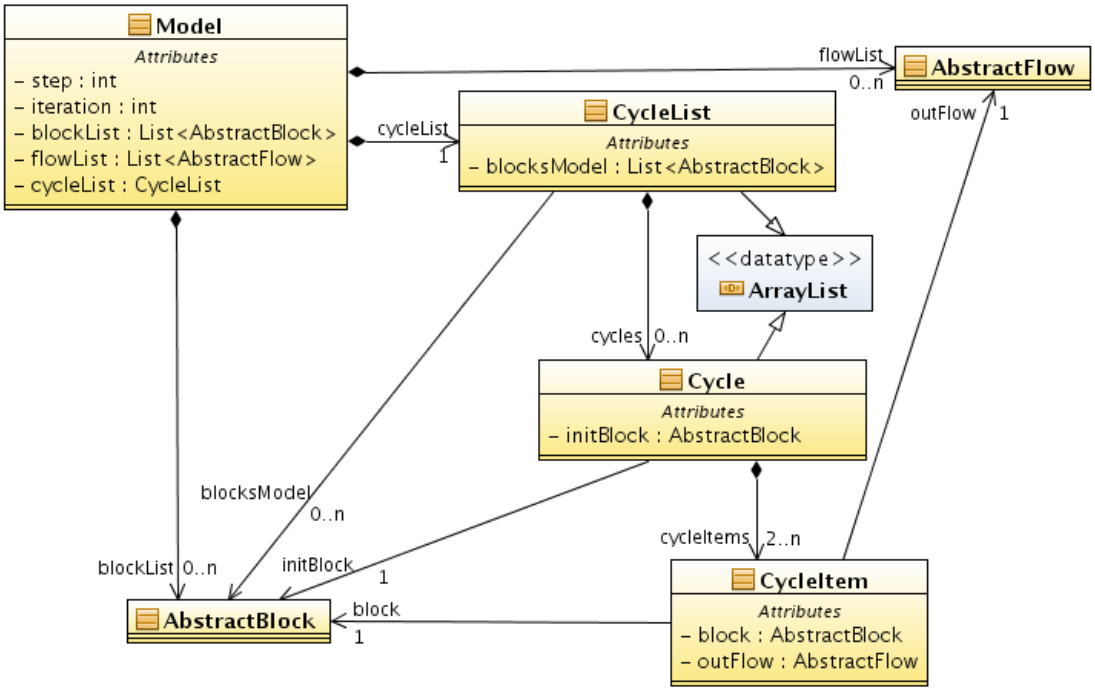
Obrázek 4.4: Diagram balíčků pro model

4.3.1 Třídy vztahující se obecně k modelu

Nejdůležitější roli hraje třída `Model`, jejímž úkolem je zapouzdřit modelovaný proces jako celek. Musí proto obsahovat *seznamy všech bloků* a *proudů* v daném procesu. Je také nutné uchovávat některé parametry podstatné pro průběh simulace, konkrétně *seznam cyklů* (smyčky uvnitř grafu) a *čítače iterací* a *kroků*. Uchovávání seznamu cyklů vyžaduje vytvoření několika nových tříd, které umístíme do ***w2e.model.cycle***:

- `CycleList` – seznam cyklů (`Cycle`), obsahuje operace nad celým seznamem
- `Cycle` – třída pro jednotlivé cykly, kromě seznamu objektů, které cyklus tvoří (`CycleItem`), musí obsahovat i odkaz na *počáteční blok cyklu*
- `CycleItem` – dvojice blok a jeho výstupní proud

Vzájemné vztahy tříd modelu lze nejlépe znázornit diagramem tříd (obr. 4.5). Dosud neposané třídy `AbstractBlock` a `AbstractFlow` jsou implementace bloků a proudů, blíže se jim budu věnovat v následujících dvou částech této kapitoly.



Obrázek 4.5: Diagram tříd, detail pro třídy modelu

Modelu se týkají ještě dvě speciální třídy, které nejsou v diagramu tříd znázorněny, neboť nejsou s třídou `Model` přímo svázány. První souvisí se spuštěním a během simulace, nazvěme ji proto `ModelExecution`. Tato třída musí být implementována jako samostatné vlákno aplikace, aby mohla simulace běžet nezávisle na rozhraní. Jejím úkolem je kontrola modelu před spuštěním simulace a řízení běhu simulace po jejím spuštění. Druhá třída je rozšířením třídy `Exception`, jedná se o třídu `ModelException`, jejímž prostřednictví se bude provádět oznamování a zachytávání chyb spojených s modelem.

4.3.2 Bloky

Rozhodování o podobě návrhu bloků bylo nejsložitějším v celém procesu návrhu. V tomto bodě totiž dochází ke kolizi několika představ. První variantou byla možnost uživatelské tvorby bloků přímo z aplikace. Po konzultacích s odborníky z FSI (zmíněných v poděkování na začátku práce) jsme raději zvolili variantu programového rozšiřování, neboť odlišné návrhy bloků by vyžadovaly tvorbu příliš složité návrhové komponenty uživatelského rozhraní. Základní představa rozšiřitelné architektury je tato: Návrh bude obsahovat pouze abstraktní definici bloku, jež půjde snadno rozšířit pomocí konkrétních modelů bloků. Modely bloků pak budou závislé na aplikační doméně nástroje.

Základní třídu bloku pojmenujme `AbstractBlock` (jelikož jde o abstraktní třídu, tj. pouze základní šablonu společnou pro všechny odvozené třídy) a věnujme se v této chvíli vlastnostem, které by měly být všem blokům společné. Základními identifikačními rysy bloku jsou jeho *název* a označení *typu*. Dále musí blok udržovat informaci o všech svých *vstupních* a *výstupních proudech*. Nedílnou součástí každého bloku také musí být *seznam atributů* (parametrů), které umožňují jeho regulaci. Posledním parametrem, který by měl blok uchovávat, je *informace, zda je daný blok součástí nějakého cyklu*, tato informace je totiž velmi důležitá z hlediska simulačních výpočtů. Tím je struktura bloku úplná, některé parametry však nejsou primitivních typů, a proto vyžadují návrh vlastních tříd:

- `BlockEnum` – enumerační typ obsahující definice všech typů bloků implementovaných v simulátoru
- `BlockAttribute` – třída reprezentující atribut bloku, obsahuje tyto parametry: *jméno*, *typ atributu* (`BlockAttrEnum`), *jednotku*, *aktuální hodnotu*, *minulou hodnotu* a *parametr used*, který určuje, zda se má tento atribut používat ve výpočtech bloku
- `BlockAttrEnum` – enumerační typ obsahující definice všech typů atributů bloku implementovaných v simulátoru

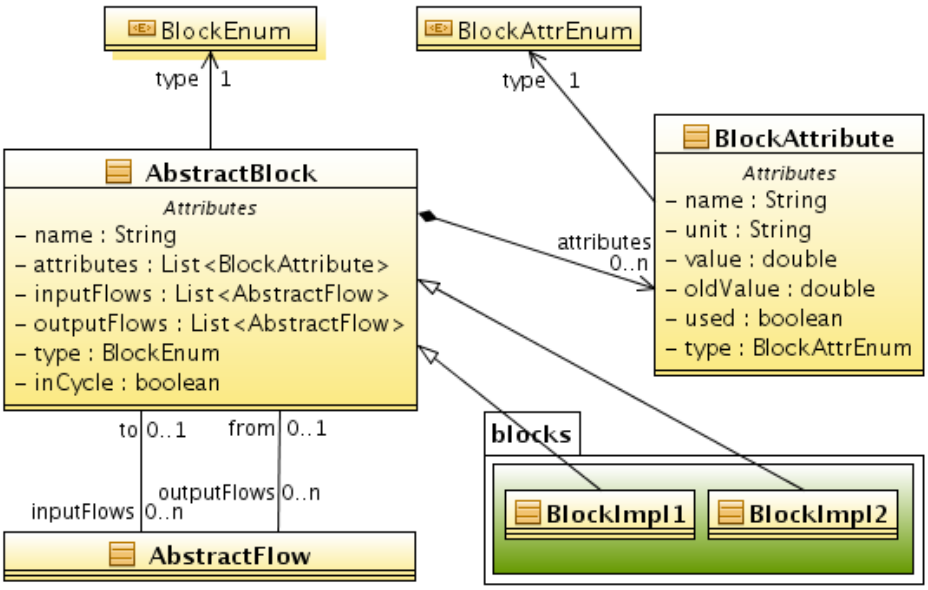
Vzájemné vztahy tříd bloku jsou opět dobře patrné v diagramu tříd (viz obr. 4.6). Význam a způsob implementace různých enumeračních tříd používaných v návrhu (např. `BlockEnum` a `BlockAttrEnum`) nalezne čtenář v kap. 5.1.

4.3.3 Proudy

Návrh proudu je částečně příbuzný s návrhem bloku. Hlavní podobnost spočívá ve faktu, že jednotlivé typy proudů se mohou velmi výrazně lišit, a proto není zcela výhodné vytvářet jednu generickou třídu pro všechny typy proudů, mnohem efektivnější je zopakovat postup aplikovaný u bloků: vytvořit abstraktní třídu a odložit problém návrhu detailů jednotlivých typů proudů do implementační fáze.

Základní třídu proudu nazveme `AbstractFlow` a definujeme parametry společné pro všechny typy proudů. Patří mezi ně samozřejmě *název* a *typ* proudu. Proud také potřebuje znát bloky, které spojuje, tedy *zdrojový* a *cílový blok*. U proudu je o něco složitější definice jeho vlastností, kromě základních *parametrů* (fyzikálních vlastností jako teplota, tlak, průtok) daného proudu je potřeba také uchovávat informaci o *složkách* a *přísadách* obsažených v tomto proudu. Opět je nutné definovat třídy pro některé netriviální parametry:

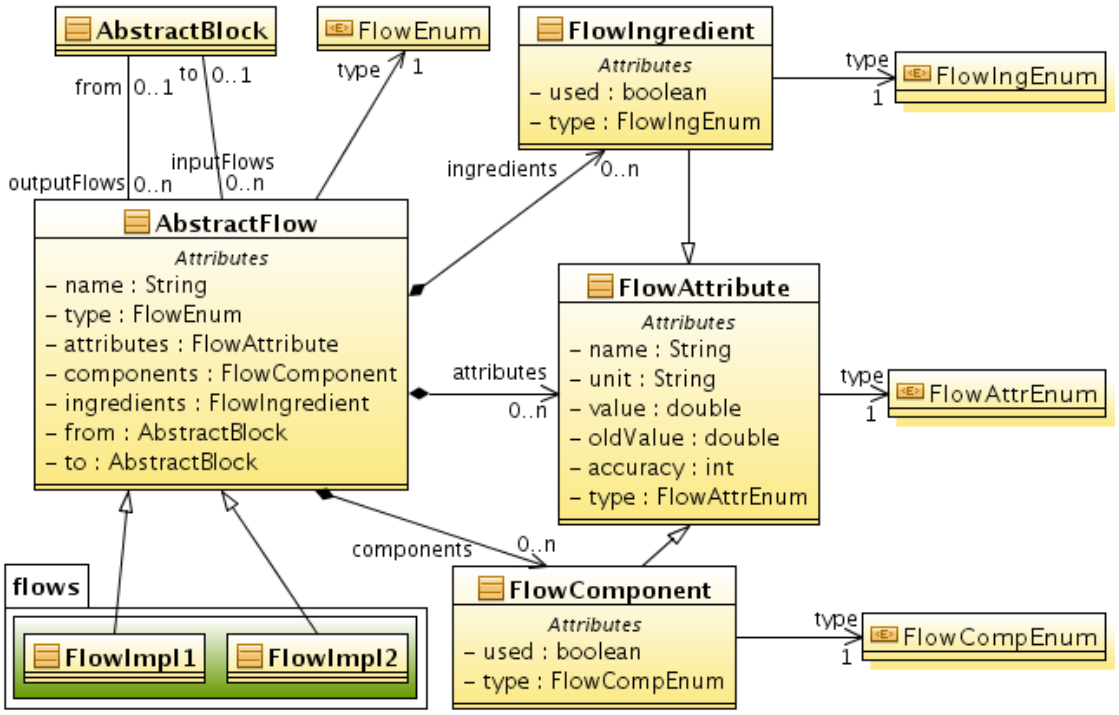
- `FlowEnum` – enumerační třída obsahující definice všech typů proudů implementovaných v simulátoru



Obrázek 4.6: Diagram tříd pro model bloku

- `FlowAttribute` – třída reprezentující parametr proudu, obsahuje tyto parametry: *jméno*, *typ atributu* (`FlowAttributeEnum`), *jednotku*, *aktuální hodnotu*, *minulou hodnotu* a *přesnost*, tj. maximální počet desetinných míst
- `FlowAttributeEnum` – enumerační třída obsahující definice všech typů parametrů proudu implementovaných v simulátoru
- `FlowComponent` – třída reprezentující složku proudu, je definována jako potomek třídy `FlowAttribute`, definuje si však vlastní parametr *typ složky* (`FlowComponentEnum`) a parametr *used*, který určuje, zda se má tento atribut používat ve výpočtech
- `FlowComponentEnum` – enumerační třída obsahující definice všech typů složek proudu implementovaných v simulátoru
- `FlowIngredient` – třída reprezentující přísadu proudu, je definována jako potomek třídy `FlowAttribute`, definuje si však vlastní parametr *typ přísady* (`FlowIngredientEnum`) a parametr *used*, který určuje, zda se má tento atribut používat ve výpočtech
- `FlowIngredientEnum` - enumerační třída obsahující definice všech typů přísad proudu implementovaných v simulátoru

Pokud se podíváme na diagram tříd (obr. 4.7), můžeme vyčíst některé zajímavé detaily. Složky i přísady proudu jsou definovány jako specializované třídy odvozené od parametru proudu, liší se pouze v definici parametru *typu* a možnosti volby, zda se má daná proměnná používat v simulačních výpočtech. Můžeme si také všimnou nápadné podobnosti tohoto diagramu s diagramem tříd pro bloky, to je dáno do značné míry určitou podobností modelovaných skutečností, navíc tato jednotnost usnadňuje pochopení a práci s modelovaným systémem.



Obrázek 4.7: Diagram tříd pro model proudu

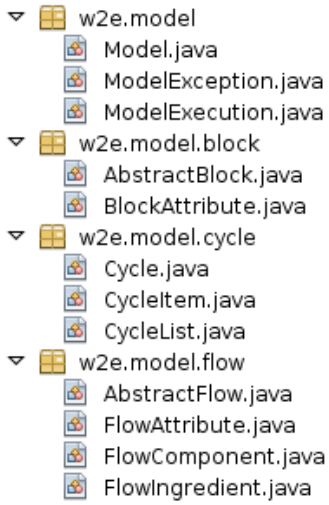
Jelikož je v tomto bodě návrh modelu dokončen, je vhodné promítnout jej ještě do diagramu balíčků (obr. 4.8).

4.4 Architektura rozhraní

Úkolem uživatelského rozhraní je zajistit interakci uživatele s programem. Grafické uživatelské rozhraní by mělo být z větší části intuitivní, tedy uživatel by měl pochopit funkci většiny prvků bez zdlouhavého studia uživatelského manuálu. Toho se dosahuje jednak unifikovaným návrhem rozhraní, v druhé řadě pak pomocí různých grafických prvků jako jsou ikony a rozbalovací komentáře. Speciálně pro vývoj rozhraní je v Javě určen toolkit s název Swing.

Vstupním bodem do grafické aplikace v Javě (ve Swingu) je zpravidla třída odvozená od `JFrame`, jež implementuje hlavní okno aplikace. Hlavní okno by mělo obsahovat tyto prvky:

- **Lišta s menu:** pomocí komponenty `JMenuBar`, skrze vysouvací menu je zpřístupněna většina funkčnosti aplikace.
- **Lišta s nástroji:** pomocí komponenty `JToolBar`, obsahuje přepínače režimů (selekce / vkládání) a vybrané položky z menu (rychlá dostupnost často používaných funkcí).
- **Plátno:** hlavní prvek rozhraní, umožňuje vytvářet a měnit model procesu, představuje přehledný způsob zobrazení modelu.
- **Postranní panel:** zobrazuje se jen pokud je to potřeba, poskytuje nástroje k úpravě



Obrázek 4.8: Úplný diagram balíčků pro model

plátna v závislosti na zvolené akci či stavu plátna (výběr typu vkládaného objektu, nastavení vybraného objektu).

- **Stavový řádek:** pomocí komponenty `JTextField`, slouží jako forma nápovědy, poskytuje textový popis vztahující se k aktuální pozici kurzoru myši.

Všechny zmíněné komponenty s výjimkou plátna a postranního panelu jsou primitivní, a proto je možné provést jejich implementaci přímo ve třídě `MainFrame`. Implementace postranního panelu může být provedena v několika různých třídách v závislosti na typu obsahu tohoto panelu, jediným omezením je, že postranní panel musí být vždy implementován na komponentě vycházející ze třídy `JPanel`. Nejsložitější komponentou hlavního okna je plátno, proto bude implementováno v samostatné třídě `Canvas`.

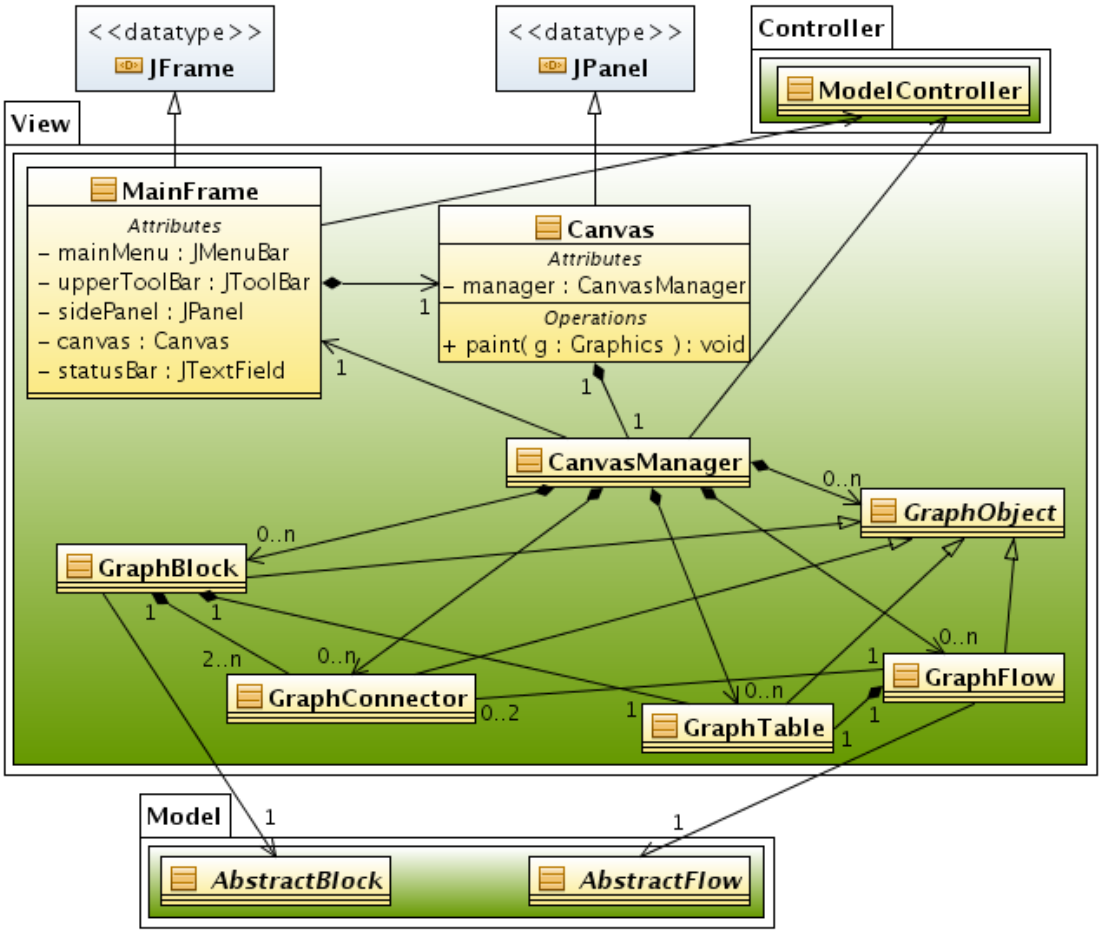
U třídy `Canvas` se na chvíli pozastavíme. Jelikož jde z praktického hlediska pouze o plochu s určitým grafickým obsahem, založíme její implementaci na třídě `JPanel`. Nejdůležitější funkcí plátna je vykreslování grafické reprezentace aktuálního modelu procesu. Problémem je, že obsah plátna nezávisí pouze na modelu procesu, je nutné brát v úvahu také kontext aktuální uživatelské činnosti (tedy určitou stavovou informaci). Pokud bychom ponechali veškerou implementaci plátna ve třídě `Canvas`, docházelo by ke zbytečným komplikacím a konfliktům s metodami třídy `JPanel`, proto raději vytvoříme speciální třídu `CanvasManager`, která bude uchovávavat stav i obsah plátna a na požádání vygeneruje veškerý grafický obsah. Třída `Canvas` tak bude obsahovat jen základní metody pro komunikaci plátna s hlavní aplikací, veškeré detaily související s vykreslováním obsahu a interakcí s uživatelem zůstanou zapouzdřeny ve třídě `CanvasManager`, případně v dalších pomocných třídách.

Aby mohla třída `CanvasManager` spravovat plátno, musí si uchovávat vlastní představu o jeho obsahu. Pro objekty plátna definujeme abstraktní třídu `GraphObject`. Tato třída musí obsahovat základní metody pro práci s grafickými objekty (poloha, přesun, získání rozměrů, vykreslení), detaily pro různé typy grafických objektů pak definujeme ve čtyřech specifických třídách odvozených od `GraphObject`:

- `GraphBlock` – grafická reprezentace bloku, zahrnuje obrázek bloku, funkce pro grafickou editaci obrázku a rozmístění grafických konektorů pro připojení proudů; musí

mít přístup k modelu bloku, odkud získává veškeré parametry, jež pak mohou být zobrazeny buď přímo blokem nebo pomocnými tabulkami (případně změněny prostřednictvím předvoleb rozhraní)

- `GraphFlow` – grafická reprezentace proudu, zahrnuje model křivky, pomocí níž je možné propojit dva grafické konektory; analogicky jako v případě grafického bloku musí mít přístup k modelu proudu
- `GraphConnector` – grafická reprezentace vstupů a výstupů bloku, případně vstupů a výstupů okolí procesu, jinak také grafické konektory
- `GraphTable` – tabulka pro výstup aktuálních hodnot v různých místech simulace; pouze pro zobrazení údajů, čerpá hodnoty z příslušného grafického bloku nebo proudu (ty je pro změnu získávají přímo z modelu, jak bylo řečeno výše)



Obrázek 4.9: Diagram tříd z pohledu rozhraní

Tímto je výčet hlavních komponent rozhraní úplný, v diagramu tříd (obr. 4.9) jsme se tentokrát zaměřili nejen na interakci objektů v rámci rozhraní, ale i na interakce vedoucí k ostatním komponentám architektury. Z diagramu je mimo jiné patrné, že třídy `MainFrame` a `CanvasManager` mohou ovlivňovat model prostřednictvím třídy `ModelController`.

Nyní můžeme tuto kapitolu uzavřít, ta další se bude zabývat konkrétní implementací nástroje pro modelování a simulaci procesů souvisejících se zpracováním biomasy.

Kapitola 5

Implementace modelu procesů pro zpracování biomasy

Způsob implementace modelu je rozhodujícím faktorem pro chování simulačního systému, jeho použitelnost a rozšiřitelnost. Tato kapitola se zaobírá způsobem implementace jádra simulátoru nezávisle na rozhraní. Čtenář zde nalezne ukázky implementace jednotlivých bloků a technologických proudů, způsob provádění simulačních výpočtů a řízení simulace. Vzorce používané pro výpočty v jednotlivých blocích zde uvedeny nejsou (alespoň ne ve své původní podobě), všechny lze však nalézt v [8].

Na vyvíjený systém byly ze strany FSI kladeny tyto nároky (některé z nich jsou uvedeny v [7], další vyplynuly z konzultací s odborníky z FSI):

- Schopnost výpočtu hmotnostních a energetických bilancí pro vybrané modely bloků.
- Schopnost řešit modely procesů obsahující recykly (recyklové proudy).
- Možnost přidávat nové výpočetní bloky a nové typy proudů bez potřeby nějak zásadně zasahovat do programu (= snadná rozšiřitelnost).
- Možnost editace parametrů a vlastností bloků a proudů bez potřeby hlubšího zásahu do programu (= snadná úprava parametrů).

První dva body souvisejí s výpočetními schopnostmi nástroje, kterými se budeme zabývat v druhé polovině této kapitoly, nyní se budeme věnovat řešení druhé dvojice požadavků. Vznikly z jednoho prostého důvodu: simulační nástroj je vyvíjen speciálně pro potřeby pracovníků FSI, případně jiných odborníků na technologické procesy, u kterých se nepředpokládá zevrubná znalost programovacích technik, přesto je však žádoucí umožnit těmto pracovníkům experimentování s existujícími prvky simulátoru a snadné přidávání nových prvků (čímž mohou upravovat a rozšiřovat možnosti simulátoru). K dosažení tohoto cíle je v aplikaci bohatě využíváno výčtových tříd Javy.

5.1 Výčtové třídy pro definici vlastností modelů používaných v simulátoru

Výčtový typ v jazyce Java se velmi významně liší od svého pojetí v jiných programovacích jazycích (jako např. C, C++, C#, Visual Basic, viz [3]). V Javě vystupuje výčtový typ jako speciální typ třídy, nejde tedy jen o množinu konstant, ale každý prvek výčtového typu je

samostatným objektem a jako takový může být nositelem neomezeného počtu parametrů. Pro každou skupinu výčtů vytvoříme samostatnou výčtovou třídu, v této třídě pak můžeme definovat specifický konstruktor, čímž umožníme provedení podrobné definice vlastností u každého prvku výčtu (viz. [12]).

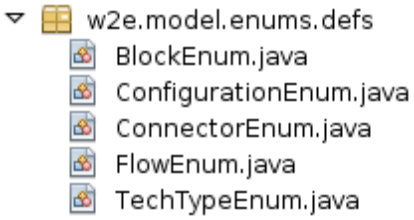
Této vlastnosti využíváme v implementaci tak, abychom „neprogramátorům“ co nevíce usnadnili správu modelů v systému. Veškeré definice týkající se modelovaných skutečností jsou deklarované ve výčtových třídách, které mají přesně danou strukturu a stačí pouze upravit definice položek na začátku těchto souborů (je samozřejmě potřeba v tomto případě překompilovat aplikaci), aby se změnily určité parametry simulátoru. Veškeré výčtové třídy modelu jsou uloženy v balíčku ***w2e.model.enums*** a můžeme je rozdělit na dvě skupiny:

1. výčtové třídy sloužící k výčtu všech modelů bloků a proudů v systému a k popisu jejich vzájemných vztahů ⇒ **Výčtové třídy pro deklaraci modelů a jejich vztahů**,
2. výčtové třídy sloužící k definici a popisu jednotlivých parametrů bloků a parametrů, složek a přísad proudů ⇒ **Výčtové třídy pro definici parametrů modelů**

Podívejme se blíže na implementační detaily těchto tříd.

5.1.1 Výčtové třídy pro deklaraci modelů a jejich vztahů

Tyto třídy jsou definovány v balíku ***w2e.model.enums.defs***, jeho diagram je znázorněn na obr. 5.1.



Obrázek 5.1: Diagram balíčku ***w2e.model.enums.defs***

Třídy `BlockEnum`, resp. `FlowEnum` slouží k deklaraci všech typů bloků resp. proudů, které mají být v simulátoru k dispozici. U každého deklarovaného typu bloku i proudu se očekává jeho dodatečná definice v patřičném balíčku modelu (viz kap. 5.2). Jejich deklarace však nestačí, je potřeba také zadat vzájemné vztahy. Každý blok musí mít nejprve definovány technologické typy všech svých vstupních a výstupních konektorů. Každému vstupnímu konektoru se přiřadí množina povolených typů proudů. V závislosti na tom je pak potřeba pro každou povolenou kombinaci vstupů každého bloku definovat jednoznačnou matici výstupních proudů. Popsaná funkčnost je implementována ve zbylých třech výčtových třídách balíčku – podívejme se nyní, jakým způsobem.

Třída `TechTypeEnum` definuje jeden technologický typ pro každý konektor každého modelu bloku. Pro snadnou orientaci je dodržována jednotná konvence při pojmenovávání technologických typů, každý název se skládá ze tří částí oddělených podtržítkem: první část je zkrácená verze názvu bloku, druhá je řetězec „IN“ v případě vstupního a „OUT“ v případě výstupního konektoru, třetí je pořadové číslo, které rozlišuje konektory se shodnou první i druhou částí. Konstruktor objektu obsahuje jeden parametr booleovského typu, který označuje, zda jde o vstupní konektor. Takto by vypadala třída `TechTypeEnum`, jež by definovala jeden vstupní a dva výstupní konektory pro blok *Mixer (směšovač)*:

```
public enum TechTypeEnum {
    // zde se vkládají instance (= položky) enumerační třídy
    // každá položka odpovídá jedné modelované skutečnosti
    // část část může měnit kdokoliv
    MIXING_IN_1(true),
    MIXING_IN_2(true),
    MIXING_OUT_1(false);

    // zde začíná definice struktury enumerační třídy
    // tato část se běžně nemění, zasahuje do ní jen programátor
    private boolean input;

    // konstruktor - definuje jeden parametr
    TechTypeEnum(boolean input) {
        this.input = input;
    }

    // zjištění typu konektoru
    public boolean isInput() {
        return this.input;
    }
}
```

Třída `ConnectorEnum` definuje jednu podobu konektoru pro každou přípustnou kombinaci technologického typu konektoru a typu proudu. Konstruktor nese tedy dva parametry, jedním je typ proudu, druhým technologický typ. Konvence pojmenovávání podob konektorů je podobná jako u třídy `TechTypeEnum`, pořadové číslo je tentokrát však až na čtvrtém místě, na třetím je číslo odpovídajícího technologickému typu. Takto vypadá třída `ConnectorEnum` pro směšovač:

```
public enum ConnectorEnum {
    MIXING_IN_1_1 (FlowEnum.GAS_MIX, TechTypeEnum.MIXING_IN_1),
    MIXING_IN_1_2 (FlowEnum.AIR, TechTypeEnum.MIXING_IN_1),
    MIXING_IN_1_3 (FlowEnum.WATER_STEAM, TechTypeEnum.MIXING_IN_1),
    MIXING_IN_2_1 (FlowEnum.GAS_MIX, TechTypeEnum.MIXING_IN_2),
    MIXING_IN_2_2 (FlowEnum.AIR, TechTypeEnum.MIXING_IN_2),
    MIXING_IN_2_3 (FlowEnum.WATER_STEAM, TechTypeEnum.MIXING_IN_2),
    MIXING_OUT_1_1 (FlowEnum.GAS_MIX, TechTypeEnum.MIXING_OUT_1),
    MIXING_OUT_1_2 (FlowEnum.AIR, TechTypeEnum.MIXING_OUT_1),
    MIXING_OUT_1_3 (FlowEnum.WATER_STEAM, TechTypeEnum.MIXING_OUT_1);

    private FlowEnum flowType;
    private TechTypeEnum techType;

    // konstruktor
    ConnectorEnum(FlowEnum flowType, TechTypeEnum techType) {
        this.techType = techType;
        this.flowType = flowType;
    }
```

```
    // zjištění typu proudu
    public FlowEnum getFlowType() {
        return flowType;
    }

    // zjištění technologického typu konektoru
    public TechTypeEnum getTechType() {
        return techType;
    }
}
```

Poslední z trojice tříd, `ConfigurationEnum`, jednoduše udává všechny možné kombinace vstupních a výstupních podob konektorů (= konfigurace). Konstruktor nese dva parametry, prvním je pole vstupních podob konektorů, duhým pole odpovídajících výstupních podob konektorů. Název konfigurace má pouze dvě části oddělené podtržítkem, první je zkrácenou verzí názvu bloku, ke kterému konfigurace patří, druhá pořadové číslo konfigurace pro daný blok. Třída `ConfigurationEnum` pro směšovač by vypadala takto (z důvodu úspory místa uvádím pouze dvě ukázkové konfigurace z celkových devíti):

```
public enum ConfigurationEnum {
    MIXING_1(new ConnectorEnum[]
        {ConnectorEnum.MIXING_IN_1_1, ConnectorEnum.MIXING_IN_2_1},
        new ConnectorEnum[] {ConnectorEnum.MIXING_OUT_1_1}),
    MIXING_2(new ConnectorEnum[]
        {ConnectorEnum.MIXING_IN_1_1, ConnectorEnum.MIXING_IN_2_2},
        new ConnectorEnum[] {ConnectorEnum.MIXING_OUT_1_1});

    private final LinkedList<ConnectorEnum> inputsRequired;
    private final LinkedList<ConnectorEnum> outputsRequired;

    // konstruktor
    ConfigurationEnum(ConnectorEnum[] inputs, ConnectorEnum[] outputs) {
        this.inputsRequired = new LinkedList<ConnectorEnum>();
        for (int i = 0; i < inputs.length; i++) {
            this.inputsRequired.add(inputs[i]);
        }
        this.outputsRequired = new LinkedList<ConnectorEnum>();
        for (int i = 0; i < outputs.length; i++) {
            this.outputsRequired.add(outputs[i]);
        }
    }

    public LinkedList<ConnectorEnum> getOutputsRequired() {
        return outputsRequired;
    }

    public LinkedList<ConnectorEnum> getInputsRequired() {
        return inputsRequired;
```

```
    }
}
```

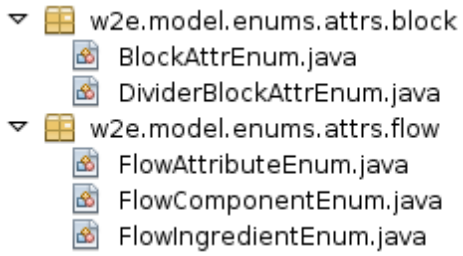
V žádné z těchto tříd není uveden do přímé souvislosti typ bloku s vytvořenými konfiguracemi. To se ve skutečnosti děje přímo ve třídě `BlockEnum`, která kromě deklarace všech typů bloků přiřazuje každému bloku odpovídající seznam konfigurací. Pro úplnost uvedu ukázku položek třídy `BlockEnum`:

```
public enum BlockEnum {
    // položky nesou dva parametry:
    // 1. klíč do tabulky řetězců (kvůli lokalizaci)
    // 2. pole dovolených konfigurací

    // spalování plynných látek
    GAS_COMBUSTION("BlockEnum_Gas_Combustion",
            new ConfigurationEnum[]
            {ConfigurationEnum.GAS_1, ConfigurationEnum.GAS_2}),
    // směšovač
    MIXING("BlockEnum_Mixing",
            new ConfigurationEnum[] {ConfigurationEnum.MIXING_1,
            ConfigurationEnum.MIXING_2, ConfigurationEnum.MIXING_3,
            ConfigurationEnum.MIXING_4, ConfigurationEnum.MIXING_5,
            ConfigurationEnum.MIXING_6, ConfigurationEnum.MIXING_7,
            ConfigurationEnum.MIXING_8, ConfigurationEnum.MIXING_9}),
```

5.1.2 Výčtové třídy pro definici parametrů modelů

V předchozí části byly představeny výčtové třídy, které umožňují provádění rychlých změn v deklaracích modelů a jejich vzájemných vztazích, nyní se pokusím nastínit, jak je vyřešena definice parametrů bloků a parametrů, složek a přísad proudů. Třídy pro tuto funkčnost jsou definovány v balících ***w2e.model.enums.attrs.block*** a ***w2e.model.attrs.flow***, jejich diagram je na obr. 5.2.



Obrázek 5.2: Diagram balíčku ***w2e.model.enums.attrs***

Řešení použité pro bloky a proudy se vzájemně liší, proto si probereme každé zvlášť. Bloky mají definováno jednotné rozhraní pro parametry ve třídě `BlockAttrEnum`. Toto rozhraní předepisuje metody, které musí mít libovolná třída pro parametry bloků implementovány, nijak však neřeší způsob jejich implementace:

```
public interface BlockAttrEnum {
```

```
    // vrací lokalizovaný název parametru
    public String title();

    // vrací unikátní klíč pro vyhledání lokalizovaného názvu
    public String getTitleKey();

    // vrací jednotku parametru
    public String getUnit();

    // vrací přesnost parametru (počet desetin. míst)
    public int getAcc();

    // má se param. automaticky zobrazovat v tabulkách
    public boolean isWatchTable();

    // je možné parametr měnit
    public boolean isEditable();

    // je možné volit použití parametru ve výpočtech
    public boolean isUsable();

    // je možné sledovat parametr
    public boolean isLoggable();
}
```

Balíček ***w2e.model.enums.attrs.block*** pak musí obsahovat jednu výčtovou třídu pro každý typ bloku implementovaný v simulátoru (na obr. 5.2 je v diagramu zapsána pouze třída `DividerBlockAttrEnum`, ve skutečnosti je v aplikaci podobných tříd samozřejmě více). Každá z těchto tříd pak obsahuje definice parametrů charekteristických právě pro jeden typ bloku a každá implementuje rozhraní `BlockAttrEnum`, tudíž lze se všemi zacházet jednotným způsobem. Následuje ukázka implementace výčtové třídy pro parametry bloku *Divider* (implementace metod byly vynechány):

```
public enum DividerBlockAttrEnum implements BlockAttrEnum {
    /** poměr výstupních proudů */
    OUT_FLOWS_RATIO("BlockAttrEnum_Divider_Flows_Ratio", "[-]", 2,
            true, true, true, true),

    /** průtok prvního výstupního proudu */
    OUT_FIRST_FLOW_RATE("BlockAttrEnum_Divider_First_Flow_Rate",
            "<html>kg/h|m<sub>N</sub><sup>3</sup>/h</html>", 2,
            true, true, true, true),

    /** tepelná ztráta */
    HEAT_LOSS("BlockAttrEnum_Global_Heat_Loss", "kJ/h", 2,
            true, true, true, true);

    /** lokalizovaný název parametru */
    private final String titleKey;
```

```
    /** jednotka */
    private final String unit;
    /** přesnost */
    private final int acc;
    /** automaticky zobrazovat v tabulkách */
    private boolean watchTable;
    /** možnost měnit parametr */
    private boolean editable;
    /** možnost zvolit použití parametru ve výpočtech */
    private boolean usable;
    /** možnost sledovat parametr */
    private boolean loggable;

    // konstruktor
    DividerBlockAttrEnum(String titleKey, String unit, int acc,
            boolean watchTable, boolean editable, boolean usable,
            boolean loggable) {
        this.titleKey = titleKey;
        this.unit = unit;
        this.acc = acc;
        this.watchTable = watchTable;
        this.editable = editable;
        this.loggable = loggable;
        this.usable = usable;
    }
}
```

Proudy nepotřebují oddělené definice parametrů pro různé typy, tudíž jsou všechny parametry definovány v jediné enumerační třídě `FlowAttributeEnum`, narozdíl od bloků však musejí nést informaci příslušnosti k jednomu či více proudům. Parametry proudu také postrádají pole *usable*, jelikož by u proudů nemělo smysl[1]. Jinak je implementace parametrů proudů naprosto shodná s parametry bloků.

Narozdíl od bloků nesou proudy kromě svých parametrů ještě další dvě skupiny informací: složky a přísady. Složky jsou definovány ve výčtové třídě `FlowComponentEnum`. Jejich úkolem je nést informaci o chemickém složení daného proudu. Implementačně se složky liší od parametrů hlavně tím, že si uchovávají pole s různými konstantami, které jsou charakteristické pro danou sloučeninu a využívají se při bilančních výpočtech.

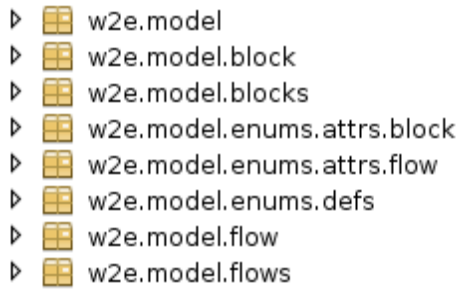
Přísady mají vlastní třídu `FlowIngredientEnum`. Během implementace se ukázalo, že přísady nejsou potřebné, proto se v současné implementaci ve výpočtech vůbec nevyužívají. V budoucím vývoji aplikace se však počítá s rozšířením návrhu, kde budou přísady využity k definici emisí.

5.2 Implementace modelů bloků a proudů

První část kapitoly se zabývala implementací „servisních“ tříd, díky kterým je možné snadno definovat a měnit definice různých detailů modelovaných objektů systému. Teprve nyní se

[1] Parametr *usable* dává uživateli možnost zvolit si u každého parametru bloku, zda se má používat ve výpočtech. U proudu by nebylo smysluplné dovolit uživateli ignorovat určitý parametr (např. teplotu proudu).

však dostáváme k implementaci modelů samotných, tedy ke způsobu, jakým jsou v simulátoru modelovány konkrétní typy bloků a proudů.



Obrázek 5.3: Diagram balíčků důležitých z hlediska bloků a proudů

Na obrázku 5.3 jsou v jednoduchém diagramu shrnuty všechny balíčky, které obsahují třídy implementující funkčnost pro modelování bloků a proudů. Proveďme nyní malou rekapitulaci významu jednotlivých balíčků:

- ***w2e.model.block*** – implementace obecných tříd pro bloky (`AbstractBlock`) a jejich parametry (`BlockAttribute`)
- ***w2e.model.blocks*** – třídy implementující konkrétní modely bloků
- ***w2e.model.enums.attr.block*** – výčtové třídy definující konkrétní statické parametry jednotlivých bloků
- ***w2e.model.enums.attr.flow*** – výčtové třídy definující parametry (`FlowAttributeEnum`), složky (`FlowComponentEnum`) a přísady proudů (`FlowIngredientEnum`)
- ***w2e.model.enums.defs*** – výčtové třídy deklarující existenci jednotlivých typů bloků (`BlockEnum`) a proudů (`FlowEnum`) a jejich vzájemné závislosti (`TechTypeEnum`, `ConnectorEnum`, `ConfigurationEnum`)
- ***w2e.model.flow*** – implementace obecných tříd pro proudy (AbstractFlow), jejich parametry, složky a přísady; také jsou zde třídy, které v závislosti na dříve zmíněných výčtových třídách spravují konfigurace proudů u každého bloku
- ***w2e.model.flows*** – třídy implementující konkrétní modely proudů a také několik statických tříd s pomocnými výpočty pro některé typy proudů

Z výčtu je vidět, že implementace konkrétních modelů jsou uchovávány stranou od obecnějších tříd. Důvodem je snadná orientace, tedy vymezení balíčků, do kterých je nutné zasahovat, pokud chceme do systému přidat nový model bloku nebo technologický proud. Balíčky tříd ***w2e.model.block*** a ***w2e.model.flow*** by měly zůstat podobnými změnami vždy netknuté. Ostatních se přidání nového bloku či proudu dotkne následujícím způsobem:

1. Přidáme deklaraci nového typu bloku, resp. proudu do třídy `BlockEnum`, resp. `FlowEnum`.
2. Vytvoříme provázání mezi novým typem bloku a existujícími proudy, resp. mezi existujícími bloky a novým typem proudu (upravíme třídy `TechTypeEnum`, `ConnectorEnum` a `ConfigurationEnum`).

3. Vytvoříme implementaci nového bloku, resp. proudu a přidáme ji do balíčku ***w2e.model.blocks***, resp. ***w2e.model.flows***.

4. Do metody pro tvorbu nového bloku, resp. proudu (umístěny v třídě `ModelController`, viz kap. 5.3.2) přidáme odkaz pro nový typ bloku, resp. proudu a propojíme tak nově deklarovaný typ s definicí modelu.

Podívejme se nyní na implementační detaily modelovaných objektů.

5.2.1 Třída AbstractBlock

V návrhu architektury modelu bloků (kap. 4.3.2) jsem vyčlenil nejdůležitější pole parametrů, které musí třída `AbstractBlock` (třída nadřazená všem modelům bloků) uchovávat. Během implementace se ukázalo, že je potřebné přidat některé další, konkrétně to jsou tyto:

- `executed : boolean` – booleovská hodnota symbolizující, zda byla již v aktuální iteraci provedena simulace nad daným blokem
- `storeCounter : int` – počítadlo poslední verze bloku, slouží k synchronizaci modelu bloku s rozhraním

Funkčnost objektů je zapouzdřena ve veřejných metodách objektů. Třída `AbstractBlock` obsahuje dva typy těchto metod: abstraktní (tj. pouze deklarace metod, které musí být implementovány v potomcích této třídy) a konkrétní (úplné definice metod). Do abstraktních patří tyto:

- `execute()`: Nejdůležitější metoda kterou musí povinně obsahovat každý blok, je volána za běhu simulace a (jak napovídá její název) jejím úkolem je provedení výpočtů nad daným blokem, tedy získání nových hodnot výstupních proudů daného bloku.
- `fillAttributes()`: Každý blok musí mít speciální metodu pro načtení všech parametrů z výčtových tříd v balíčku ***w2e.model.enum.attrs.block***. Údaje se z výčtových tříd přepíší do objektů třídy `BlockAttribute`, která poskytuje jednotnou strukturu pro parametry statické (získané z výčtových tříd) i dočasné (vytvořené za běhu simulace a nepřístupné z formulářů).

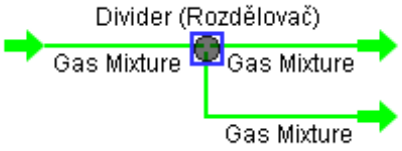
Většina metod je všem blokům společná, a proto jsou implementovány přímo ve třídě `AbstractBlock` (nic samozřejmě nebrání, aby si konkrétní třídy bloků jednotlivé metody předefinovaly dle svých potřeb):

- `isExecutable()`: Zjišťuje, zda je možné spustit výpočet nad daným blokem. Výpočet je možné zpustit v případě, že se tak již nestalo v dané iteraci, a pouze za podmínky, že jsou pro danou iteraci známe hodnoty všech vstupních proudů daného bloku (tj. všechny předchozí bloky jsou již vyčísleny).
- `isInitExecutable(Cycle)`: Tato metoda je velmi podobná předchozí metodě, ale používá se pro bloky, které jsou označeny jako „inicializační“ pro určitý cyklus (recykl) v modelu procesu. Při testech se ignorují nevyčíslené bloky, které jsou součástí zadaného cyklu.
- `grantPort(TechTypeEnum, AbstractFlow, boolean)`: Metoda pro osazení zadaného vstupu / výstupu daného bloku zadaným proudem. Při úspěchu vrátí booleovskou proměnnou „true“, jinak „false“.

- `releasePort(AbstractFlow, boolean isInput)`: Metoda pro odpojení zadaného proudu od daného bloku. Při úspěchu vrátí booleovskou proměnnou „true“, jinak „false“.
- `testPort(TechTypeEnum, AbstractFlow, boolean)`: Podobná metodě `grantPort`, liší se tím, že pouze testuje možnost připojení zadaného proudu k zadanému konektoru bloku.

5.2.2 Třídy implementující konkrétní bloky

Implementace modelů bloků (tedy jednotkových operací pro technologické procesy) je ve své podstatě velmi jednoduchá. Pokud si správně připravíme výčtové třídy podle dříve uvedených informací (kap. 5.1) a máme k dispozici vzorce modelující reakce či pochody probíhající v daném bloku, je tvorba bloku otázkou správného vyplnění šablony.



Obrázek 5.4: Schéma zapojení bloku Divider

Podívejme se například na konstrukci bloku *Divider* (model rozdělovače). Jednoduché schéma zapojení tohoto bloku je na obrázku 5.4, kde můžeme jasně vidět, že tento blok má jeden vstup a dva výstupy. Na vstup můžeme dát libovolný proud, na výstupy se pak vygenerují proudy stejného typu (na uvedeném obrázku jde o proudy typu *směs plynů*). Úkolem bloku je rozdělit původní proud na dva výstupní proudy v závislosti na parametrech nastavených v bloku. U bloku Divider jsou definovány tři parametry:

1. poměr výstupních proudů
2. průtok prvního výstupního proudu (v kg/h nebo m^3_N/h v závislosti na skupenství proudu)
3. tepelná ztráta (v kJ/h)

Za předpokladu, že máme správně definovány všechny potřebné výčtové třídy, je potřeba pouze vytvořit třídu `Divider` a v ní:

- **Vytvořit konstruktor objektu**, který provede následující operace:
  - ▷ nastaví správný název a typ bloku,
  - ▷ inicializuje parametry bloku (metoda **fillAttributes()**),
  - ▷ přidá nový blok do instance **Modelu**.
- **Implementovat metodu fillAttributes()**:
  - ▷ atributy vygenerujeme z výčtové třídy;
  - ▷ ukázka implementace metody pro blok Divider:

```
public void fillAttributes() {
    // na základě odpovídající výčtové třídy
    // přidá parametry typické pouze pro tento typ bloku
    for (DividerBlockAttrEnum ba : DividerBlockAttrEnum.values())
        getAttributes().add(new BlockAttribute(ba));
}
```

- **Implementovat metodu execute()**:
  - ▷ hodnoty výstupních proudů jsou funkcí hodnot vstupních proudů a hodnot parametrů bloku.

Z předchozí ukázky je zřejmé, že modely bloků se hlavně liší v implementaci metody `execute()`. V této metodě jsou prováděny výpočty důsledků fyzikálních procesů a chemických reakcí, jež probíhají v jednotlivých blocích, a na těchto výpočtech je tedy založena simulace v programu W2E. Většina výpočtů používaných v aplikaci je publikována v [8], aplikace je však pod neustálým vývojem a spolupracovníci z FSI výpočty průběžně upravují.

Během první fáze vývoje bylo do aplikace implementováno sedm základních modelů bloků: Divider (rozdělovač), Exchanger (výměník), Gas Combustion (spalování plynů), Solid Combustion (spalování pevných látek), Heating/Cooling (ohřev/chlazení), Mixing (směšovač) a Steam Turbine (parní turbína). Další tři modely byly později úspěšně začleněny do systému pracovníky FSI (konkrétně bloky Dryer, Flash a HRSG), čímž se v praxi ověřila snadná rozšiřitelnost aplikace.

5.2.3 Třída AbstractFlow

Pole parametrů uvedených pro třídu AbstractFlow v předchozí kapitole (kap. 4.3.3) zůstala během implementace bez změny. Co se týče funkčnosti této třídy, můžeme metody rozdělit do několika skupin podle kontextu:

- metody pro inicializaci součástí proudu
  - ▷ `fillAttributes()` – inicializuje parametry proudu na základě výčtové třídy `FlowAttributeEnum`
  - ▷ `fillComponents()` – inicializuje složky proudu na základě výčtové třídy `FlowComponentEnum`
  - ▷ `fillIngredients()` – inicializuje přísady proudu na základě výčtové třídy `FlowIngredientEnum`
- metody pro hromadné nastavení součástí proudu
  - ▷ `setAttrValsByPattern(List<FlowAttribute>)` – nastaví parametry proudu podle uvedeného seznamu parametrů
  - ▷ `setCompByPattern(FlowComponentEnum, double)` – nastaví složku podle uvedeného typu složky a hodnoty
  - ▷ `setCompValsByPattern(List<FlowComponent>)` – nastaví složky proudu dle uvedeného seznamu složek
  - ▷ `setIngredValsByPattern(List<FlowIngredient>)` – nastaví přísady proudu podle uvedeného seznamu přísad

- ▷ `setAllValsByPattern(...)` – nastaví parametry, složky i přísady proudu podle zadaných seznamů
- ▷ `setAllValsByPattern(AbstractFlow)` – nastaví parametry, složky i přísady proudu po vzoru zadaného proudu

- metody pro výpočet různých parametrů proudu na základě jeho vlastností a složení
  - ▷ `solveEntropy()` – vypočte entropii
  - ▷ `solveEnthalpy()` – vypočte entalpii
  - ▷ `solveTemp(Double enthalpy)` – vypočte teplotu na základě zadané entalpie
  - ▷ a několik dalších

- metody pro nastavování, úpravy a získávání informací z proudu
  - ▷ metody začínající na **set-**, **get-**, **add-** a **remove-**

- abstraktní metoda pro přepočet parametrů proudu (např. po provedení uživatelských změn v proudu)
  - ▷ **updateFlowAttributes()**

Z přehledu je patrné, že velká část metod slouží k pohodlné správě vlastností a složení proudu. Velmi specifická je skupina metod pro výpočty parametrů na základě složení aktuálního proudu (metody začínající na solve-). V metodách třídy `AbstractFlow` jsou použity velmi obecné postupy výpočtů, konkrétní modely proudů si mohou dané metody přeimplementovat dle své potřeby (a velmi často takto také činí). Výpočty v těchto metodách jsou založeny na vzorcích z fyzikální chemie, jejich zápis lze nalézt v [8].

5.2.4 Třídy implementující konkrétní proudy

Tvorba modelů proudů je řešena analogicky jako u bloků rozšiřováním třídy `AbstractFlow`. Třída modelující konkrétní proud musí splnit pouze tyto podmínky:

1. v konstruktoru
   - ▷ nastaví správný název proudu a typ proudu,
   - ▷ inicializuje své parametry, složky a ingredience
2. implementuje metodu `updateFlowAttributes()` tak, aby mohla být tato metoda volána vždy po provedení uživatelských změn za účelem přepočtu a úprav parametrů, které byly zneplatněny provedenými změnami

Většina modelů tříd s tímto nevystačí a přeimplementuje si také vybrané metody pro výpočet parametrů proudu. Pro usnadnění výpočtů prováděných v různých blocích navíc modely proudů implementují mnoho vlastních veřejných metod pro výpočty různých parametrů, které lze dopočítat z vlastností a složení daného proudu (tento postup zvyšuje přehlednost programu a výrazně usnadňuje implementaci modelů bloků).

Během první fáze vývoje byly do aplikace implementovány modely čtyř proudů: vzduch (air), směs plynů (gas mixture), tuhé palivo (solid fuel) a voda/pára (water/steam). Později byl do aplikace úspěšně přidán pracovníky FSI proud termální tekutina (termal liquid), tedy

i rozšiřitelnost modelů proudů byla důkladně otestována. Vzorce použité při prvotní implementaci proudů jsou publikovány v [8], modely jsou však během vývoje aplikace průběžně upravovány pracovníky FSI, tudíž některé výpočty mohly doznát změn.

V balíku ***w2e.model.flows*** je mezi modely proudů ještě několik speciálních tříd:

- **SteamTable** – třída implementující parní tabulky
- **XSteam** – třída implementující vlastnosti vody a páry dle IAPWS IF-97 (převedeno z dll knihovny ze stránek www.x-eng.com)
- **SolidWasteMath** – knihovna s výpočty výhřevnosti pro tuhá paliva, využívána pouze v grafickém uživatelském rozhraní (viz kap. 6.5.1)

Zatím jsam v této kapitole prošel dva důležité aspekty simulátoru, konkrétně způsob, jakým je řešeno modelování procesů a technologických proudů, a definice vzájemných vztahů mezi objekty modelu. Chybí nám jednak prvek, který by model nějak spravoval (jelikož třída `Model` je ve skutečnosti pouze zapouzdřením modelu procesu a neobsahuje žádné metody pro vlastní správu modelu) a v druhé řadě prvek pro řízení simulace. Právě třídám implementující tyto činnosti bude věnován zbytek kapitoly.

5.3 Správa modelu a simulace

Odložme problém správy modelu na později a podívejme se nejprve na problematiku řízení simulace. Simulační výpočty jsou již zahrnuty v modelech bloků a proudů, tudíž schází pouze mechanismus, který by postupně spouštěl výpočty nad modely a dohlížel na správné pořadí prováděných operací. Tato funkčnost je implementována v metodě `ModelExecution`.

5.3.1 Třída ModelExecution pro řízení simulace

Třída `ModelExecution` rozšiřuje Javovskou třídu `Thread`, což v praxi znamená, že je implementována jako samostatné vlákno. To je velmi důležité, neboť je žádoucí, aby simulace běžela nezávisle na rozhraní. (V opačném případě by aplikace po spuštění simulace přestala reagovat.) Jádro třídy tvoří tři metody:

- `run()`: Tato metoda je deklarována jako abstraktní v třídě `Thread`, tudíž je povinností každého potomka této třídy, aby tuto metodu implementoval. Třída `Thread` pak dále poskytuje metodu `start()`, která automaticky vytvoří nové vlákno a spustí v něm metodu `run()` [10]. Do metody `run()` proto umístíme spuštění metody `execute()`, jejíž náplní je řízení simulace.
- `execute()`: Je implementována jako potenciálně nekonečná smyčka závislá na booleovské proměnné `stopped`. Po zavolání metody je do pole `stopped` přiřazena hodnota „false“ a spustí se smyčka, jež skončí v případě že `stopped` = „true“. V této smyčce pak probíhá simulace, dokud nedojde k jejímu zastavení nebo k nečekané chybě.
- `stopExecute()`: Zastaví nenásilně simulaci tím, že přířadí poli `stopped` hodnotu „true“. To způsobí doběhnutí metody `execute()` a s ní ukončení činnosti vlákna.

V metodě `execute()` jsou implementovány dva způsoby simulace:

1. *s průběžným testováním* – testy se provádějí vždy před spuštěním každého bloku

2. *s hromadným testováním* – testy se provádějí naráz před provedením výpočtů

Pořadí provádění výpočtů v metodě `execute()` přesně odpovídá teorii uvedené u sekvenčně modulárních metod (kap. 3.3.1). Dokud nejsou zpracovány všechny bloky, metoda prochází jeden po druhém a u každého, který dosud nebyl spuštěn, testuje dostupnost dat jeho vstupních proudů (blok nelze spočítat, pokud nejsou dostupné všechny jeho vstupní proudy). Vstupní proud určitého bloku je dostupný, pokud splňuje jednu z těchto podmínek:

- Jde o nástřikový proud do procesu, tedy proud který vstupuje do procesu z vnějšího prostředí.
- Daný proud je výstupním proudem již spočítaného bloku.
- Jedná se o první iteraci a daný proud je součástí cyklu, ve kterém je daný blok zvolen jako počáteční (inicializační).

Automatické zastavení simulace po dosažení rovnováhy prozatím nebylo implementováno, tudíž simulace probíhá buď neomezeně, nebo dokud není dosaženo předem nastaveného počtu iterací.

Úmyslně jsem se třídě `ModelExecution` věnoval velmi důkladně, jelikož jde o jednu z klíčových tříd celého simulátoru. Další třída je neméně důležitá, jejím úkolem je totiž správa modelu uchovávaného v aplikaci. Přestože je zmíněná třída součástí řadiče a její vývoj probíhal v součinnosti s vývojem rozhraní, uvádím ji zde, neboť implementačne je spojená především s modelem.

5.3.2 Třída ModelController, správce modelu

Rozhraní si v případě potřeby může brát data přímo z objektů modelu, pokud chce však model měnit, mělo by k tomu využívat služeb řadiče. Třída `ModelController` je v pravém slova smyslu správcem modelu. Poskytuje metody pro vytváření a úpravu objektů modelu, ovládání simulace a verifikaci vytvořených schémat.

Opět uvedu výčet nejdůležitějších metod této třídy:

- metody pro operace s modelem jako celkem
  - ▷ `createModel()` – vytvoří novou instanci modelu
  - ▷ `clearModel()` – smaže obsah modelu, maže prvky zároveň i z rozhraní
- metody pro úpravu modelu, zpravidla s vazbou na GUI (tj. metody, které při úspěšném volání automaticky synchronizují GUI)
  - ▷ `createBlock(BlockEnum)` – vytvoří nový blok daného typu v modelu i rozhraní
  - ▷ `destroyBlock(AbstractBlock, boolean destroyGUI)` – smaže zadaný blok z modelu, v závislosti na druhém parametru případně i z GUI
  - ▷ `createIOFlow(FlowEnum, AbstractBlock, TechTypeEnum, boolean)` – vytvoří vstupní nebo výstupní (v závislosti na čtvrtém parametru) proud daného typu u daného bloku
  - ▷ `connectBlocks(AbstractFlow, AbstractBlock, TechTypeEnum)` – propojí daný výstupní proud s daným konektorem daného bloku

- ▷ `connectBlocks(AbstractFlow outputFlow, AbstractFlow inputFlow)` – propojí dané dva bloky
- ▷ `disconnectBlocks(AbstractFlow)` – rozpojí daný proud (tj. smaže daný proud a opraví celou hierarchii schématu)
- ▷ `destroyIFlow(AbstractFlow)` – smaže daný vstupní proud a opraví hierarchii schématu
- ▷ `destroyOFlow(AbstractFlow)` – smaže daný výstupní proud (synchronizace není potřeba)

- metody pro přidávání hotových objektů do modelu (tj. metody používané při načítání schématu); bez vazby na GUI
  - ▷ `addBlock(AbstractBlock)` – pouze přidá daný blok do modelu
  - ▷ `addFlow(AbstractFlow)` – pouze přidá daný proud do modelu
  - ▷ `addInputConnection(AbstractFlow, AbstractBlock, TechTypeEnum)` – přidá daný vstupní proud do daného konektoru daného bloku
  - ▷ `addOutputConnection(AbstractFlow, AbstractBlock, TechTypeEnum)` – přidá daný výstupní proud do daného konektoru daného bloku
  - ▷ `addConnection(AbstractFlow, AbstractBlock from, TechTypeEnum techTypeFrom, AbstractBlock to, TechTypeEnum techTypeTo)` – přidá zadaný proud a propojí jím dané konektory daných bloků

- metody pro ovládání simulace
  - ▷ `executeModel(boolean)` – vytvoří a nastaví novou instanci třídy `ModelExecution` a spustí simulaci (booleovská proměnná určuje způsob provádění testů během simulace, viz kap. 5.3.1)
  - ▷ `stopExecuteModel()` – zastaví simulaci

- pomocné metody
  - ▷ `flowCopy(AbstractFlow from, AbstractFlow to)` – překopíruje data z jednoho proudu do druhého
  - ▷ `isBlockInit(AbstractBlock)` – zjistí, zda je daný blok počátečním blokem nějakého cyklu

U několika metod je v popisu uvedeno, že provádějí opravu hierarchie schématu. Důvodem je fakt, že výstupní proudy bloku mohou být definovány teprve v momentě, kdy známe všechny jeho vstupní proudy. Proto se `ModelController` stará o integritu schématu tak, že ruší výstupní proudy bloků, které nemají kompletní vstupy. Také je nutné automaticky přetypovat výstup bloku při změně typu jeho vstupního proudu, což by teoreticky mohlo odstartovat řetězovou reakci změn po celém schématu; v současnosti se tomuto jevu zamezuje jednoduchým rozpojením problémových proudů (tj. problémové spojení bloků se nahradí samostatným výstupním a vstupním proudem).

Vzhledem k složitosti simulátoru (odpovídající již poměrně dlouhé době jeho vývoje) nemohly být do detailu probrány všechny součásti modelu aplikace, proto jsme se zaměřili hlavně na jeho nejdůležitější rysy. V této kapitole byly popsány všechny třídy důležité pro

modelování technologických procesů, včetně speciálních tříd orientovaných na procesy zpracování biomasy, v závěru kapitoly byl pak detailně popsán způsob řízení simulace a správy modelu z rozhraní. Další kapitola se bude zabývat stavbou grafického uživatelského rozhraní aplikace W2E.

Kapitola 6

Grafické uživatelské rozhraní simulátoru

Grafické uživatelské rozhraní je pro simulační nástroj velmi důležitou součástí. Umožňuje mnohem přehlednější výstup simulačních výsledků, než je tomu v případě konzolové aplikace, ale co je nejdůležitější, může uživatele provádět procesem návrhu modelu a velmi mu tento proces usnadnit. Z tohoto důvodu byly na grafické uživatelské rozhraní nástroje W2E kladeny vysoké nároky a po dokončení základní implementace modelu se drtivá část vývoje soustředila tímto směrem.

Než začneme s popisem jednotlivých součástí rozhraní a jejich implementace, pojďme si nejprve projít základní vlastnosti rozhraní a jeho externí součásti. Celá aplikace je implementována v jazyce Java (verze 1.6.x), tudíž je dokonale přenositelná včetně rozhraní. Vzhled rozhraní se dokáže přizpůsobit aktuální platformě, ale obsahuje i řadu témat, z nichž část je součástí implementace Javy, největší skupinu však tvoří témata knihovny *Substance*, jež je integrována v aplikaci W2E. Rozhraní je multilinguální, teoreticky je možné doplnit libovolný počet jazyků; aktuální implementace podporuje dva (český a anglický). Pro rozšířenou schopnost práce s obrázky je do aplikace zakomponována knihovna JAI (Java Advanced Imaging), jež sice není součástí základní distribuce Javy, je však volně ke stažení na stránkách Sun Developer Network. Pro lepší orientaci a zvýšení intuitivity aplikace byly do rozhraní integrovány ikony z kolekce *Silk*, jež jsou volně ke stažení na www.famfamfam.com/lab/icons/silk/.

Rozhraní aplikace je grafický a ovládací prvek, jež odděluje uživatele od implementace funkčnosti a dovoluje mu zacházet s aplikací co nejpřirozenějším způsobem. Proto je GUI zároveň i vstupním bodem aplikace a řídí hlavní tok dat aplikací. V Java aplikacích často mluvíme o tzv. *hlavní třídě*, což je právě třída představující vstupní bod do aplikace. Touto třídou je ve W2E třída `MainFrame`, která je umístěna přímo v kořenovém balíku rozhraní ***w2e.view***. Povinností hlavní třídy je implementace metody `main(String args[])`, pole argumentů v parametrech metody dává uživateli možnost předat aplikaci vstupní parametry z příkazové řádky. V grafických aplikacích je pak typickým obsahem metody `main()` vytvoření nové instance třídy hlavního okna a spuštění řídícího vlákna této instance. Jelikož je hlavní okno v aplikaci W2E implementováno přímo třídou `MainFrame`, podívejme se nyní na implementaci této třídy.

6.1 Hlavní okno aplikace, třída MainFrame

V návrhu grafického rozhraní (kap. 4.4) byl zmíněn fakt, že třída `MainFrame` rozšiřuje abstraktní třídu Javy `JFrame`, která představuje implementaci samostatného okna aplikace. Při návrhu grafické části okna bylo využíváno možností vývojového prostředí Netbeans 6.5, jež poskytuje nástroje pro grafickou tvorbu rozhraní (kód je automaticky vygenerován). Struktura hlavní okna byla již jednou popsána v návrhu rozhraní, nyní si ji projdeme znovu a zaměříme se na implementační hledisko.

Na obrázku 6.1 je zobrazeno hlavní okno aplikace včetně popisu jednotlivých komponent, kterými jsou:

- **Lišta s menu:** Po grafické stránce pouze komponenta `JMenuBar` s určitou strukturou menu a položek (komponenty `JMenu` a `JMenuItem`). Každá z položek je sledována tzv. posluchačem (listener), konkrétně posluchačem akcí (implementace třídy `ActionListener`) v podobě vnitřní třídy `MainMenuManager`. Tato třída reaguje na výběr libovolné položky z kteréhokoliv menu a podle kontextu volá funkce tříd `Canvas`, `CanvasManager`, `ModelController` a `ViewController`, případně otevře nový formulář či dialog.

- **Lišta s nástroji:** Graficky jde o komponentu `JToolBar`, která nese panel (`JPanel`) s nápisy (`JLabel`) a dvěma typy tlačítek, klasickými a přepínacími (`JButton` a `JToggleButton`). Dvě přepínací tlačítka slouží k výběru módu plátna a jsou spravovány speciálním posluchačem, který na základě jejich přepínání informuje třídu `Canvas`. Zbytek tlačítek je sledováno posluchačem akcí implementovaným vnitřní třídou `ToolBarManager`. Ten jenom volá metody různých tříd po vzoru třídy `MainMenuManager`, jelikož tyto tlačítka pouze zrcadlí některé často používané položky z menu.

- **Plátno:** Plátno ve skutečnosti není vloženo přímo do hlavního okna aplikace, ale je zapouzdřeno v instanci třídy `JScrollPane` (implementace posuvníků) a dále v instanci `JPanel`, jež tvoří základní podkladový panel šedé barvy. Funkčnost plátna je implementována v třídách balíku ***w2e.view.canvas***, více o nich najdete v kap. 6.2.

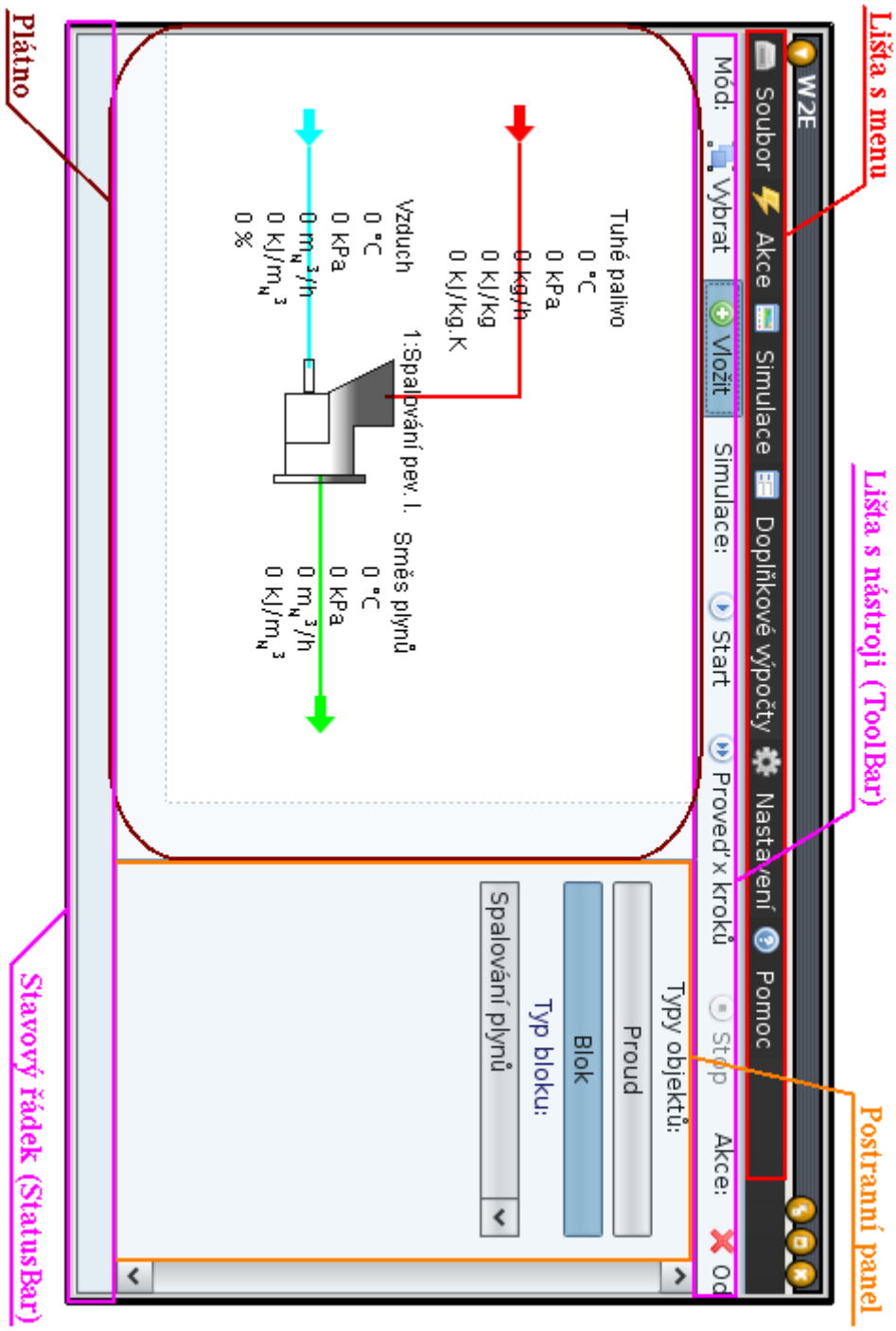
- **Postranní panel:** Musí být vždy implementován potomkem třídy `JPanel`, třída `MainFrame` v závislosti na stavu plátna vytvoří instanci správné třídy a zobrazí ji v pravé části okna. V aktuální implementaci existují tři typy (třídy) postranních panelů:

  - ▷ panel pro vkládání, `InsertPanel` (kap. 6.3.1)
  - ▷ panel pro úpravu proudu, `SimpleFormGenerator` (kap. 6.3.2)
  - ▷ panel pro úpravu bloku, `MainFormGenerator` (kap. 6.3.2)

- **Stavový řádek:** Jde pouze o komponentu třídy `JTextField`, její obsah je generován v třídě `CanvasManager` během detekce stavu plátna (viz kap. 6.2.2).

Z popisu součástí hlavního okna by se mohlo zdát, že třída `MainFrame` neimplementuje prakticky žádnou vlastní funkčnost a pouze zapouzdřuje objekty, které tak činí. Ve skutečnosti má i tato třída několik velmi důležitých funkcí, které dosud nebyly zmíněny:

Nastavení aplikace: Jak bylo již zmíněno, W2E si uchovává konfiguraci některých grafických prvků (viz kap. 6.4). Důsledkem toho je, že první věcí, kterou musí aplikace W2E po spuštění udělat, je načtení nastavení z perzistentního úložiště. Teprve na



Obrázek 6.1: Hlavní okno aplikace W2E

jeho základě je totiž možné nastavit parametry prostředí a zobrazit grafické prvky rozhraní. Metoda `main()` tedy ještě před startem vlákna rozhraní zavolá statickou metodu `MainFrame.loadConfig()` a načte si nastavení, které později předá instanci třídy `MainFrame`.

Lokalizace: Po startu aplikace a také při volbě příslušné položky z menu je nutné nastavit v prostředí Javy správné lokalizační informace. Práce s různými lokalizacemi je v Javě na velmi vysoké úrovni, kromě všech řetězců je možné lokalizovat i formáty čísel, dat a dalších textových polí, čehož se v aplikaci W2E samozřejmě využívá. Nastavení lokalizace při startu je snadné, stačí v metodě `main()` zavolat funkci Javy `Locale.setDefault(Locale)`. Horší je situace za běhu grafického rozhraní, jelikož změna proměnné prostředí neovlivní již zobrazené prvky. Z tohoto důvodu musí každá grafická komponenta implementovat metodu `updateLocale()`, která obnoví všechny aktuálně zobrazené prvky ovlivněné změnou lokalizace. Zadání příkadu ke změně lokalizace za běhu rozhraní tak způsobí, že třída `MainFrame` spustí nejprve svoji metodu `updateLocalize()` a následně metody `updateLocalize()` všech svých grafických potomků – tím se změna šíří postupně celým rozhraním.

Vzhled aplikace: Další činností, jíž je vhodné (i když nikoliv nezbytně nutné) provést ještě před startem GUI, je volba vzhledu aplikace. Tato funkce je implementována ve statické metodě `MainFrame.updateLookAndFeel(String)`.

Řádné ukončení aplikace: Pro řádné ukončení aplikace slouží `exitApplication()`, jež nejprve uloží aktuální konfiguraci metodou `saveConfig(Configuration)`, poté zavře model a ukončí běh aplikace. V budoucnu se počítá s implementací funkcí Undo a Redo pro plátno a s nimi i detekce změn od posledního uložení schématu, tudíž bude možné zavést před ukončením aplikace ještě kontrolu, zda je aktuální schéma uložené, a nabídnout dialog k jeho uložení.

Další důležitou komponentou (ve skutečnosti tou nejdůležitější komponentou) je plátno. Vzhledem ke složitosti rozhraní a množství tříd, které často implementují funkčnost toho, co z vnějšku vypadá jako jediná komponenta, nebude zbytek této kapitoly strukturován po vzoru Javovských balíčků, jak tomu bylo v předchozí kapitole. Namísto toho bude struktura zbytku práce spíše napodobovat stavbu rozhraní, jak ji vnímá uživatel.

6.2 Plátno

Plátno v aplikace W2E je možné vnímat dvěma způsoby: podle prvního je plátno tvořeno instancí třídy `Canvas`. Při bližším studiu implementace je však zřejmé, že `Canvas` je jen začátkem, na plné funkčnosti plátna se podílejí veškeré třídy, jež jsou obsaženy v balíčku ***w2e.view.canvas*** (v současnosti rovných třicet tříd rozdělených do třech balíčků). Než se pustíme do přehledu různých součástí plátna, pokusíme se nejprve vysvětlit, jaké jsou vlastně úlohy plátna v aplikaci W2E.

První a nejdůležitější funcí plátna je zobrazování aktuálního stavu modelu. K tomu aby tak mohlo plátno činit, nestačí jen prostý přístup k modelu a jeho obsahu, jelikož model neobsahuje žádné informace grafického charakteru. Plátno si tedy musí udržovat vlastní reprezentaci všech modelovaných objektů zameřenou na informace vizuálního charakteru. Navíc je žádoucí, aby bylo možné zobrazit nejen reprezentace modelů, ale i jejich fyzikální

a chemické vlastnosti, tudíž plátno musí být schopno udržovat složitější hierarchii objektů, než jaká je uložena v modelu.

Zobrazení je však jen jednodušší polovinou funkčnosti plátna. S modelem totiž musí být možné manipulovat, takže se od plátna očekává schopnost intuitivní interakce. To v praxi znamená, že by mělo být možné jednotlivé objekty přirozeným způsobem označovat, vybírat, upravovat a nastavovat. S narůstající složitostí modelů se navíc objevuje požadavek na možnost ukládání a načítání dat z/do souborů.

Když máme nyní dány požadavky kladené na plátno, přejděme k jeho struktuře a zodpovědnosti jednotlivých součástí:

- **třída Canvas:** Ve skutečnosti pouze panel s bílým pozadím, který udržuje informaci o aktuální velikosti plátna a po jeho okrajích vykresluje čárkované ohraničení. Implementuje tyto důležité metody:
  - ▷ `getCanvasController()` – vrátí speciální rozhraní, které zpřístupňuje malou skupinu metod třídy `CanvasManager` pro správu některých parametrů a funkcí plátna
  - ▷ `showPopup(JPopupMenu, int x, int y)` – zobrazí dané pop-up menu na zadaných souřadnicích plátna
  - ▷ `paint(Graphics)` – překreslí panel, požádá `CanvasManager` o vykreslení obsahu plátna a dokreslí jeho ohraničení
  - ▷ `setActionType(ActionType, ObjectType)` – předá tř. CanvasManager žádost o změnu aktuálně nastaveného módu plátna a objektu
  - ▷ `setBlockType(BlockEnum)` – předá tř. `CanvasManager` žádost o změnu aktuálně nastaveného typu bloku
  - ▷ `setFlowType(FlowEnum)` – předá tř. `CanvasManager` žádost o změnu aktuálně nastaveného typu proudu
  - ▷ `setFlowStyle(FlowStyle)` – předá tř. `CanvasManager` žádost o změnu aktuálně nastaveného stylu proudu
  - ▷ `updateLocale()` – předá tř. `CanvasManager` žádost o aktualizaci dat podléhajících lokalizaci

  Metody začínající na „set“ jsou implementovány kvůli přehlednosti a čistotě aplikace, aby třídy mimo ***w2e.view.canvas*** nepřistupovaly přímo k třídě `CanvasManager`. Alternativní způsob, který dává přístup k části funkčnosti třídy `CanvasManager`, je získání rozhraní `CanvasController`.

- **grafické objekty plátna** Každý modelovaný objekt má svoji grafickou reprezentaci, která se často skládá z více objektů různých tříd. Podrobný popis je v kap. 6.2.1.

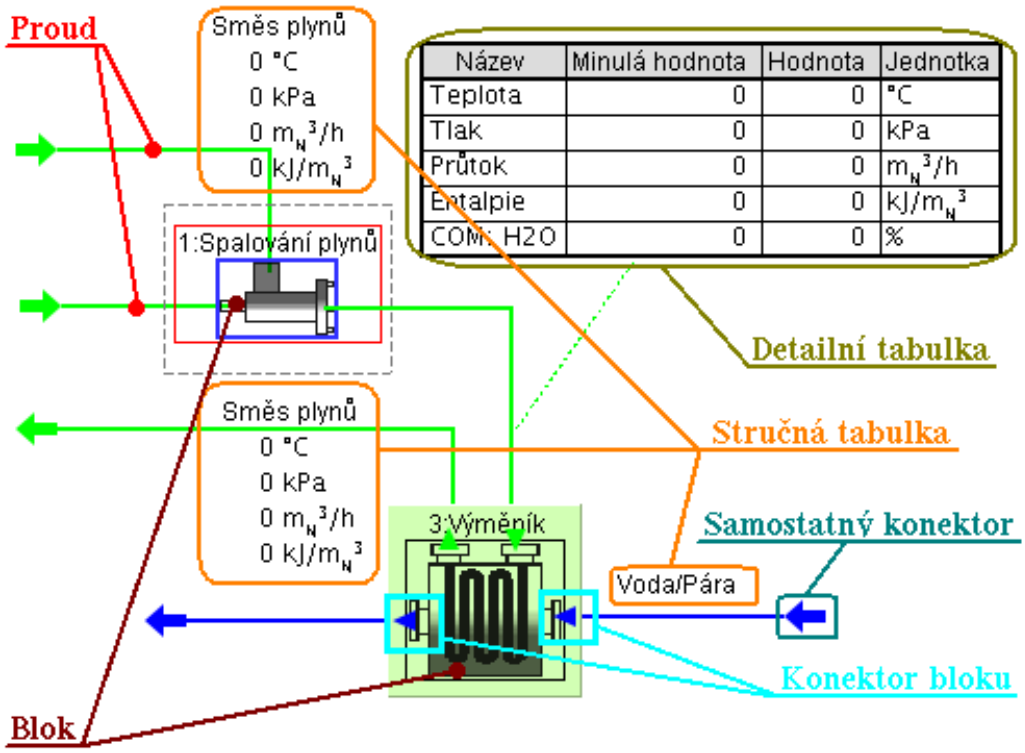
- **správce plátna, třída `CanvasManager`:** Nejsložitější třída rozhraní, udržuje stav plátna a implementuje široký zásobník funkcí pro práci s plátnem. Pro detailní popis viz kap. 6.2.2.

- **správce myši, třída `MouseManager`** Tato třída implementuje veškerou interakci s plátnem pomocí myši. Rozšiřuje třídu *MouseInputAdapter* a je připojená jako posluchač veškerých akcí myši ke komponentě `Canvas`. Na zachycené akce potom reaguje na základě stavové informace získané z třídy `CanvasManager` voláním různých funkcí této třídy.

- **další správci** Objekty nebo skupiny objektů, které zapouzdřují nějakou konkrétní část funkčnosti plátna. Vznikly za účelem snížení objemu (a zvýšení přehlednosti) třídy `CanvasManager`. Podrobnosti o jednotlivých třídách jsou uvedeny v kap. 6.2.2.

6.2.1 Grafické objekty plátna

Grafické objekty plátna jsou vizuálními protějšky modelovaných objektů. Jejich úkolem je modelovat objekty z hlediska schématu zapojení, tudíž nás zajímá v první řadě jejich poloha ve dvourozměrném prostoru, v druhém pak vzájemná poloha a vztahy jednotlivých objektů. Dovolím si nyní projít jednotlivé typy objektů a popsat jejich základní vlastnosti. Na obrázku 6.2 je výřez plátna aplikace W2E s význačenými typy objektů plátna. Veškerá funkčnost spojená s objekty plátna je definována v balíčku ***w2e.view.canvas.object***.



Název	Minulá hodnota	Hodnota	Jednotka
Teplota	0	0	°C
Tlak	0	0	kPa
Průtok	0	0	m$_N^3$/h
Entalpie	0	0	kJ/m$_N^3$
COM: H2O	0	0	%

Obrázek 6.2: Objekty plátna

GraphObject Abstraktní třída pro definici grafických objektů. Každý grafický objekt musí být potomkem této třídy, takže všechny grafické objekty mají určité minimum společných vlastností.

Důležité parametry:

- *unikátní identifikátor* – pro jednoznačné řazení objektů stejného typu, důležité při zjišťování pořadí vykreslování a detekci vrchního objektu
- *hladina* – pro jednoznačné řazení objektů různých typů, důležité při zjišťování pořadí vykreslování a detekci vrchního objektu

- *typ* – typ grafického objektu

Důležité metody:

- `draw(Graphics2D)` – metoda pro vykreslení daného objektu na daný grafický kontext
- `getBounds()` – metoda pro zjištění polohy (v podobě nejmenšího obdélníku obsahujícího daný objekt)
- `updatePosition(Point)` – metoda pro změnu aktuální polohy objektu

GraphBlock Grafický protějšek modelu bloku, tedy uzel grafu.

Důležité parametry:

- *model* – instance bloku v modelu (třída `AbstractBlock`)
- *typ* – odvozen od typu modelu bloku (tř. `BlockType`)
- *obrázek* – obrázek udávající základní vzhled bloku (tř. `RenderedImage`)
- *seznam konektorů* – grafická reprezentace vstupů a výstupů modelu bloku (seznam objektů tř. `GraphBlockConnector`)
- *tabulka* – tabulka pro zobrazení aktuálních parametrů modelu bloku (tř. `GraphBlockTable`)
- *tag* – jmenovka bloku (tř. `Tag`)
- *obdélník* – udává aktuální polohu a rozměry bloku
- *zvětšení/zmenšení* – parametr zmenšení nebo zvětšení oproti původní velikosti bloku
- *rotace* – počet 90-tistupňových rotací
- *převrácení* – booleovský parametr, zda je blok zrcadlově převrácený

Vzhled bloku je definován tak, že se na daný obrázek aplikuje afinní transformace složená ze zadané rotace, posunu, zvětšení/zmenšení a zrcadlení, teprve poté se obrázek vykreslí. Při vytváření nové instance třídy jsou parametry důležité pro vzhled bloku inicializovány na základě typu bloku (viz kap. 6.4).

GraphConnector Abstraktní třída reprezentující konektor, ke kterému je možné připojit proud.

Důležité parametry:

- *proud* – připojený proud (tř. `GraphFlow`)
- *orientace* – natočení konektoru

Potomci třídy:

- `GraphBlockConnector`
- `GraphFloatingConnector`

GraphBlockConnector Konektor bloku, reprezentuje místo k připojení vstupního nebo výstupního proudu bloku.

Důležité parametry:

- *blok* – grafický blok, který vlastní tento konektor (tř. `GraphBlock`)
- *offset* – posunutí konektoru vzhledem k majiteli

Pozice konektoru je odvozena od bloku pomocí parametrů orientace a offset.

GraphFloatingConnector Konektor, který označuje buď zdroj nástřikového proudu do procesu, nebo cíl výstupního proudu z procesu.

Důležité parametry:

- *střed* – pozice středu konektoru vzhledem k plátnu
- *connectionFlow* – umožňuje připojení k nástřikovému proudu nebo k výstupnímu proudu procesu (hodí se při napojování bloků nebo při tvorbě recyklu)

Pozice konektoru je odvozena od středu konektoru a jeho orientace.

GraphTable Abstraktní třída reprezentující grafickou tabulku.

Důležité parametry:

- *popup dialog* – výstup tabulky v samostatném dialogovém okně, které pro zobrazení tabulky využívá Javovskou třídu `JTable`
- *datová tabulka* – uchovává aktuální datovou strukturu tabulky; je založená na třídě Javy `JTable`, jelikož však není možné použít tuto třídu při vykreslení tabulky na plátno, implementuje vlastní vykreslovací metody (tř. `DataTable`)
- *line* – spojovací linie mezi tabulkou a objektem, který popisuje
- *aktivní pole* – seznam vlastností zdrojového objektu, které se mají v tabulce zobrazovat
- *dokovat* – booleovský parametr, zda se má tabulka připevnit ke zdrojovému objektu
- *stručná* – booleovský parametr, zda se má zobrazovat pouze stručná verze tabulky
- *relativní pozice* – používá se u připevněné tabulky, relativní pozice vzhledem ke zdrojovému objektu
- *absolutní pozice* – pozice vzhledem k plátnu, u připevněné tabulky se generuje z relativní pozice

Stručná verze tabulky se vždy automaticky upevňuje k objektu, jinak je dokování volitelné. U detailní verze tabulky se vždy vykresluje spojovací linie ke zdrojovému objektu.

Potomci třídy:

- `GraphBlockTable`
- `GraphFlowTable`

GraphBlockTable Tabulka pro bloky. Zobrazuje zvolené parametry zdrojového bloku.

Důležité parametry:

- *blok* – zdrojový blok (tř. `AbstractBlock`)

GraphFlowTable Tabulka pro proudy. Zobrazuje zvolené parametry zdrojového proudu.

Důležité parametry:

- *flow* – zdrojový proud (tř. `AbstractFlow`)

GraphFlow Grafický protějšek modelu proudu, tedy hrana grafu. Hrany musejí vést v pravých úhlech, střídají se horizontální a vertikální segmenty. Každá hrana musí spojovat dva konektory (kromě hrany právě vytvářené, ta spojuje jeden konektor s aktuální pozicí kurzoru). Uživatel může podobu hrany postupně „naklikat“, nebo přímo propojit dané konektory, čímž se hrana vygeneruje automaticky (je snaha o inteligentní obcházení zdrojového a cílového bloku či konektoru).

Důležité parametry:

- *model* – instance proudu v modelu (tř. `AbstractFlow`)
- *křivka* – grafická křivka reprezentující tvar hrany pro vykreslení (tř. FlowCurve)
- *typ* – typ modelu hrany (tř. `FlowEnum`)
- *styl* – vyjádření tloušťky, barvy a vzoru hrany (tř. `FlowStyle`)
- *zdroj* – zdrojový konektor hrany (tř. `GraphConnector`)
- *cíl* – cílový konektor hrany (tř. `GraphConnector`)
- *tabulka* – tabulka pro zobrazení aktuálních parametrů, složek a přísad modelového proudu (tř. `GraphFlowTable`)
- *kontrolní body* – speciální body, které vymezují polohu a tvar hrany (seznam objektů tř. `FlowControlPoint`)
- *dočasné body* – dočasné body vymezují křivku jen v podobě střídajících se souřadnic, používají se při vytváření křivky
- *aktivní kontrolní bod* – index kontrolního bodu, nad kterým je v daném momentě kurzor myši

Při uživatelském vytváření křivky se postupně ukládají dočasné body. V momentě, kdy je křivka dokončena, se dočasné body převedou na kontrolní body a dočasný seznam je smazán. Křivka může být vygenerována z dočasných i kontrolních bodů (samostatně i kombinovaně).

6.2.2 Správce plátna

Úkolem správce plátna (tj. třídy `CanvasManager`) je stanovit pravidla interakce s plátnem. Rozsah funkčnosti této třídy je skutečně velký, proto používá množinu podsprávců, kteří se starají o různé dílčí činnosti. Nejdůležitější podstatou funkčnosti správce plátna je spolupráce s třídou `MouseManager`. Ta totiž rozděluje role mezi tyto třídy tím způsobem, že `MouseManager` vystupuje coby komponenta řídící tok dat a `CanvasManager` vystupuje jako komplexní stavový stroj s rozsáhlou knihovnou funkcí pro práci s plátnem.

Nejprve projdu seznam nižších správců, které správce plátna využívá a ovládá:

Správce pro objekty, třída CanvasPropertyManager Instance této třídy udržuje seznamy všech grafických objektů vykreslených na plátně. Poskytuje metody pro přidávání a odebírání objektů a dále různé typy iterátorů pro pohodlné procházení různých podmnožin objektů.

Správce selekce, třída SelectionManager Správce selekce není tvořen pouze zmíněnou hlavní třídou, ve skutečnosti jde o celý balíček tříd ***w2e.view.canvas.selection***. Správce udržuje množinu objektů plátna, které jsou aktuálně vybrány (označeny). Dále poskytuje metody pro vytváření selekcí (zjištění, které prvky jsou uvnitř selekčního čtverce), vykreslování grafických prvků selekce a pro provádění hromadných operací nad selekcemi. Úkolem správce je také zobrazovat u vybraných objektů množinu povolených geometrických operací (to platí hlavně pro bloky a samostatné konektory) a zpřístupnit tyto operace uživateli.

Správce upravovaného objektu, třída EditingManager V porovnání se správcem selekce jde o velmi jednoduchou třídu. Udržuje informaci pouze o jediném grafickém objektu, totiž o objektu, ke kterému je právě otevřen postranní panel pro úpravu daného objektu (viz kap. 6.3.2). Také nese malé množství informací potřebných k vykreslení označení upravovaného prvku.

Správce vyskakovacích menu, třída PopupMenuManager Úkolem tohoto správce je generování menu, jež vyskakují při stisknutí pravého tlačítka myši. Základní informací, kterou správce potřebuje, je aktuální objekt pod kurzorem myši, zvolený režim a existence výběru nad objekty. Tento správce přistupuje k třídě `CanvasManager` přes speciální rozhraní `CanvasController4Popup`, jež mu dává možnost provádět veškeré potřebné změny plátna a jeho obsahu. Velké množství operací je dostupné pouze přes menu vygenerované tímto správcem, například nastavení vlastního obrázku bloku, úprava kontrolních bodů hrany a mnoho dalších.

Již bylo zmíněno, že správce plátna je v první řadě stavovým strojem. Obsahuje velmi detailní informaci o aktuálním stavu plátna, která je tvořena obsahem plátna, aktuálně vybranými volbami GUI a pozicí kurzoru (případně ještě posloupností posledních provedených činností). Pokud neběží simulace a není hýbáno myší, správce plátna má vygenerovanou určitou stavovou informaci a vyčkává na podněty z okolních komponent. Na základě pohybu myši či podnětu od jiné třídy GUI správce vyhodnotí nový stav a provede potřebné činnosti spojené s přechodem do nového stavu. Pokud je k danému stavu dostupná nápověda či nějaký komentář, zobrazí jej správce do *stavového řádku* hlavního okna aplikace.

Každá změna stavu je obvykle spojena s potřebou překreslení plátna. V tomto ohledu dostala přednost rychlost vykreslování před paměťovou náročností aplikace, proto si třída `CanvasManager` ukládá pět různých úrovní grafických kontextů a podle požadavku na překreslení přegeneruje jen ty nejnutnější součásti plátna. Úrovně překreslování je možné popsat následujícím způsobem:

1. **aktuální objekt** – překresluje se pouze objekt pod kurzorem myši (a součásti rozhraní, které se překreslují vždy, například selekční čtverec)
2. **sousedství aktuálního objektu** – objekty ve schématu přímo sousedící s daným objektem
3. **výběr** – všechny označené objekty
4. **sousedství výběru** – všechny prvky, které sousedí s některým označeným objektem
5. **vše** – přesněji vše, co není na ostatních kontextech

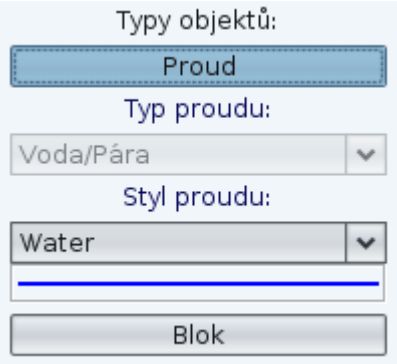
Zvláštní součástí správce plátna je také systém ukládání a načítání obsahu plátna. K vytvoření perzistetní reprezentace aktuálního modelu se používá tříd speciálního balíčku ***w2e.model.io***. Tento balíček je úmyslně umístěn do modelu, aby byly třídy určené pro export dat chráněny před změnami modelu i rozhraní. Úkolem tohoto balíčku je zapouzdřit pouze nejnutnější vlastnosti grafických i modelových objektů (ořezat stavové informace) a umět tyto informace ukládat do souborů. O detaily implementace uložení a načtení dat se starají Javovské třídy pro práci s formátem XML. Hlavní starostí správce plátna je nastavení prostředí podle dat načtených ze souboru a vyrovnání se s různými verzemi souborů a aplikace.

6.3 Postranní panely

Postranní panely jsou (s výjimkou vyskakovacích menu) jediné ovládací prvky rozhraní, které mění svůj obsah na základě stavu plátna a rozhraní. Úkolem postranních panelů je nabízet specializované komponenty pro uživatelem vyžádané akce. Pokud aktuální činnost uživatele nevyžaduje žádné z těchto komponent, postranní panel jednoduše zmizí a plátno se roztáhne po celé šířce aplikace.

6.3.1 Panel pro vkládání

Objeví se vždy po přepnutí rozhraní do režimu vkládání nových objektů. Poskytuje prostředky pro volbu typu a modelu vkládaného objektu. V případě, že aktuálně zvoleným typem objektu je proud, dává uživateli také možnost volby stylu jeho vykreslení (obr. 6.3). Pokud uživatel najede kurzorem na konektor, dokáže panel rozpoznat typ a styl proudu vedoucího do nebo z konektoru.



Obrázek 6.3: Panel pro vkládání (při vkládání proudu)

6.3.2 Panely pro úpravu modelů

Panely pro úpravy modelu jsou obecně odvozeny od třídy `AbstractFormGenerator`. Tato třída je určená pro tvorbu formulářů, které je třeba generovat na základě vlastností objektů modelu. Formuláře využívají upravené implementace komponent Javy, které jsou uloženy v balíku ***w2e.view.form.components***. Z těchto komponent se pak generují složitější komponenty, které vždy představují určitou vlastnost modelu. V balíku ***w2e.view.form.edit*** je pět těchto složitějších komponent:

- `BlockFormGenerator` – vygeneruje komponentu pro nastavení parametrů bloku
- `FlowChooserGenerator` – vygeneruje komponentu pro nastavení typu proudu
- `FlowFormGenerator` – vygeneruje komponentu pro nastavení parametrů proudu
- `FlowComponentGenerator` – vygeneruje komponentu pro výběr a nastavení složek proudu
- `FlowIngredientGenerator` – vygeneruje komponentu pro výběr a nastavení přísad proudu

Zbylé dvě třídy balíčku ***w2e.view.form.edit*** představují implementace postranních panelů:

Panel pro úpravu proudu, třída SimpleFormGenerator Vygeneruje popis proudu, pod který vloží panel pro úpravy vlastností proudu. Do tohoto panelu se vygeneruje volič typu proudu, parametry a tlačítka pro přístup ke složkám a přísadám proudu. Výsledek je vidět na obrázku 6.4.

Panel pro úpravu bloku MainFormGenerator Do panelu se nejprve vygeneruje pole pro úpravu názvu bloku, pod něj tlačítka pro vstupní proudy, dále parametry bloku a nakonec tlačítka pro výstupní proudy. Po stisknutí tlačítka proudu se pod tlačítko vygeneruje panel pro úpravu daného proudu. Pro výsledek viz obr. 6.5.

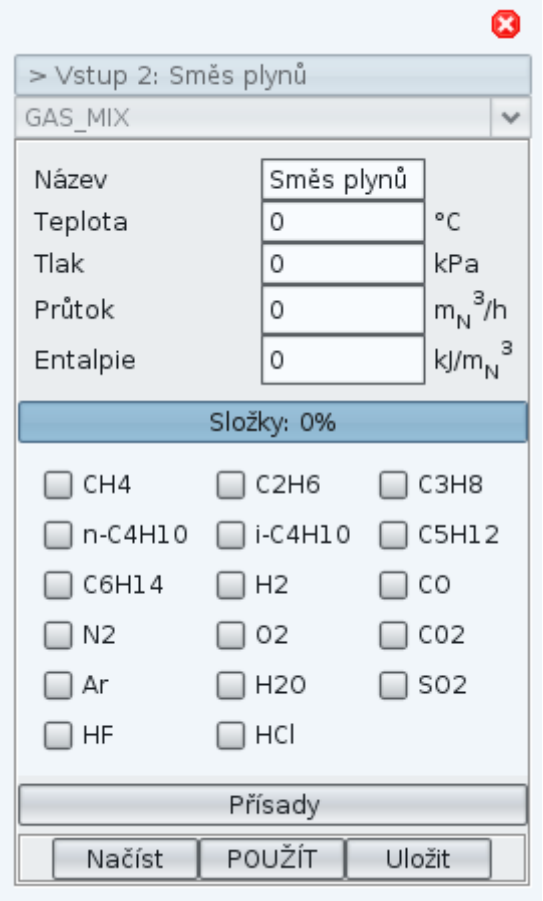
6.4 Správa konfigurace rozhraní

Na několika místech práce se zmiňujeme o schopnosti rozhraní uchovávat si konfiguraci. Pro tento účel je v balíčku ***w2e.view*** třída `Configuration`, která je zapouzdřením veškeré konfigurace aplikace W2E. Pro ukládání, resp. načítání konfigurace do, resp. ze souboru je využíváno Java API pro práci s XML. Třída `Configuration` uchovává tyto údaje:

- zvolený vzhled rozhraní
- zvolený jazyk
- seznam předdefinovaných stylů proudů
- seznam nastavení pro jednotlivé typy bloků
- tabulku, která každému typu proudu přiřazuje jeden ze stylů

O stylech proudů a typech bloků padlo již v textu několik zmínek, ale do tohoto okamžiku nebyl nikde uveden žádný bližší popis. Narozdíl od třídy `Configuration` jsou veškeré třídy definující součásti konfiguračního systému implementovány ve zvláštních balíčcích:

- ***w2e.view.settings*** – kořenový balíček pro třídy implementující součásti konfiguračního systému; obsahuje třídu implementující konfigurační dialog (`SettingsDialog`)
- ***w2e.view.settings.blocktypes*** – obsahuje třídy pro konfiguraci bloků, jmenovitě `BlockTypes` a formulář pro změnu konfigurace bloků `BlockTypeEditPanel`



Obrázek 6.4: Panel pro úpravu proudu

- ***w2e.view.settings.flowstyles*** – obsahuje třídy pro konfiguraci proudů, konkrétně `FlowStyle`, `LineStyles` a formulář pro změnu konfigurace proudů `FlowStyleEditPanel`

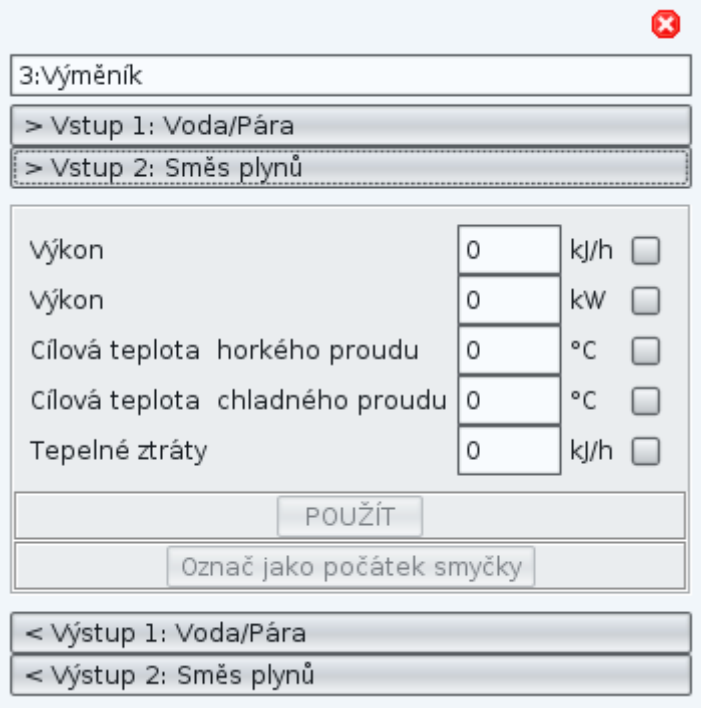
Třída `BlockTypes` umožňuje definovat základní konfiguraci pro určitý typ bloku. Konfigurace bloku zahrnuje tyto informace: typ modelu bloku, seznam konektorů (včetně nastavení jejich polohy), vlastní obrázek, zvětšení/zmenšení, rotace a převrácení bloku.

Třída `FlowStyles` pouze definuje určitý styl vykreslování proudu. Definice stylu zahrnuje tyto položky: název stylu, šířka křivky, index vzoru křivky a barva. Index vzoru křivky odkazuje na definici třídy `LineStyles`, která definuje sadu vzorů křivek.

V balíčcích je také definována třída dialogu `SettingsDialog`, kterou je možné vyvolat v aplikaci z hlavního menu a s jejím pomocím nastavit veškeré hodnoty konfigurace rozhraní. Ukázka jednoho z panelů dialogu nastavení je na obrázku 6.6.

6.5 Další komponenty rozhraní

Nakonec se ještě zmíním o dvou komponentách, které se nehodily do žádné z předcházejících kapitol, přesto však hrají v programu významnou roli a určitě stojí za zmíňku.



Obrázek 6.5: Panel pro úpravu bloku

6.5.1 Výpočetní formulář pro tuhá paliva

Jednou z forem rozšiřování aplikace je přidávání speciálních formulářů, jejichž úlohou je předzpracování dat. Tyto formuláře nijak nezasahují do simulačních výpočtů, jejich úkolem je usnadnit procesním inženýrům práci a s využitím dat zadaných v modelu předzpracovat požadovaným způsobem nějaká vstupní data.

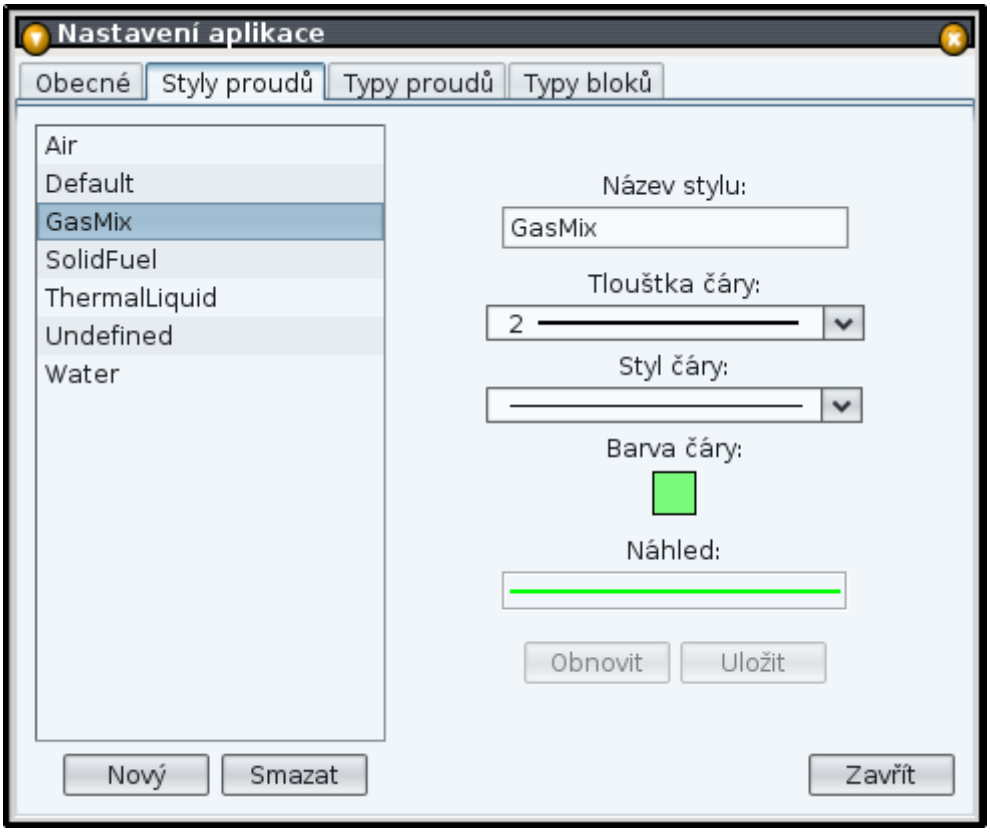
Prozatím je v aplikaci pouze jediný takový formulář a jeho implementace se nachází v balíčku ***w2e.view.form.solidfuel*** (grafická část) a v třídě `SolidWasteMath` balíčku ***w2e.model.flows*** (výpočty). Jeho úkolem je spočítat výhřevnost pevného paliva na základě jednoho (nebo více) z předdefinovaných vzorců. Dokud tento formulář v aplikaci nebyl, musela se výhřevnost počítat mimo aplikaci a následně zadávat do modelu. Jelikož hodnoty, se kterými vzorce pro výhřevnost operují, musíme do modelu stejně zadat, je výhodné mít formulář integrován v aplikaci, kde má k potřebným hodnotám automaticky přístup.

6.5.2 Třída ViewController

Během implementace simulátoru se ukázalo, že je vhodné umožnit modelu nějaký způsob komunikace s rozhraním. Rozhraní totiž celkem často potřebuje od modelu zpětnou vazbu, aby nedocházelo k nesynchronizovanému chování obou komponent (např. když by rozhraní požádalo model o provedení akce, kterou provést nelze, a přesto počítalo s jejím vykonáním). Z tohoto důvodu existuje třída `ViewController`, která modelu poskytuje sadu metod pro interakci s rozhraním.

Uvedu výčet nejvýznamnějších metod třídy `ViewController`:

- metody pro tvorbu a rušení grafických objektů



Obrázek 6.6: Konfigurační dialog W2E

- ▷ `createIOFlow(AbstractFlow, TechTypeEnum)` – vytvoří v rozhraní reprezentaci daného proudu
- ▷ `removeBlock(AbstractBlock)` – odstraní grafickou reprezentaci daného bloku
- ▷ `removeFlow(AbstractFlow)` – odstraní grafickou reprezentaci daného proudu

- metody upozorňující rozhraní na změny v modelu
  - ▷ `processNewData()` – oznamuje rozhraní dokončení výpočtů
  - ▷ `repaint()` – žádá rozhraní, aby překreslilo plátno
  - ▷ `simulationStopped()` – oznamuje ukončení simulace
  - ▷ `updateBlock(AbstractBlock)` – upozorňuje rozhraní na změnu dat daného bloku
  - ▷ `updateFlow(AbstractFlow)` – upozorňuje rozhraní na změnu dat daného proudu
  - ▷ `updateEditPanel()` – žádá rozhraní o obnovení dat postranního panelu
- metody žádající úpravu rozhraní
  - ▷ `hideEditPanel()` – žádá rozhraní o schování postranního panelu
  - ▷ `closeEditPanel()` – žádá rozhraní o schování a výmaz postranního panelu
  - ▷ `resizeSideToolBar()` – upozorňuje rozhraní na nutnost přegenerovat rozměry postranního panelu

    - ▷ `showFlowDetails(AbstractFlow)` – žádá rozhraní o zobrazení výpočetního formuláře pro daný proud

- metody pro zobrazení chybových zpráv
    - ▷ `showWarningDialog(String message, String warningType` – žádá rozhraní o zobrazení varovné zprávy
    - ▷ `showErrorDialog(String message, String errorType)` – žádá rozhraní o zobrazení chybové zprávy

Kapitola 7

Závěr

Cílem práce bylo navrhnout univerzální architekturu nástroje pro modelování a simulaci technologických procesů a implementovat tento nástroj v jazyce Java. Motivací pro vývoj zmíněného nástroje byla absence kvalitního a otevřeného simulačního nástroje, který by splňoval přísné požadavky pracovníků FSI VUT a umožnil jim provádění potřebných simulací v oboru zpracování biomasy.

Pokusím se ve stručnosti shrnout požadavky kladené na simulátor ze strany FSI:

1. sekvenčně modulární simulace, řešení základní simulační úlohy
2. řešení hmotnostních a energetických bilancí pro dané modely jednotkových operací
3. řešení recyklových proudů
4. rozšiřitelnost
5. otevřenost

Problematikou simulačních metod jsem se zabýval v kap. 3.3 a došel jsem k závěru, že sekvenčně modulární simulace je nejvhodnější volbou pro daný nástroj. Návrh architektury byl vytvořen tak, aby se simulátor vyrovnal s recyklovými proudy a aby jej šlo snadno přizpůsobit libovolným požadavkům ohledně modelovaných procesů. Implementace pak probíhala za neustálé spolupráce s FSI, zvláště na implementaci modelů a vztahů mezi bloky a proudy byl kladen velký důraz. V souladu s přáním FSI se implementace modelů soustředila na modelování procesů spalování biomasy. Aplikace byla během vývoje i po dokončení jeho první fáze průběžně testována, což by mělo zaručovat správnost implementovaných výpočtů. Z uvedeného plyne, že nástroj určitě splňuje první tři požadavky.

Požadavek rozšiřitelnosti byl otestován tak, že pracovníci FSI v posledním roce do aplikace úspěšně přidali nové modely bloků i proudů (viz kap. 5.2). Zatím nebyl proveden pokus o implementaci nového typu procesů, na FSI však existují plány na využití aplikace W2E pro simulaci energetických bilancí prádelen, tudíž i tento požadavek bude brzy v plné míře otestován. Pokud jde o otevřenost aplikace, byly již provedeny kroky k poskytnutí aplikace veřejnosti pod licencí GNU GPL.

Vývoj aplikace W2E nadále probíhá ve spolupráci FIT a FSI a v plánu je mnoho nových rozšíření. V nejbližší budoucnosti se budu věnovat implementaci výpočtů emisí a jejich roli v modelech bloků a proudů. Dalším krokem pak bude vývoj průvodce nad simulací, který umožní řešit návrhovou úlohu a provádět různé vyhodnocování provedených simulací.

Literatura

[1] Sun Microsystems, Inc.: A Swing Architecture Overview. [online], Rev. 21.5.2009, http://java.sun.com/products/jfc/tsc/articles/architecture/.

[2] Wikipedia.org: Entalpie. [online], Rev. 21.5.2009, http://cs.wikipedia.org/wiki/Entalpie.

[3] Wikipedia.org: Enumerated type. [online], Rev. 21.5.2009, http://en.wikipedia.org/wiki/Enumerated_type.

[4] Wikipedia.org: Model-view-controller. [online], Rev. 21.5.2009, http://en.wikipedia.org/wiki/Model-view-controller.

[5] Felder, R. M.; Rousseau, R. W.: *Elementary principles of chemical processes*. John Wiley & Sons, Inc., třetí vydání, 2000, ISBN 978-04-7153-478-5.

[6] Fishwick, P. A.: *Simulation Model Design and Execution: Building digital worlds*. Prentice-Hall, 1995, ISBN 978-01-3098-609-2.

[7] Pavlas, M.: *Systém pro výpočet technologických parametrů procesů včetně energetických aspektů*. Dizertační práce, FSI VUT v Brně, Brno, 2008.

[8] Pavlas, M.; Stehlík, P.; Šikula, J.; aj.: Výpočtový systém pro analýzu využití energie u technologických linek. Výzkumná zpráva VZ-EU-UPEI 2004/08, FSI VUT v Brně, Brno, 2004.

[9] Požívil, J.; Vaněk, T.; Bernauer, B.: *Procesní systémové inženýrství*. Praha: Vysoká škola chemicko-technologická v Praze, 1997, ISBN 80-7080-311-8.

[10] Spell, B.: *Java: Programujeme profesionálně*. Praha: Computer press, 2002, ISBN 80-7226-667-5.

[11] Václavek, V.; Eckert, E.; Vaněk, T.: *Základy chemického systémového inženýrství*. Praha: Vysoká škola chemicko-technologická v Praze v Čs. redakci MON, 1990.

[12] Zakhourl, S.: *Java 6: Výukový kurz*. Brno: Computer press, 2007, ISBN 978-80-251-1575-6.

Seznam použitých zkratek a symbolů

LAR – lineární algebraické rovnice

LAE – linear algebraic equations

NAR – nelineární algebraické rovnice

NAE – non-linear algebrauc equations

ODR – obyčejné differenciální rovnice

ODE – ordinary differential equations

DAR – differenciální algebraické rovnice

DAE – differential algebraic equations

PDR – parciální diferenciální rovnice

PDE – partial differential equations

MVC – Model-View-Controller (softwarová architektura)

IAPWS IF-97 – International Association for Properties of Water and Steam Industrial Formulation 1997

DLL – dynamic-link library

GUI – graphical user interface

XML – extensible markup language

API – application programming interface

GNU GPL – GNU General Public Licence (GNU = GNU's Not Unix)