VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

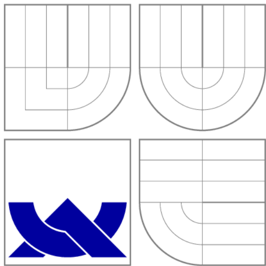
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

WEBOVÁ APLIKACE PRO VZDÁLENÉ MODELOVÁNÍ A SIMULACI NA BÁZI DEVS

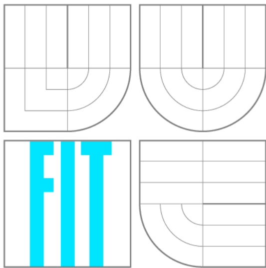
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PAVEL GAVLÍK
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

WEBOVÁ APLIKACE PRO VZDÁLENÉ MODELOVÁNÍ A SIMULACI NA BÁZI DEVS

WEB APPLICATION FOR REMOTE DEVS-BASED MODELLING AND SIMULATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PAVEL GAVLÍK
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.
SUPERVISOR

BRNO 2013

// Abstrakt

Práce se zabývá návrhem a realizací webové aplikace pro vzdálený přístup k nástroji SmallDEVS, implementaci formalismu DEVS. Komunikace po síti je realizována pomocí RESTful API. Webová aplikace je napsána převážně v programovacím jazyce JavaScript s využitím frameworku AngularJS. Při vývoji webové aplikace byla použita technika Test Driven Development.

Abstract

This paper deals with the design and implementation of a web application for remote access to SmallDEVS, implementation of DEVS formalism. Network communication is done using RESTful API. The web application is written mostly in JavaScript programming language using the framework AngularJS. Web application was developed using the technique of Test Driven Development.

Klíčová slova

DEVS, SmallDEVS, Smalltalk, REST, JavaScript, AngularJS, Test Driven Development

Keywords

DEVS, SmallDEVS, Smalltalk, REST, JavaScript, AngularJS, Test Driven Development

Citace

Pavel Gavlík: Webová aplikace pro vzdálené modelování a simulaci na bázi DEVS, bakalářská práce, Brno, FIT VUT v Brně, 2013

Webová aplikace pro vzdálené modelování a simulaci na bázi DEVS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Vladimíra Janouška, Ph.D.

. . . .
Pavel Gavlík
15. května 2013

Poděkování

Touto cestou děkuji panu doc. Ing. Vladimíru Janouškovi, Ph.D. za nasměrování a poskytnutí cenných rad během vypracovávání bakalářské práce.

© Pavel Gavlík, 2013.
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 **2**

2 Simulace a návrh vyvíjejících se systémů ve SmallDEVS **3**
2.1 Smalltalk . . . 4
2.2 SmallDEVS . . . 5
2.3 Grafické uživatelské rozhraní SmallDEVS a jeho prvky . . . 6

3 Tvorba webových aplikací s RESTful API pomocí frameworku AngularJS **9**
3.1 Representational State Transfer . . . 9
3.2 Webový prohlížeč jako platforma pro klientskou aplikaci . . . 10
3.3 Framework AngularJS . . . 12
3.4 JavaScript pro vývoj serverových aplikací . . . 14
3.5 Další nástroje pro vývoj webových aplikací . . . 14

4 Návrh řešení **16**
4.1 Specifikace požadavků . . . 16
4.2 Návrh síťové komunikace . . . 17
4.3 Návrh serveru . . . 18
4.4 Návrh klientské aplikace . . . 20

5 Implementace **23**
5.1 Implementace klientské aplikace . . . 23
5.2 Implementace serveru . . . 26

6 Testování aplikace a vyhodnocení výsledků **27**
6.1 Testování klientské aplikace pomocí automatických testů . . . 27
6.2 Testování funkčnosti . . . 28

7 Závěr **30**
7.1 Možnosti rozšíření . . . 30

A Obsah CD **33**

Kapitola 1

Úvod

Modelování a simulace je v rámci IT silný podobor s množstvím aplikací např. v biologii, fyzice či dopravě. I proto se objevilo množství nástrojů pro modelování a simulaci systémů. Jedním z takových nástrojů je i SmallDEVS, který je implementací formalismu DEVS. SmallDEVS umožňuje modelovat systémy pomocí prototypových objektů. Simulované systémy je možné upravovat a zkoumat i během simulace. Součástí SmallDEVS je grafické uživatelské rozhraní, které modifikaci a zkoumání modelů usnadňuje. Úvodem do problematiky modelování, simulací a popisem nástroje SmallDEVS se zabývá kapitola 2.

Tato práce rozebírá návrh a realizaci serveru, který nabízí služby jádra SmallDEVS. Klient k tomuto serveru by měl umožnit vzdálenou manipulaci se simulacemi podobným způsobem, jakým to umožňuje současné grafické uživatelské rozhraní. Pro komunikaci mezi serverem a klientem je nutné navrhnout vhodný protokol. Takový komunikační protokol by měl být dostatečně univerzální, aby jej bylo možné využít i při implementaci jiných klientů, jako např. aplikace zobrazující datový výstup simulace na grafu. O stylu architektury REST, který lze využít pro návrh komunikačního protokolu, pojednává kapitola 3.

World Wide Web prošel překotným vývojem. Už neslouží pouze pro přenášení statických webových stránek, ale je možné vytvářet aplikace, které obsahují funkcionalitu donedávna běžnou pouze u desktopových aplikací. I přes pokrok v technologiích používaných ve webových aplikacích neztratil web jednu ze svých zásadních vlastností – portabilitu. V porovnání s desktopovou aplikací je nutné vyvinout mnohem nižší úsilí, aby webovou aplikaci bylo možné spustit na různých operačních systémech a druzích zařízení. Klienta pro manipulaci se simulacemi je tedy vhodné navrhnout a implementovat jako webovou aplikaci. Kapitola 3 popisuje také technologie používané při vývoji webových aplikací.

Kapitola 4 obsahuje návrh serveru, síťového protokolu a webové aplikace. V kapitole 5 je rezebrána realizace návrhu z předchozí kapitoly. Kapitola 6 popisuje, jak je implementovaný kód otestován a jak se používá klientská aplikace.

Kapitola 2

Simulace a návrh vyvíjejících se systémů ve SmallDEVS

Než začneme popisovat nástroj SmallDEVS, je nutné zavést několik základních pojmů.

Systém je abstraktní koncept, který definuje chování entit v čase. Popisuje výstupní chování na základě vstupu a stavové informace. Zeigler [ZPK00] definuje tři základní modelovací formalismy: Systém popsaný diferenciálními rovnicemi, systém s diskrétním časem a systém s diskrétními událostmi.

Discrete Event System Specification (zkráceně DEVS) je specifikace systému s diskrétními událostmi. Neformálně lze systém s diskrétními událostmi popsat následujícím způsobem: Systém má vstupy a výstupy pozorovatelné jako události. Na některé vstupy systém reaguje výstupem (okamžitě nebo se zpožděním), na některé viditelně nereaguje (pouze přechází mezi vnitřními stavy). Někdy generuje výstup bez přímé vnější příčiny. Uvnitř systém přechází mezi vnitřními stavy, a to buď samovolně (poté, co v daném stavu strávil jistý čas), nebo na základě vnější události (události na vstupu). Výstup vždy závisí jen na aktuálním stavu a je pozorovatelný jako událost při samovolném přechodu systému do stavu následujícího. [Jan11, ZPK00]

Definice 2.0.1. *Základní model* je sedmice $M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta)$, kde

X je *množina vstupních událostí*,
S je *množina stavů*,
Y je *množina výstupních událostí*,
$\delta_{int} : S \longrightarrow S$ je *interní přechodová funkce*,
$\delta_{ext} : Q \times X \longrightarrow S$ je *externí přechodová funkce*, kde
  $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ je *množina úplných stavů*,
  e je *čas uplynulý od poslední události*,
$\lambda : S \longrightarrow Y$ je *výstupní funkce*,
$ta : S \longrightarrow \mathbb{R}_0^+ \cup \{\infty\}$ je *funkce posuvu času*.

Uvedený formalismus umožňuje specifikovat jakýkoli systém s diskrétními událostmi. Chápeme-li ale systém jako hierarchickou strukturu, tak výše uvedená definice odpovídá specifikaci atomického modelu, základní nedělitelné jednotce systému. Systémy složené ze subsystémů pak specifikujeme jako složené modely. [Jan11]

Pro specifikaci systému nemusí být vždy výhodné použití formalismu DEVS. Je možné použít jiné formalismy (např. Petriho sítě) a zapouzdřit jejich simulátor do atomické komponenty DEVS. [Jan11]

Definice 2.0.2. *Složený model* je sedmice $N_{self} = (X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, select)$, kde

X je *množina vstupních událostí,*
Y je *množina výstupních událostí,*
D je *množina jmen submodelů,*
$\{M_d \mid d \in D\}$ je *množina submodelů,*
$\{I_d \mid d \in D \cup \{self\}\}$ je *specifikace propojení,*
$\forall d \in D \cup self : I_d \subseteq D \cup \{self\}, d \notin I_d,$
$\{Z_{i,d} \mid i \in I_d, d \in D \cup \{self\}\}$ je *specifikace překladu událostí,*
$Z_{i,d} : X \longrightarrow X_d$ pro $i = self,$
$Z_{i,d} : Y_i \longrightarrow Y$ pro $d = self,$
$Z_{i,d} : Y_i \longrightarrow X_d$ pro $i \neq self$ a $d \neq self,$
$select : 2^D - \{\} \longrightarrow D$ je *preferenční funkce.*

Hierarchie je dalším konceptem DEVS. Systém je možné rozdělit na subsystémy, které jsou propojeny a mohou na sebe působit pomocí událostí.

2.1 Smalltalk

V této kapitole rozebereme Smalltalk, programovací jazyk a prostředí, ve kterém byl implementován nástroj SmallDEVS. Pro účely této práce použijeme Squeak – open source implementaci Smalltalku, pro kterou je k dispozici poslední stabilní verze SmallDEVS. Popisem základů Squeaku se zabývá kniha Squeak by Example [BDNP09].

2.1.1 Smalltalk jako programovací jazyk

Smalltalk je založen na posílání zpráv mezi objekty. Při vyhodnocování výrazu mají nejvyšší prioritu zprávy unární (s jedním argumentem), poté binární (se dvěma argumenty) a zprávy s klíčovými slovy. Prioritu vyhodnocování lze změnit použitím závorek. Jinak se zprávy vyhodnocují zleva doprava. Aritmetická pravidla (přednost násobení před sčítáním) nejsou zachována, proto je nutné vždy použít závorky.

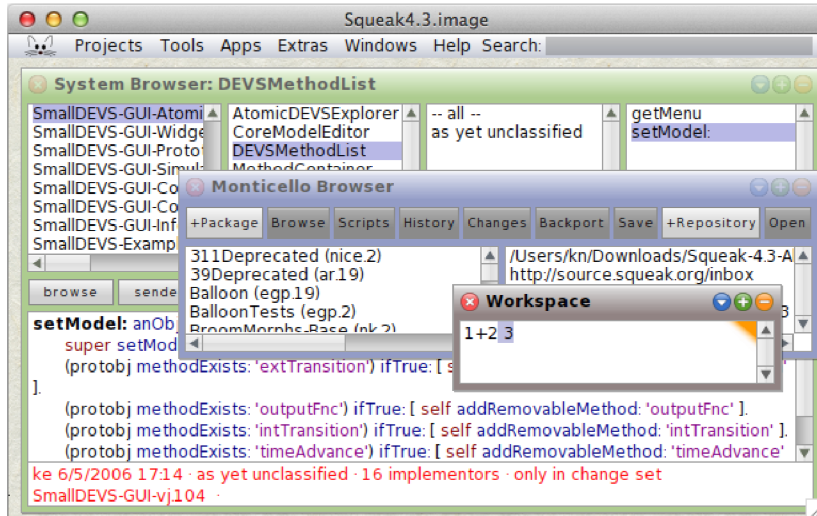
```
3 factorial "unární zpráva, objektu 1 je zaslána zpráva factorial"
1 + 2       "binární zpráva, objektu 1 je zaslána zpráva + s argumentem 2"
2 raisedTo: 6 modulo: 10   "objektu 2 je zaslána zpráva raisedTo:modulo:"
                           "s argumenty 6 a 10"
2 + (2*3)   "příklad nutnosti použití závorek"
```

Smalltalk pracuje s tzv. živými objekty. To znamená, že při vývoji aplikace vidíme stav těchto živých objektů, který pomocí nástrojů vývojového prostředí můžeme upravovat. Vývojové prostředí Smalltalku umožňuje i úpravu zdrojového kódu během ladění programu.

2.1.2 Vývojové prostředí Smalltalku

Smalltalk obsahuje pro vývoj aplikace mnoho užitečných nástrojů s grafickým rozhraním. System browser zobrazuje všechny definované třídy. S jeho pomocí je možné třídy upravovat, přidávat či mazat. Pro uložení je nutné kategorii, třídu, resp. metodu akceptovat. Během akceptace je ověřen zdrojový kód a jsou objeveny možné chyby, např. pokud prostředí najde nedeklarovanou proměnnou, tak nabídne automatickou úpravu zdrojového kódu, aby k deklaraci došlo.

Pro ověření nebo provedení krátkých výrazů můžeme použít Workspace. Inspector a Explorer jsou dva nástroje, které slouží pro zkoumání živých objektů. Pro import, export, verzování a sdílení balíčků tříd slouží Monticello. Některé z uvedených nástrojů se nachází na obrázku 2.1. Podrobnější popis nástrojů lze najít v knize Squeak by Example [BDNP09].



Obrázek 2.1: Prostředí Smalltalku (implementace Squeak 3.9)

Framework Morphic slouží pro implementaci grafických uživatelských rozhraní ve Smalltalku. Grafická uživatelská rozhraní ve frameworku Morphic se skládají z malých prvků, tzv. morphů. Morphy lze skládat do větších celků a sestavit tak složitější rozhraní. Každý morph může mít nastaven objekt, ze kterého načítá data, která se zobrazí uživateli. Tento objekt je označován jako model a je nastaven např. pomocí metody `setModel`.

2.2 SmallDEVS

Tato kapitola se zabývá popisem jádra SmallDEVS a jeho částí. Dokumentace jádra SmallDEVS se nachází na webových stránkách projektu [Jan12] v sekci Kernel API.

Ve SmallDEVS se pro modelování používají prototypové objekty, což umožňuje dynamické modifikace systému (viz [Jan11]). Perzistence modelů je ve SmallDEVS řešena uložením do hierarchického úložiště `MyRepository`. Jednotlivé modely jsou instancemi tříd `AtomicDEVSPrototype` nebo `CoupledDEVSPrototype`. Sdílené chování atomických modelů lze uložit do instance třídy `AtomicDEVSTrait`.

2.2.1 MyRepository

Třída `MyRepository` implementuje ve SmallDEVS perzistentní úložiště objektů. Úložiště `MyRepository` má formu stromové struktury. Kořenový uzel obsahuje po instalaci SmallDEVS několik složek (např. Documents, Simulations a TMP). Složky slouží pouze pro organizaci objektů, žádnou další funkci nemají. Simulace jsou umístěny zpravidla ve složce Simulations.

Každá simulace obsahuje ukazatel na objekt modelu, který má být simulován. Tento model může být složený či atomický.

`MyRepository` umožňuje přistoupit k objektům několika způsoby. Kořenový uzel vrací metod `root`. Potomky uzlu je možné získat pomocí method `componentNames` (vrací jména všech potomků) a `componentNamed:ifAbsent:` (vrátí potomka se známým jménem). K dispozici jsou také metody pro přidávání, odebírání či import objektů.

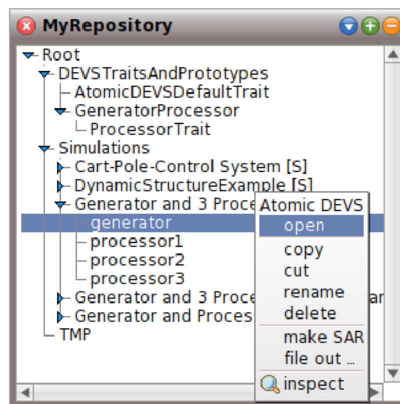
2.3 Grafické uživatelské rozhraní SmallDEVS a jeho prvky

Pro správný návrh klientské aplikace je nutné pochopit fungování grafického uživatelského rozhraní, které je součástí SmallDEVS. Toto rozhraní používá podobně jako jiná grafická uživatelská rozhraní ve Smalltalku framework Morphic. Většina prvků uživatelského rozhraní obsahuje metodu `setModel`, která po předání odpovídajího objektu z jádra SmallDEVS vykreslí data na obrazovku. Ostatní metody se starají zpravidla o zpracování uživatelského vstupu (z myši či klávesnice), nebo o podpůrné funkce při vykreslování na obrazovku.

Zbytek kapitoly se věnuje popisu jednotlivých prvků grafického uživatelského rozhraní SmallDEVS.

2.3.1 Prohlížeč repozitáře

Pro prohlížení objektů a manipulaci s objekty v repozitáři slouží prohlížeč repozitáře. Umožňuje objekty přidávat, přejmenovávat, mazat, vyjmout, kopírovat a vložit. Jeho implementace je uložena ve třídě `MyRepositoryBrowser`.



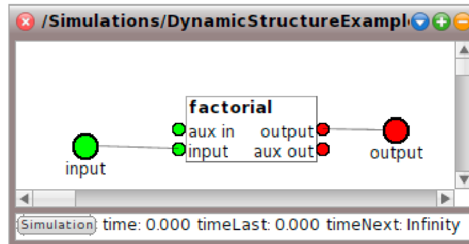
Obrázek 2.2: Prohlížeč repozitáře

V okně prohlížeče repozitáře (viz obrázek 2.2) je hierarchická struktura objektů zobrazena podobně jako v programech určených pro správu souborového systému. Stromovou strukturu repozitáře lze postupně procházet rozbalováním objektů. V repozitáři lze vytvořit složky, které v simulaci nemají žádnou funkci a slouží pouze jako kontejner pro objekty. Složené modely je také možné rozbalit a zobrazit tak objekty, které obsahují, ať už to jsou další složené modely nebo atomické modely. Atomické modely již pochopitelně dál rozbalovat není možné.

Prohlížeč repozitáře slouží také jako výchozí bod ke zkoumání objektů uvnitř repozitáře. Po označení objektu a výběru akce open v kontextovém menu (příp. po poklepání na objekt) je otevřeno okno odpovídající typu objektu.

2.3.2 Inspektor složených modelů

Tato část uživatelského rozhraní umožňuje prohlížet a editovat strukturu složeného modelu. V horní části okna inspektoru (viz obrázek 2.3) se nachází plocha s grafickou reprezentací jeho vnitřní struktury. Jsou zde zobrazeny vstupní či výstupní porty patřící právě zobrazenému složenému modelu, submodely včetně jejich vstupních či výstupních portů a propojení portů. V dolní části okna inspektoru je umístěn stavový řádek s aktuálním časem simulace.



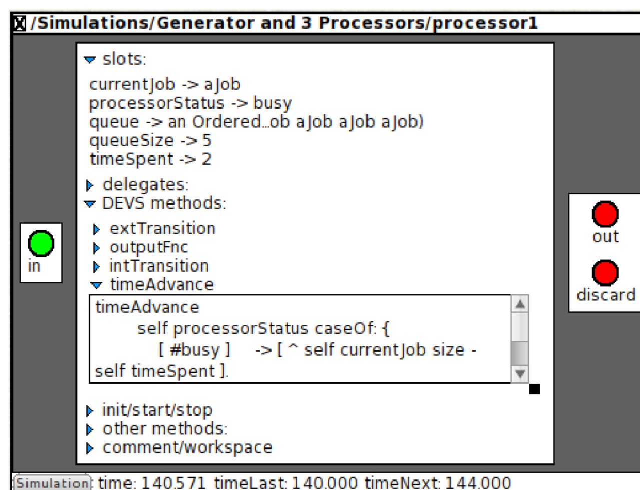
Obrázek 2.3: Inspektor složených modelů

Submodely i porty je možné přidávat, přejmenovávat, mazat a provádět operace se schránkou (kopírovat, vyjmout či vložit). Submodely lze pomocí operace open v kontextovém menu otevřít v inspektoru složených či atomických modelů (v závislosti na typu submodelu).

V rámci složeného modelu je možné upravovat propojení portů. Pokud uživatel provede propojení tam, kde to není možné (např. z výstupního portu zobrazeného složeného modelu do vstupního portu submodelu), je zobrazeno chybové hlášení.

2.3.3 Inspektor atomických modelů

Pro zkoumání a upravování atomických modelů je k dispozici tento inspektor. Inspektor implementuje třída `AtomicDEVSExplorer`, která vytváří jednotlivé části rozhraní inspektoru. Uživatelské rozhraní se skládá ze 4 částí (viz obrázek 2.4): stavového řádku s aktuálním časem simulace (dole), seznamu portů (vstupní porty jsou umístěny vlevo, výstupní porty vpravo) a hlavní stromové struktury s vlastnostmi modelu.

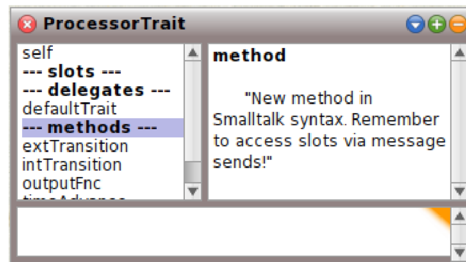


Obrázek 2.4: Inspektor atomických modelů

Stromová struktura modelu umožňuje manipulaci se sloty, delegáty, DEVS metodami, metodami specifickými pro různé fáze simulace, dalšími metodami, které nepatří do žádné z předchozích skupin a s komentářem k modelu. Po úpravě zdrojového kódu atomického modelu je nutné provedené změny potvrdit podobně jako při editaci zdrojového kódu v prostředí System Browser.

2.3.4 Inspektor prototypových objektů

Inspektor prototypových objektů (na obrázku 2.5) umožňuje zkoumání a úpravu prototypových objektů. Tato část grafického uživatelského rozhraní sice není součástí SmallDEVS, ale používá se při modelování systémů prototypovými objekty (viz [Jan11]).



Obrázek 2.5: Inspektor prototypových objektů

Prototypové objekty jsou kombinací třídy a její instance, proto je inspektor prototypových objektů kombinací prostředí Inspector a System Browser. Umožňuje zkoumat a modifikovat aktuální stav objektu, ale i manipulovat s delegáty či metodami. Nové sloty, delegáty či metody vytvoříme vybráním požadované skupiny, vyplněním hodnoty či kódu a akceptováním provedených změn. Existující sloty, delegáty či metody je možné po vybrání upravit či smazat. V dolní části okna je možné nad objektem vyhodnocovat kód podobně jako v okně inspectoru.

Prototypové objekty je možné klonovat. Výsledkem operace klonování, která je dostupná přes kontextové menu, je mělká kopie originálního objektu.

Kapitola 3

Tvorba webových aplikací s RESTful API pomocí frameworku AngularJS

3.1 Representational State Transfer

Pojem Representational State Transfer (zkráceně REST) zavedl poprvé Fielding ve své práci zabývající se návrhem architektur aplikací pracujících se sítí [Fie00]. REST je styl architektury pro distribuované systémy (např. World Wide Web). Více praktickými aspekty návrhu aplikací se stylem architektury REST se zabývá kniha RESTful Web Services [RR07], která zavádí pojem RESTful Web Service, což je webová služba, která používá principy REST.

REST pracuje s pojmem zdroj (anglicky resource). Každý zdroj má vlastní URL, což usnadňuje např implementaci cache. Webová služba pro každý zdroj vrací jeho reprezentaci.

Další vlastností REST je bezestavovost (anglicky statelessness). Klient musí serveru zaslat všechny informace nutné k sestavení odpovědi a server nesmí spoléhat na informace zaslané v předchozích požadavcích. Tato vlastnost umožňuje zasílat další požadavky na zcela jiné servery a tak zajistit škálovatelnost aplikace.

V REST jsou standardizovány názvy metod (GET, DELETE). Názvy metod jsou stejné napříč zdroji a službami. Výpis často používaných metod je uveden v tabulce 3.1. Pokud má zdroj velkou velikost, je vhodné implementovat metodu HEAD, která vrátí pouze metadata. Celá data (včetně metadat) jsou vrácena metodou GET.

Metoda	**Popis**
GET	získání celého zdroje (včetně metadat)
HEAD	získání metadat
POST	vytvoření nového zdroje
PUT	vytvoření či úprava zdroje se známou URL
DELETE	smazání zdroje

Tabulka 3.1: HTTP metody

RESTful webová služba má standardizován i způsob vracení chybových hlášení. Při použití protokolu HTTP je využito stavových kódů HTTP. Pokud nedošlo k chybě, je vrácen stavový kód z rozsahu 2xx, jinak je vrácen některý z kódů z rozsahů 3xx, 4xx nebo 5xx. Často používané stavové kódy jsou uvedeny v tabulce 3.2.

Při tvorbě klienta je vhodné využít možností knihovny, která pracuje s protokolem

Kód	Anglický název	Popis
200	OK	Požadavek byl úspěšný, jsou vrácena data.
201	Created	Podařilo se přidat zdroj.
204	No Content	Server požadavek úspěšně zpracoval, zároveň nemá smysl vracet žádná data
400	Bad Request	Požadavek na server není ve správném formátu.
401	Unauthorized	Klient nebyl autorizován pro přístup ke zdroji (např. nezaslal uživatelské jméno a heslo).
403	Forbidden	Server nechce danému klientovi povolit přístup k požadovanému zdroji.
404	Not Found	Zdroj nebyl na serveru nalezen.
405	Method Not Allowed	Server pro tento zdroj danou HTTP metodu nepodporuje.
409	Conflict	Tento požadavek by uvedl zdroje na serveru do nekonzistentního stavu.
500	Internal Server Error	Nastala chyba na straně serveru.

Tabulka 3.2: HTTP stavové kódy

HTTP. Takové knihovny jsou dostupné pro většinou programovacích jazyků. Před použitím knihovny je vhodné se ujistit, zda podporuje všechny výše uvedené vlastnosti protokolu HTTP. Existují i knihovny, které jsou určeny přímo pro tvorbu klientů k RESTful webových službám.

Data v odpovědi webové služby bývají zpravidla ve formátu XML nebo JSON. Formát JSON (JavaScript Object Notation) je vhodnější pro aplikace napsané v programovacím jazyce JavaScript, protože je možné načíst data pomocí funkce `eval` přímo do datového typu objekt. Před načtením je zpravidla provedena bezpečnostní kontrola, zda data neobsahují kód, který by se spustil v kontextu aplikace.

3.2 Webový prohlížeč jako platforma pro klientskou aplikaci

Pro vývoj klientské webové aplikace je nutné pochopit aspoň základy tří standardů – HTML, CSS a JavaScriptu.

HTML (Hypertext Markup Language) bylo navrženo pro označení částí dokumentu jako jsou nadpisy, odstavce či obrázky. Části dokumentu se uzavírají do tzv. značek, které mohou uvnitř obsahovat text nebo další značky. Např. nadpis první úrovně vypadá v HTML kódu takto:

```
<h1>Název stránky</h1>
```

CSS (Cascade Style Sheets) umožňují HTML dokumentu dodat požadovaný vzhled. V CSS kódu najdeme vždy několik pravidel. Pravidla obsahují selektor a blok deklarací. Selektor popisuje, na které části webové stránky budou styly aplikovány. Blok deklarací obsahuje jednotlivé deklarace, které se dál dělí na identifikátor vlastnosti a hodnotu vlastnosti. Důležitým konceptem CSS je kaskáda, což je použití hodnoty vlastnosti z nejkonkrétnějšího selektoru, pokud je vlastnost nastavena ve více pravidlech.

JavaScript je programovací jazyk, který běží ve webovém prohlížeči a dodává webové aplikaci dynamické chování. Pro interakci s HTML dokumentem je v JavaScriptu k dispozici tzv. Document Object Model (DOM). DOM umožňuje programátorovi přistupovat

ke stromové struktuře HTML dokumentu a provádět modifikace této struktury (vkládání, odebírání, změna obsahu značek).

3.2.1 Javascriptové frameworky

Při vývoji aplikací na straně webového prohlížeče se objevila nutnost strukturovat předhledněji zdrojový kód. To bylo hlavním důvodem vzniku javascriptových frameworků.

Většina současných javascriptových frameworků implementuje návrhový vzor MVC či jeho obměny. MVC se skládá ze tří částí – modelu, view a controlleru. Model obsahuje data. View vhodným způsobem zobrazuje data uživateli. Controller reaguje na události (většinou pocházející od uživatele) a odpovídajícím způsobem aktualizuje model či view.

Užitečným zdrojem pro porovnání JavaScriptových frameworků je projekt TodoMVC [Tod13]. TodoMVC obsahuje zdrojové kódy jednoduché aplikace pro správu úkolů. Porovnáním zdrojových kódů aplikace v různých frameworcích lze zjistit, jak se frameworky liší.

Některé číselné údaje týkající se nejpopulárnějších javascriptových frameworků jsou uvedeny v tabulce 3.3. První dva údaje se týkají velikosti komunit okolo daných projektů. Je uveden počet osob, které projekt sledují na serveru GitHub [git13], a počet otázek, které byly položeny na serveru StackOverflow [SO13]. Větší komunita okolo frameworku může znamenat vyšší kvalitu dokumentace, větší množství tutoriálů a článků. Menší rozsah komunity může naopak vést k nutnosti příliš často číst zdrojový kód frameworku a přínos použití frameworku nemusí být tak vysoký.

Posledním údaj v tabulce 3.3 je velikost souboru se zdrojovým kódem frameworku. Tento údaj je důležitý při vývoji webových aplikací, protože celý zdrojový kód je nutné aspoň jednou přenést ze serveru a při každém načtení aplikace interpretovat na straně klienta. Vyšší velikost frameworku může vést při pomalejší rychlosti internetového připojení k pomalejšímu prvnímu zobrazení stránky. Na méně výkonných zařízeních (zejména tablety a mobilní telefony) může interpretace velkého množství zdrojového kódu způsobit znatelné zpomalení startu aplikace.

	Počet GitHub stars	Počet otázek na StackOverflow	Velikost[1] [kB]
Backbone.js	13756	8409	19 (128/62)[2]
AngularJS	8846	5207	80
Ember.js	6754	3417	192 (295)
Knockout	3593	4626	41

Tabulka 3.3: Porovnání javascriptových frameworků

Výše uvedené údaje by neměly být jediným důvodem pro výběr frameworku. Mezi další důvody patří mimo jiné kvalita dokumentace, čitelnost výsledného zdrojového kódu aplikace či jeho testovatelnost.

[1] Velikosti odpovídají minifikovaným verzím knihoven. Frameworky se závislostmi mají v závorce uvedenu celkovou velikost včetně závislostí.

[2] Backbone.js závisí na knihovně Underscore.js. Při použití Backbone.Router a Backbone.View vyžaduje Backbone.js také knihovnu pro manipulaci s DOM. První hodnota v závorce odpovídá použití jQuery, druhá použití knihovny Zepto.

3.3 Framework AngularJS

Jedním z javascriptových frameworků je AngularJS od společnosti Google. Pro pochopení struktury klientské aplikace je nutné zavést několik pojmů, se kterými AngularJS pracuje, a stručně popsat fungování aplikace napsané v AngularJS. Detailní popis frameworku je možné nalézt v dokumentaci AngularJS [Ang12]. Užitečné tipy pro vývoj aplikací v AngularJS obsahuje kniha AngularJS [GS13].

3.3.1 Model, controller, scope a view

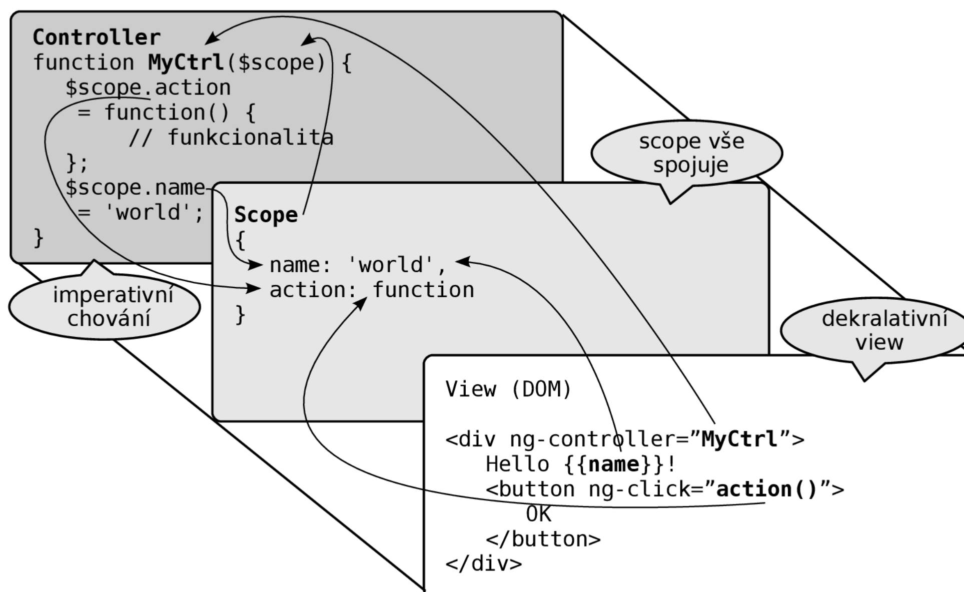
Narozdíl od jiných frameworků nedefinuje AngularJS vlastní bázové třídy pro tvorbu modelu[3]. Modelem může být v aplikaci jakýkoli datový typ, který je přiřazen do scope.

Controller se stará o konstrukci modelu a jeho přiřazení do scope. Zdrojový kód controlleru obsahuje také chování aplikace, např. zpracování dat z formuláře a odeslání těchto dat na server.

Scope uchovává aktuální data, která view prezentuje vhodným způsobem uživateli. Úkolem scope je také sledovat změny modelu, aby mohlo dojít k aktualizaci view. Instance scope mají hierarchickou strukturu podobně jako elementy DOM. Každá instance scope má pomocí prototypové dědičnosti přístup k vlastnostem rodičovského scope, čehož se často využívá ve view při přístupu k datům.

Zdrojovým kódem view je v AngularJS HTML šablona rozšířená o direktivy (viz níže) a výrazy v dvojitých složených závorkách – `{{expression}}`, které jsou později nahrazeny skutečným obsahem scope. View je tedy pouze deklarativní. Framework se sám stará o registraci DOM událostí pro akce uživatele a o změny obsahu DOM při aktualizaci scope.

Vzájemný vztah controlleru, scope a view je znázorněn na obrázku 3.1.



Obrázek 3.1: Znázornění fungování controlleru, scope a view (převzato z [Ang12])

[3] Model je zde pojem ve smyslu návrhového vzoru Model-View-Controller, nikoli pojem z modelování a simulací.

3.3.2 Direktiva

Direktiva (anglicky directive) umožňuje rozšiřovat jazyk HTML o naše vlastní značky, atributy a CSS třídy. AngularJS obsahuje množství zabudovaných direktiv, které najdou využití ve většině aplikací. Většina zabudovaných direktiv má předponu `ng`, zbylé direktivy se vážou na již existující HTML značky. Mezi některé z těchto zabudovaných direktiv patří:

- `ng-show`, `ng-hide` – v závislosti na pravdivosti výrazu, který je hodnotou atributu, zobrazí resp. skryje obsah elementu, ve kterém je tato direktiva
- `ng-class` – umožňuje elementu dynamicky přiřazovat CSS třídy
- `ng-app` – specifikuje jméno hlavního modulu aplikace
- `ng-controller` – stanovuje jméno controlleru obsluhujícího daný element
- `ng-model` – propojí element (zpravidla prvek formuláře) se scope a při změně obsahu elementu resp. scope je aktualizován scope resp. obsah elementu
- `input`, `select`, `textarea` – rozšiřují formulářové prvky o validátory, implementují validaci pro prohlížeče, které nepodporují tuto vlastnost HTML 5
- `ng-repeat` – umožňuje iterovat nad kolekcí dat a definovat HTML šablonu pro položky kolekce

Pro části aplikace, které je možné oddělit a příp. opětovně použít, je vhodné si definovat vlastní direktivy. Typickým příkladem pro definování vlastních direktiv jsou prvky uživatelského rozhraní jako rozbalovací nabídky, tlačítka pro sdílení na sociálních sítích, mapy či rozhraní pro výběr data.

K implementaci vlastní direktivy bychom měli přistoupit, pokud by se měl stejný (nebo velmi podobný) HTML kód vyskytovat na více místech aplikace nebo pokud používáme externí knihovnu, která pracuje s DOM a přímo nepodporuje AngularJS.

3.3.3 Service

Service slouží v AngularJS aplikaci k izolování zdrojového kódu, který se používá několikrát v jiných částech aplikace. Service neobsahuje zdrojový kod, který přímo souvisí s uživatelským rozhraním. Vlastní service je vhodné registrovat, pokud potřebujeme jistou část kódu použít ve více direktivách či controllerech a tento kód přímo nemanipuluje s DOM. Service je vlastně implementací návrhového vzoru Singleton. Každá service má unikátní jméno a funkci, která vrací instanci této service.

Součástí AngularJS je i několik zabudovaných services:

- `$http` – používá se pro komunikaci se serverem pomocí technologie AJAX.
- `$location` – umožňuje manipulovat s aktuální URL prohlížeče.
- `$document` a `$window` – poskytují přístup k objektům `document` a `window` z DOM, což je výhodné při testování aplikace.

Další zabudované services jsou popsány v dokumentaci AngularJS [Ang12].

3.3.4 Modul

Modul slouží k organizaci kódu v aplikaci napsané v AngularJS. Pro definování nového modulu nebo rozšíření stávajícího modulu zavoláme funkci `angular.module`. Definovanému modulu můžeme přiřadit části kódu (např. controllery či direktivy). Po načtení všech potřebných skriptů (událost `DomContentLoaded`) je spuštěn hlavní modul. Pokud modul závisí na jiných modulech, jsou načteny taktéž.

Moduly jsou v AngularJS řešením, jak se vyhnout definování globálních proměnných. Jsou také součástí implementace Dependency Injection, která se stará o vytváření instancí objektů a předává konstruktoru objektu jeho závislosti.

3.3.5 Testování AngularJS aplikací

AngularJS byl navržen s ohledem na automatické testování. Části frameworku byly napsány tak, aby byly snadno testovatelné a aby bylo možné psát testy i pro kód aplikace postavené nad AngularJS.

Pro načítání závislostí objektu používá AngularJS implementaci návrhového vzoru Dependency Injection. To umožňuje při testování jednotlivých částí kódu předat jejich závislosti. Závislosti, které nejsou při daném testu potřeba, je možné nahradit tzv. mock objektem.

AngularJS poskytuje pro účely testování modul `ngMock`. Service `$httpBackend` umožňuje definovat odpovědi na HTTP požadavky a testovat bez nutnosti spuštěného serveru, zda kód odeslal správné požadavky. Service `$timeout` je vhodná pro testování kódu, který obsahuje časovou prodlevu před spuštěním další části kódu. Po zavolání metody `flush` jsou okamžitě spuštěny všechny funkce čekající na dokončení časové prodlevy. Testování je tedy rychlejší.

3.4 JavaScript pro vývoj serverových aplikací

Programovací jazyk JavaScript je vhodný i pro vývoj serverové aplikace. K tomu je možné využít prostředí Node.js. Zvláštním rysem Node.js je asynchronnost prostředí. Při novém požadavku na server je vyvolána událost a je proveden kód, který je navázán na tuto událost.

Pro Node.js existuje množství knihoven pro různé účely. Tyto knihovny jsou dostupné zpravidla přes balíčkovací systém npm, který je součástí instalace Node.js. Balíčkovací systém npm umožňuje sledování aktualizací pro nainstalované balíčky, což zjednodušuje udržování aplikací postavaných nad Node.js. Součástí typické Node.js aplikace je soubor `package.json` obsahující jméno, popis, verzi aplikace a balíčky, na kterých aplikace závisí (jejich jména a verze). Po vyplnění souboru `package.json` stačí spustit příkaz `npm install`, který provede instalaci balíčků a jejich závislostí. Balíčkovací systém npm lze použít i pro instalaci nástrojů, které usnadní i vývoj frontend části aplikace.

3.5 Další nástroje pro vývoj webových aplikací

3.5.1 CSS preprocesory

Hlavním účelem tzv. CSS preprocesorů je zjednodušení kódu CSS. CSS preprocesory LESS, Stylus a SASS [SAS13] se mezi sebou liší hlavně syntaxí, všechny ale poskytují tyto vlastnosti:

- vnořování selektorů – Více specifické selektory není nutné stále opisovat, stačí je vnořit do méně specifických selektorů.
- proměnné – Hodnoty vlastností je možné uložit do proměnných. Proměnnou lze později použít na více místech a při případné změně není nutné procházet celý soubor.
- mixiny – Umožňují znovupoužít celé části kódu. Pomocí volitelných parametrů je tato vlastnost vhodná např. pro definici gradientů, u kterých webové prohlížeče používají tzv. vendor prefixy[4]

3.5.2 Kompilátory JavaScriptu

Při vývoji aplikací v programovacím jazyce JavaScript je nutné dávat pozor na velikost skriptů přenášených po síti. Velkou část přesených dat zaberou knihovny, které jsou nabídnuty ke stažení v minifikované verzi.

Pro odstranění nepotřebných mezer a jiné optimalizace vlastního zdrojového kódu slouží nástroje UglifyJS [ugl13] a Closure Compiler. Oba nástroje provádí parsování zdrojového kódu. Zdrojový kód je upraven tak, aby by při zachování původní funkcionality měl výsledný minifikovaný zdrojový kód menší velikost.

Closure Compiler obsahuje navíc mód `ADVANCED_OPTIMIZATIONS`, který provádí přejmenovávání funkcí na globální úrovni, odstraňování nepoužívaného kódu a nahrazení volání funkce přímo jejím kódem (pokud má kód po nahrazení menší velikost). Mód minifikace `ADVANCED_OPTIMIZATIONS` má však i nevýhodu. Je nutné psát zdrojový kód tak, aby by byl po zkompilování stále funkční. Některé frameworky (včetně AngularJS) však mód `ADVANCED_OPTIMIZATIONS` nepodporují.

3.5.3 Spouštěč úloh Grunt

Nástroj Grunt [gru13] slouží pro automatizaci opakovaných úloh při vývoji javascriptové aplikace. Grunt používá konfigurační soubor `Gruntfile.js`, který se nachází zpravidla v kořenovém adresáři aplikace. Soubor `Gruntfile.js` obsahuje konfiguraci jednotlivých úloh a definuje, kdy budou jednotlivé úlohy spuštěny a na které soubor se budou vztahovat.

Grunt mimo jiné podporuje tyto úlohy:

- compass – Slouží ke kompilaci SASS souborů.
- watch – Sleduje souborový systém a po změně souboru(ů) zavolá nastavenou úlohu.
- livereload – Po zavolání aktualizuje stránku v prohlížeči. Tato úloha se používá často v kombinaci s úlohou watch, není pak nutné provádět manuálně aktualizaci stránky.
- cssmin – Provádí minifikaci CSS souborů.
- uglify – Provádí minifikaci zdrojového kódu v programovacím jazyce JavaScript.
- concat – Spojí více souborů do jednoho. Tato úloha se často používá před minifikací javascriptových skriptů, aby se redukovalo množství provedených požadavků na server.

[4] Vendor prefix slouží pro otestování vlastnosti jádra webového prohlížeče, než je tato vlastnost standardizována. Jelikož se vendor prefixy mezi různými webovými prohlížeči liší, je nutné při použití novější CSS vlastnosti vypsat definice s vendor prefixy pro více webových prohlížečů.

Kapitola 4

Návrh řešení

V této kapitole jsou nejprve na základě předchozích zjištění z kapitoly 2.3 analyzovány požadavky na jednotlivé části projektu – komunikační protokol, server a klient. Poté je detailněji rozebráno možné řešení komunikačního protokolu, serveru a klienta.

4.1 Specifikace požadavků

- Zpřístupnit služby jádra SmallDEVS jako server.
  - Server by měl mít přehledný návrh.
  - Implementace serveru by neměla záviset na původním grafickém uživatelském rozhraní, měla by používat přímo jádro SmallDEVS.
- Navrhnout klienta pro webový prohlížeč, který bude vzdáleně manipulovat s modely podobně jako původní grafické uživatelské rozhraní.
  - Pro navigaci mezi jednotlivými části rozhraní aplikace navrhnout jednoduchý správce oken. Spráce oken by měl zobrazovat seznam oken, umožnit mezi nimi přepínat a okna zavírat.
  - Pro procházení `MyRepository` vytvořit stromovou strukturu, kterou lze postupným rozbalováním procházet. Po poklepání na položku repozitáře bude otevřeno okno s uživatelským rozhraním odpovídajím typu objektu.
  - Složené modely zobrazit jako graf, který bude obsahovat porty, submodely a propojení portů. Je možné přidat nové porty, submodely či propojení portů.
  - U atomické modely zobrazit porty, sloty, delegáty a metody. Umožnit manipulaci s nimi. Pro editaci metod zobrazit inteligentní editor se zvýrazňováním syntaxe.
  - U složených a atomických modelů zobrazit v dolní části okna aktuální čas simulace a tlačítka pro ovládání simulace (spuštění, zastavení, resetování).
- Pro komunikaci mezi serverem a klienty navrhnout vhodný protokol.
  - Protokol by měl umožnit tvorbu klientů pro různé účely (pro grafové výstupy, exportování dat či celé grafické uživatelské rozhraní) se zachováním jedné implementace serveru.

4.2 Návrh síťové komunikace

4.2.1 Architektura

Než začneme navrhovat detaily protokolu, je nutné specifikovat, jak bude probíhat komunikace mezi serverem a klientem.

Pravděpodobně nejjednodušším řešením by bylo generovat veškerá data včetně HTML kódu na straně serveru. Server by bylo možné realizovat jako součást virtuálního stroje obsahujícího jádro SmallDEVS. Při implementaci by bylo možné použít některý z frameworků pro tvorbu webových aplikací ve Smalltalku, např. Seaside. Toto řešení bohužel nesplňuje jeden z bodů specifikace, protože vygenerované statické stránky nejsou příliš vhodné pro klienty, kteří nejsou webovými prohlížeči.

Pro dosažení větší univerzálnosti serveru je vhodné posílat po síti pouze data bez HTML značek. Toho docílíme, pokud jádro SmallDEVS zpřístupníme pomocí RESTful API. Podpora REST je obsažena ve většině frameworků pro vývoj klientských aplikací v JavaScriptu, ale i v jiných jazycích, ve kterých bychom mohli implementovat jiné klientské aplikace. Stav simulace budeme při použití RESTful API zjišťovat pravidelným dotazováním na server.

Pokud budeme implementovat RESTful API, je výhodné použít technologii Cross-origin Resource Sharing, která umožní serveru komunikovat i s klienty spuštěnými na jiných portech či zcela jiných doménách. Tato architektura umožní klientům připojit se najednou k více serverům.

4.2.2 RESTful API

Při návrhu RESTful API je nutné se zaměřit především na podobu URL, přes kterou budeme přistupovat ke zdrojům a také na data, která se objeví v odpovědi serveru. Je vhodné zmínit i některé chybové stavy, k nimž může dojít, a způsob, jakým bude server na tyto stavy reagovat.

Výsledný návrh protokolu musí být bez problémů implementovatelný, proto je nutné zajistit, aby bylo možné deterministicky určit, který zdroj odpovídá dané URL. Toho docílíme přidáním řetězců jako `slots` či `delegates` do URL. Přidání těchto řetězců je nutné také kvůli přidávání nových položek, jinak by nebylo možné jednoduše rozlišit, zda přidáváme nový objekt do `MyRepository` nebo např. nový port modelu.

URL musí jednoznačně ukazovat na objekt, se kterým chceme pracovat. Jelikož objekty v `MyRepository` nemají unikátní identifikátory, musíme na ně odkazovat celou cestou v hierarchické struktuře `MyRepository`.

S přihlédnutím ke zmíněným požadavkům sestavíme návrh protokolu:

- Objekt z MyRepository
  `/<pathToObject>/`

  Pomocí tohoto zdroje budou načítány objekty všech typů. Data objektu, kterému jsou v hierarchické struktuře přiřazeny další objekty, budou obsahovat seznam těchto objektů, případně jejich dalších potomků. Pomocí URL `/` tak získáme celý obsah repozitáře. Pokud požadovaný objekt nelze nalézt, vrátí server stavový kód 404 Not Found.

  `POST` požadavkem na objekt typu složka nebo složený model bude přidán nový objekt do repozitáře. Pokud se klient bude snažit nahrát objekt, který již v repozitáři existuje, je vrácen stavový kód 409 Conflict.

- Port modelu (složeného či atomického)
  `/<pathToModel>/input_ports/<portName>/`
  `/<pathToModel>/output_ports/<portName>/`

  Jsou k dispozici kolekce pro vstupní a výstupní porty. Při odstranění portu jsou odebrána všechna jeho propojení.

- Propojení portů ve složeném modelu
  `/<pathToCoupledModel>/couplings/<couplingName>/`

- Slot atomického modelu
  `/<pathToAtomicModel>/slots/<slotName>/`

  Při úpravě hodnoty `value` je vyhodnocen zaslaný výraz a jeho hodnota je vložena do slotu. Výsledná hodnota výrazu je zaslána v odpovědi klientovi. Pokud dojde při vyhodnocování výrazu k chybě, tak je vrácen stavový kód 500 Internal Server Error spolu s textovým hlášením.

- Delegát atomického modelu
  `/<pathToAtomicModel>/delegates/<delegateName>/`

- Metoda atomického modelu
  `/<pathToAtomicModel>/methods/<methodName>/`

  Uživatelské rozhraní rozlišuje několik typů metod, ale jádro implementuje pouze jeden datový kontejner pro metody.

4.3 Návrh serveru

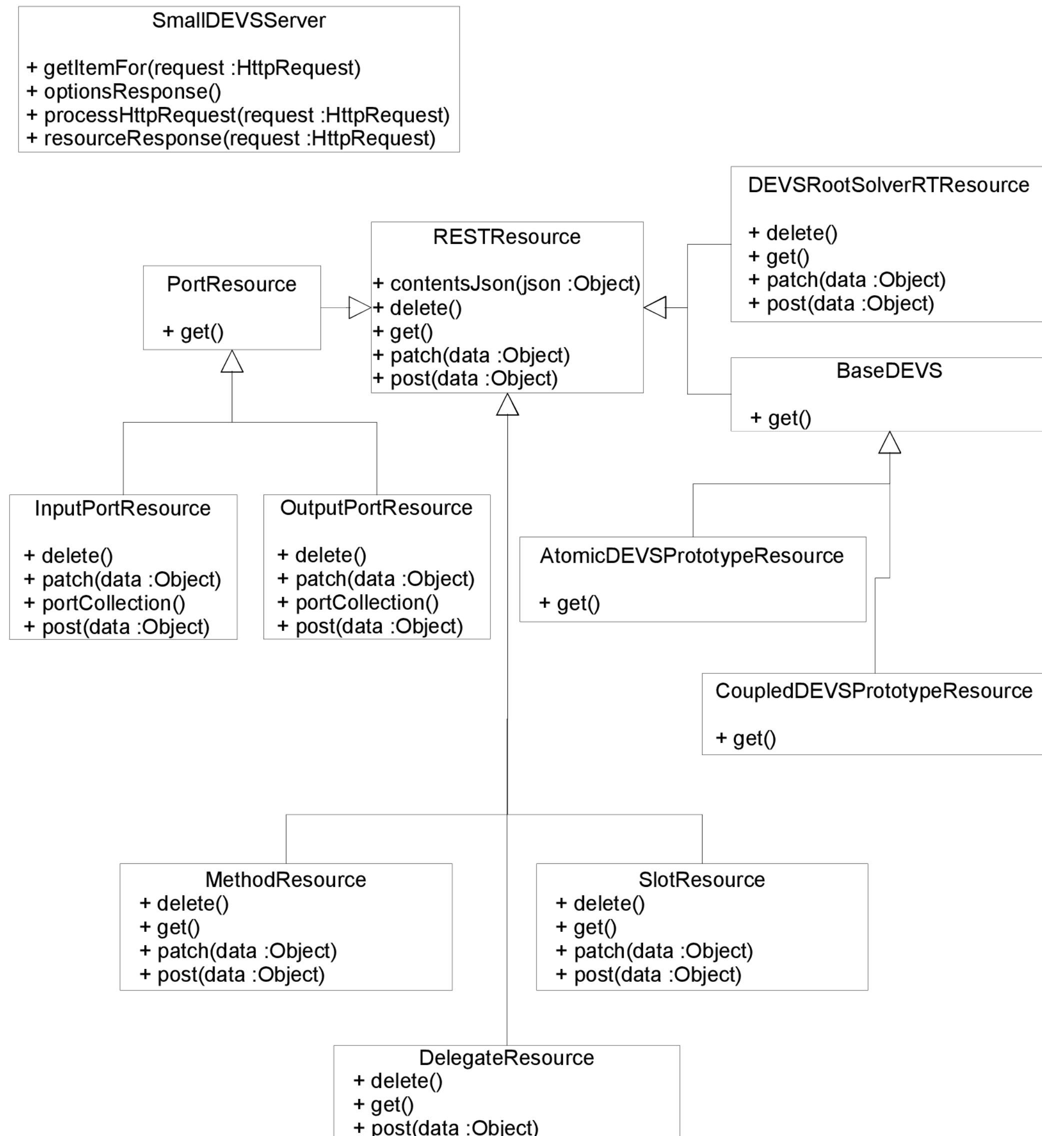
Server je vhodné implementovat jako součást jádra SmallDEVS, protože potřebujeme přístup k datovým objektům jádra. Manipulace s objekty jádra SmallDEVS bude prováděna stejným způsobem jako to dělá grafické uživatelské rozhraní SmallDEVS, ale získaná data budou posílána po síti. Pro komunikaci s klientem bude použito RESTful API navržené v sekci 4.2.2.

Diagram tříd na serveru se nachází na obrázku 4.1. Instance třídy `SmallDEVSServer` reprezentuje jeden spuštěný server. Jelikož bude použita technologie Cross-origin Resource Sharing, je nutné, aby server odpovídal i na požadavky s HTTP metodou OPTIONS. Odpovědi na tyto požadavky vrací metoda `optionsResponse`. Ostatní požadavky zpracovává metoda `resourceResponse`.

Za parsování URL a nalezení správného zdroje je zodpovědná metoda `getItemFor`. Zdroj, který vrátí metoda `getItemFor`, je instancí jednoho z potomků třídy `RESTResource`.

Každý potomek třídy `RESTResource` implementuje aspoň některé z metod `get`, `post`, `patch` a `delete`, které jsou zavolány, pokud se zpracovává HTTP požadavek se stejnojmennou HTTP metodou. Data v HTTP požadavku jsou zpracována a je nastaven stavový kód (např. 200 OK pro úspěšný požadavek HTTP metody GET). Některé metody potřebují do odpovědi uložit také data. Jelikož se odpověď sestavuje ve formě objektů, je nutné objekty přes odesláním serializovat do formátu JSON.

Server je vhodné implementovat tak, aby byl kompatibilní s původním grafickým uživatelským rozhraním (např. je nutné vyřešit ukládání pozic prvků grafu složeného modelu).



Obrázek 4.1: Diagram tříd serveru

Postup implementace serveru lze navrhnout např. takto:

1. počáteční implementace komunikace s klientem pomocí Cross-origin Resource Sharing
2. schopnost deserializovat požadavky ve formátu JSON a odpovědi serializovat opět do formátu JSON
3. vracení statických informací o modelech v `MyRepository`
4. možnost manipulovat s modely v `MyRepository`

4.4 Návrh klientské aplikace

Klientskou aplikaci je možné navrhnout z několika částí. Základem implementace bude model, který bude uchovávat aktuální data po celou dobu běhu aplikace. Diagram tříd modelu se nachází na obrázku 4.2

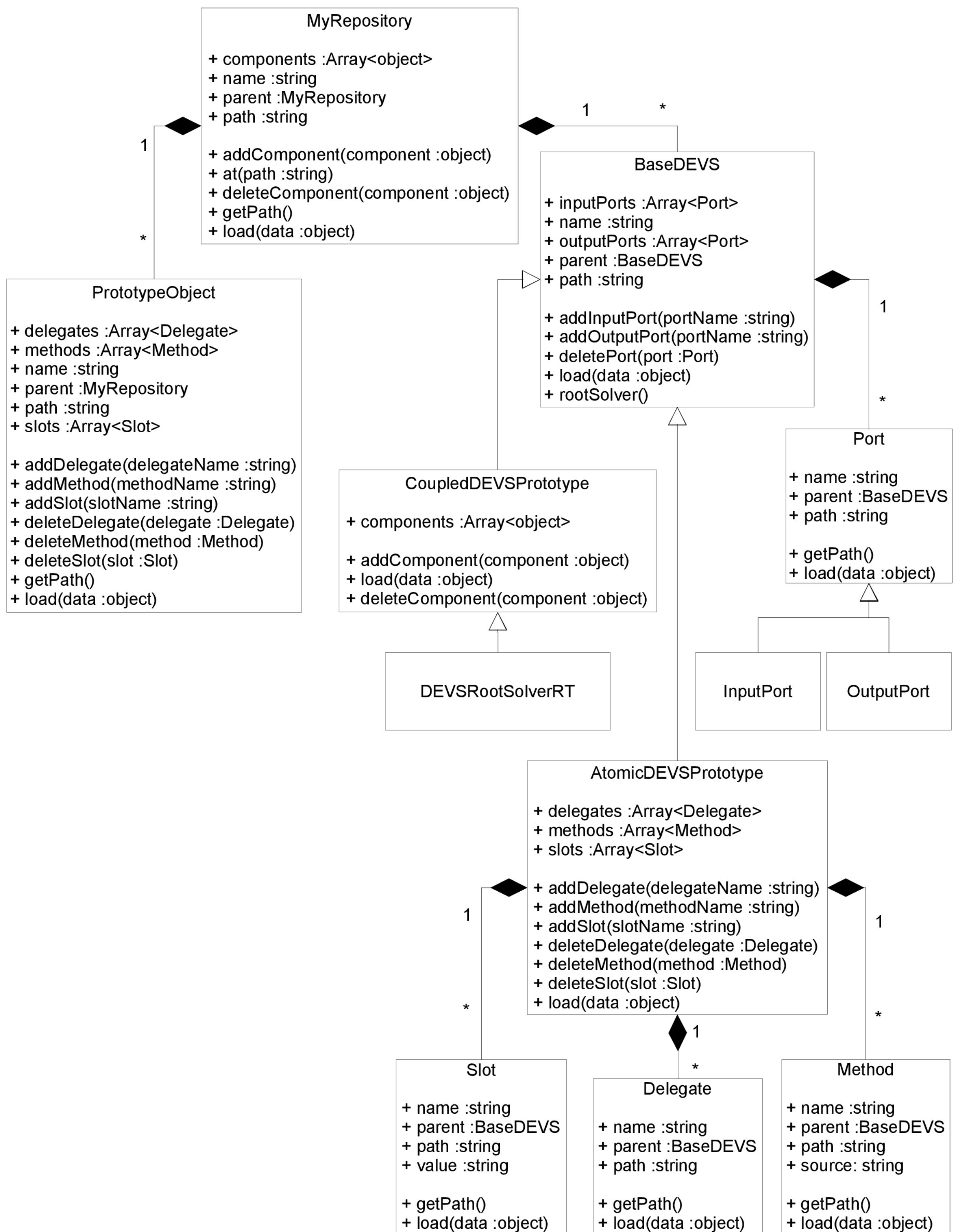
S modelem bude pracovat vrstva pro komunikaci se serverem. Diagram tříd této vrstvy (na obrázku 4.3) je oproti modelu zjednodušený, protože některé informace (např. URL zdroje na serveru) jsou uloženy v modelu.

S modelem a vrstvou pro komunikaci se serverem budou pracovat controllery a views, které starají o správné zobrazení uživatelského rozhraní. Součástí uživatelského rozhraní bude i jednoduchý správce oken, který je vhodné implementovat bez přímé závislosti na modelu. Vzhledem k tomu, že data mohou být zobrazena více částmi uživatelského rozhraní najednou, je nutné se postarat o synchronizaci modelu s uživatelkým rozhraním.

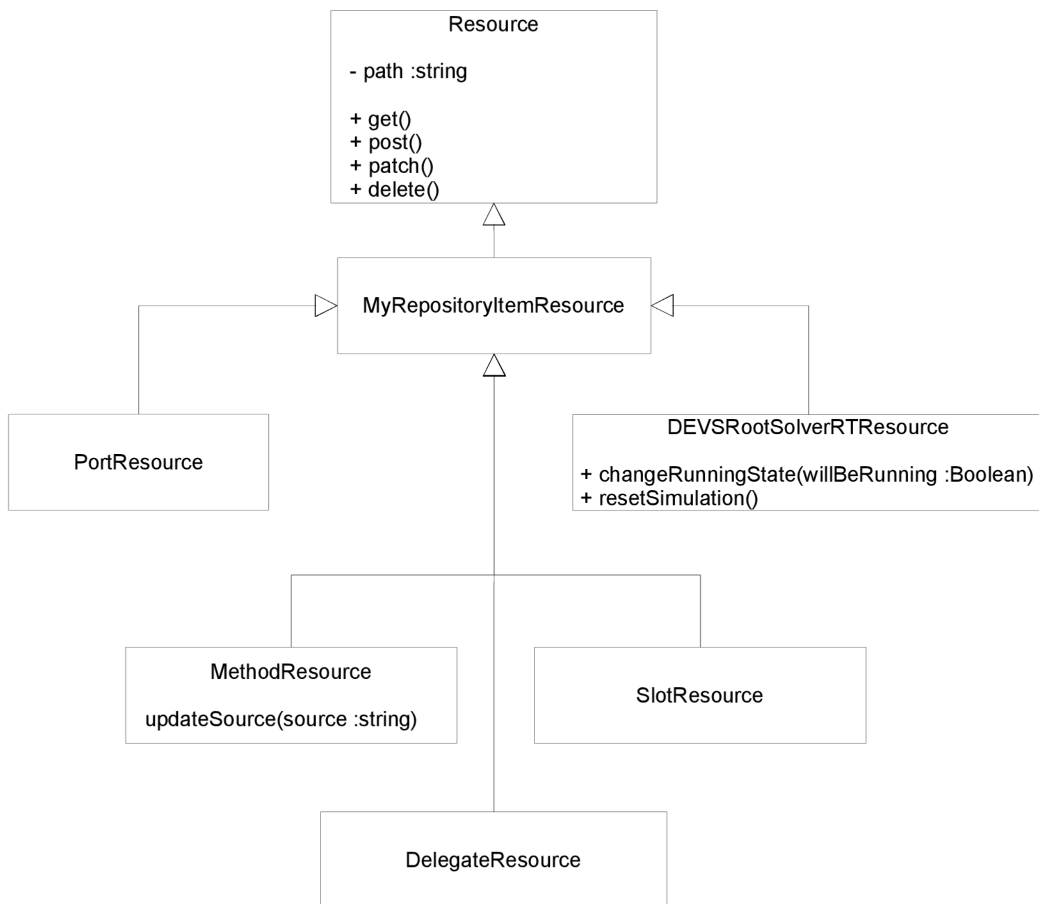
Implementačně nejnáročnější částí rozhraní bude pravděpodobně editační plocha inspektoru složených modelů. Pomocí HTML a CSS je složité reprezentovat propojení portů, které můžou mít i tvar křivky. Je výhodné tyto objekty v inspektoru reprezentovat vektorově. Pro vektorovou reprezentaci je standardem značkovací jazyk SVG.

Implementaci klienta lze rozdělit do těchto etap:

1. vytvoření základní kostry programu, která zahrnuje HTML šablony, základní vzhled a první testy
2. navázání připojení k serveru
3. implementace tříd modelu
4. zobrazení statických informací o modelech ze serveru
5. přidání možnosti manipulovat s modely



Obrázek 4.2: Diagram tříd modelu klienta



Obrázek 4.3: Diagram tříd vrstvy pro komunikaci klienta se serverem

Kapitola 5

Implementace

5.1 Implementace klientské aplikace

Pro implementaci klienta jsem si vybral javascriptový framework AngularJS. Důvodů, proč jsem si vybral právě AngularJS, bylo několik. Výhodou je jistě silná komunita okolo frameworku, kvalitní dokumentace a množství článků o vývoji aplikací v tomto frameworku. Další důvodem byla jednoduchost testování. AngularJS byl navržen tak, aby usnadil testování aplikací. Příklady testů jsou obvykle uvedeny i v dokumentaci k frameworku. Tím se AngularJS liší např. od Backbone.js, který má minimalistickou dokumentaci s popisem funkcí a krátkými příklady. Backbone.js narozdíl od ostatních frameworků uvedených v tabulce 3.3 neobsahuje data binding, pomocí kterého není nutné registrovat události pro překreslení view při změně modelu. AngularJS je dle mého názoru lepší než Knockout, který je příliš minimalistický a neobsahuje např. způsob, jak načítat data ze serveru pomocí technogie AJAX. Ember.js je oproti AngularJS naopak relativně rozsáhlý projekt, který zatím nevydal stabilní verzi (26. 4. 2013 byla k dispozici verze 1.0.0-RC.3).

Jednotlivé části uživatelského rozhraní byly implementovány jako AngularJS direktivy. To umožnilo rozdělit zdrojový kód a šablony HTML do logických celků. Synchronizaci modelu s uživatelským rozhraním není nutné pro aplikace nad AngularJS programovat, o tento proces stará sám framework.

5.1.1 Model

Model je částí klientské aplikace, která se stará o uchování dat z `MyRepository`. Model má stejně jako server podobu stromové struktury. Proto obsahuje kořenový prvek, jehož součástí jsou i další položky.

Model umožňuje uchování aktuálních dat na jednom místě, o synchronizaci mezi modelem a uživatelským rozhraním se stará AngularJS. Např. po přejmenování složeného modelu není nutné explicitně zavolat kód pro překreslení názvu okna s inspektorem složených modelů a položky v prohlížeči repozitáře.

Inicializace modelu probíhá v service `model`, která vrací instanci modelu. Nová instance modelu obsahuje pouze prázdné `App.model.MyRepository`. Tato instance je používána po celou dobu běhu aplikace.

Každá položka modelu má unikátní URL, pomocí které může být pomocí RESTful API stažena ze serveru. Tato URL není součástí dat, která zasílá server, proto je nutné URL generovat na straně serveru. K tomu slouží metoda `getPath`, která rekurzivně prochází stromovou strukturu a vrací URL.

Pro každou položku modelu existuje třída ve jmenném prostoru `App.model`. O vytváření nových instancí tříd modelu se starají metody `load`, které načtou data získaná ze sítě a vytvoří instance tříd také pro všechny potomky ve stromové hierarchii. Metodu `load` je možné zavolat i později, při každé aktualizaci dat tedy není nutné načítat celou stromovou strukturu. Reference na instance tříd z `App.model` jsou později přiřazeny do scope a webový prohlížeč vykreslí aktuální data.

Všechny třídy z jmenného prostoru `App.model` jsou zdokumentovány ve formátu JSDoc. Nástroj JSDoc je schopný zpracovat zdrojový kód s komentáři a vygenerovat dokumentaci ve formátu HTML.

5.1.2 Inspektor složených modelů

Pro zobrazení inspektoru složených modelů je k dispozici direktiva `uiCoupledExplorer`. Umožňuje zobrazit graf se submodely včetně jejich názvů a portů, porty aktuálního složeného modelu a propojení portů. Graf je vykreslen pomocí SVG, které je generováno ze šablony s AngularJS direktivami jako `ng-repeat` a `ng-show`.

Vkládání SVG do HTML dokumentu je podporováno v moderních prohlížečích a Internet Exploreru od verze 10. Vzhledem k předpokládanému použití aplikace v akademické sféře jsem se po domluvě s vedoucím práce rozhodl podporovat pouze aktuální verze moderních webových prohlížečů a Internet Explorer 10 (právě kvůli plné podpoře SVG až v této verzi).

Podařilo se mi dosáhnout velmi podobného zobrazení grafu složeného modelu jako v původním uživatelském rozhraní se zachováním možnosti manipulovat s modelem a graf překreslit.

Graf nebylo možné vykreslit pouze pomocí deklarativního kódu SVG šablony, protože bylo nutné dopočítat rozměry submodelů a segmenty křivek propojení portů.

Šířka submodelu je určena buď šířkou jeho názvu, nebo součtem šířky nejdelších názvů vstupních a výstupních portů a mezery mezi nimi. Pro výslednou šířku se použije větší hodnota. Výška submodelu se odvíjí od počtu portů. Rozměry jsou po výpočtu nastaveny v modelu a do další změny dat se aktualizace rozměrů neprovádí. Pro výpočet rozměrů textu se v service `computeSubmodelSize` používá dočasně vytvořený element typu `text`.

Pro výpočet výsledného tvaru křivky propojení portů je implementována pomocná service `computeCouplingSegments`, která obsahuje algoritmus, který převede reprezentaci tvaru křivky propojení ze serveru na křivku SVG elementu `path`. Na serveru se tvary křivek propojení portů ukládají jako souřadnice bodů, kterými toto propojení na grafu prochází. Pomocí SVG elementu `path` je možné vykreslovat Beziérovy křivky. Beziérovy křivky jsou definovány pomocí řídících bodů, kde první a poslední bod jsou krajní body křivky a ostatní řídící body definují tvar křivky. Algoritmus, který provádí převod mezi zmíněnými dvěma reprezentacemi, byl insprirován algoritmem z knihovny Connectors, kterou používá i původní uživatelské rozhraní SmallDEVS.

5.1.3 Inspektor atomických modelů

O manipulaci s atomickými modely se stará direktiva `uiAtomicExplorer`. Uživatelské rozhraní je rozděleno na části, které dovolují manipulovat s porty, sloty, delegáty, metodami a komentářem, který patří prohlíženému atomickému modelu.

Implementace inspektoru atomických modelů je rozdělena do dalších direktiv obsahujících funkcionalitu, která přímo nesouvisí s inspektorem. Direktiva `uiTreeItem` zapouzdřuje stromové zobrazení inspektoru. Část rozhraní, která obsahuje tuto direktivu lze dle potřeby rozbalit či sbalit.

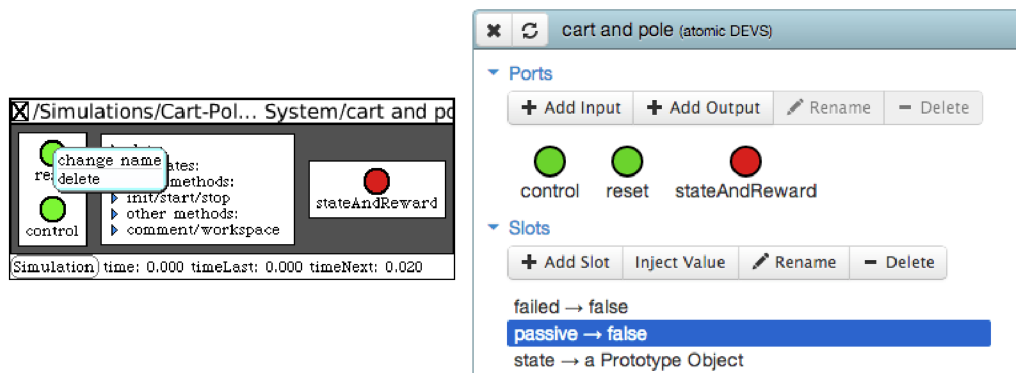
Pro editor metod je použit open source projekt Codemirror a direktiva pro AngularJS z dalšího open source projektu AngularUI. Codemirror se stará o zvýrazňování syntaxe a poskytuje základní možnosti editace zdrojového kódu. Editor je nastaven tak, aby zvýrazňoval syntaxi jazyka Smalltalk.

5.1.4 Uživatelské rozhraní klientské aplikace

Při návrhu uživatelského rozhraní klientské webové aplikace jsem se snažil napodobit původní uživatelské rozhraní SmallDEVS a upravit jej pro webovou aplikaci, pokud to bylo nutné. Ovládání aplikace běžící ve webovém prohlížeči se mírně liší od ovládání aplikace desktopové. Hlavním rozdílem, na který jsem při implementaci narazil, byla kontextová menu.

Kontextová menu se ve webových aplikacích zpravidla nepoužívají, protože nahrazení kontextového menu webového prohlížeče není podporováno napříč všemi prohlížeči na různých zařízeních. Zejména na mobilních zařízeních je tento problém velmi znatelný.

Absenci kontextových menu lze vyřešit přesunutím položek menu přímo do uživatelského rozhraní. Pokud je v daném kontextu část uživatelského rozhraní nepoužitelná, je možné ji dočasně učinit neaktivní. Příkladem takového uživatelského rozhraní je seznam portů a slotů v inspektoru složených modelů (viz obrázek 5.1), který má akce pro přejmenování a smazání neaktivní do doby, než je vybrán port resp. slot.



Obrázek 5.1: Porovnání uživatelských rozhraní inspektoru atomických modelů

Pro základní vzhled tlačítek a jiných komponent jsem použil front-end framework Bootstrap. Vlastní komponenty jsou stylovány pomocí preprocesoru SASS [SAS13].

5.1.5 Generování produkčního kódu

Klientská aplikace obsahuje konfigurační soubor pro nástroj Grunt. Grunt, který byl zmíněn v kapitole 3.5.3, byl nastaven tak, aby spojil všechny zdrojové soubory v jazyce JavaScript do jediného soubor. Výsledný soubor je ještě minifikován nástrojem UglifyJS. Kromě generování javascriptového kódu se Grunt používá také pro spojení a minifikaci CSS souborů.

Vygenerovaný produkční kód má více než 2x menší velikost než původní kód. Tato produkční verze klientské aplikace lze spustit i otevřením souboru `index.html`.

5.2 Implementace serveru

Server byl implementován podle návrhu na obrázku 4.1. Pro základní implementaci serveru jsem použil knihovnu KomHttpServer a KomServices. Deserializaci a serializaci dat ve formátu JSON provádí knihovna JSON.

Oproti původnímu návrhu jsou implementovány navíc metody `getBasic`, které vrací data ve formátu pro prvotní načtení. Při prvním načtení klient načítá celý obsah `MyRepository`.

Kapitola 6

Testování aplikace a vyhodnocení výsledků

6.1 Testování klientské aplikace pomocí automatických testů

6.1.1 Unit testing

Unit testing (lze přeložit jako jednotkové testování, ale ustálený pojem neexistuje) je způsob automatického testování, které doporučuje testování jednotlivých částí (jednotek) programu v izolaci.

Pokud použijeme objektově orientované programování, je jednotkou zpravidla třída a unit test obsahuje ověření funkčnosti metod třídy. Spolu s unit testingem je při psaní zdrojového kódu možné použít techniku Dependency Injection (česky vkládání závislostí), která odstraňuje přímé závislosti tříd. Tyto závislosti jsou třídě předány, např. pomocí argumentů konstruktoru.

Použití unit testingu obvykle vede k odděleným třídám a modulům, které jsou vhodné pro opětovné použití v dalších projektech. Je také možné ve větší míře využít refaktorování stávájícího zdrojového kódu, protože riziko porušení funkcionality je nižší než při absenci testů. Další výhodou unit testingu je využití kódu testů pro dokumentační účely, jelikož test obsahuje, k jakému účelu daná jednotka slouží a jakým způsobem se má správně používat.

6.1.2 Test Driven Development

Test Driven Development (zkráceně TDD) je technika vývoje software, kterou popularizoval Beck ve své knize Test Driven Development: By Example [Bec02]. Techniku TDD je možné využít pro tvorbu kvalitnějšího zdrojového kódu s automatickými testy. Typický postup vývoje vypadá následovně:

1. Je napsán automatický test.
2. Test je spuštěn a je ověřeno, zda selže.
3. Je doplněna funkcionalita, která má být otestována. Všechny testy musí být v této fázi úspěšné.
4. Zdrojový kód je refaktorován, dokud není efektivní a elegantní. Pomocí testů je ověřeno, zda zavedené změny nezpůsobily žádné chyby.
5. Pokračuje se opět krokem 1 pro jinou část funkcionality aplikace.

6.1.3 TDD v jazyce JavaScript

Programovací jazyk JavaScript nemá zabudované řešení pro automatické testování. Automatická testování v JavaScript má svá specifika, kterými se zabývá kniha Test-Driven JavaScript Development [Joh10]. Existuje několik knihoven, které automatické testování v JavaScriptu usnadňují. Autoři AngularJS preferují Jasmine. Tuto knihovnu doporučují rovněž pro testování aplikací postavených na frameworku AngularJS.

Testy napsané pomocí Jasmine jsou většinou strukturovány do souborů, ve kterých jsou popsány jednotky. Struktura souborů s testy zpravidla kopíruje strukturu souborů se zdrojovým kódem aplikace. Každá jednotka je popsána pomocí funkce `describe`, ve které jsou jednotlivé testy součástí funkcí `it`. Funkce `describe` je možné do sebe vnořovat. To je výhodné, pokud testujeme část jednotky, která potřebuje vlastní inicializační kód.

Jednotky zpravidla vyžadují určité množství inicializačního kódu. Pro odstranění duplicity inicializačního kódu slouží funkce `beforeEach`. Funkce `beforeEach` je spuštěna před každým testem, který je součástí aktuální jednotky. Obdobně existuje funkce `afterEach`, která je spuštěna po vykonání každého testu aktuální jednotky.

6.1.4 Spouštění testů v JavaScriptu

Po spuštění testů je možné v JavaScriptu použít obyčejný HTML soubor, do kterého vložíme jak zdrojový kód aplikace, tak specifikaci testů. Pro usnadnění práce je výhodné použít program Karma, který spustí testy automaticky ihned po uložení souboru se zdrojovým kódem.

Karma umožňuje testy spouštět jak v headless prohlížeči PhantomJS, tak i v klasických prohlížečích jako Google Chrome či Mozilla Firefox, což usnadňuje testování rozdílů mezi prohlížeči. Karma rovněž integruje nástroj istanbul pro generování code coverage (viz níže).

6.1.5 Pokrytí kódu

Pokrytí kódu (anglicky code coverage) je metrika, která se používá při testování software. Popisuje, pro jak velkou část kódu existují testy.

Nástroj instanbul umožňuje generovat pokrytí kódu pro zdrojový kód v programovacím jazyce JavaScript. Instanbul upraví reprezentaci zdrojového kódu v abstraktním syntaktickém stromu tak, aby v každé části zdrojového kódu bylo zaznamenáno, zda tato část byla během testů spuštěna. Ze zaznamenaných dat generuje instanbul přehledný výpis, který obsahuje procentuální pokrytí zdrojového kódu pro každý soubor v rámci projektu a průměrné pokrytí zdrojového kódu pro adresář nebo celý projekt.

6.2 Testování funkčnosti

Tato kapitola slouží jako popis nově implementované klientské aplikace. Funkčnost je předvedena na tvorbě jednoduchého modelu. Jako model testovací model je použit model *Generator and Processor* ze stabilní verze SmallDEVS.

Postup pro tvorbu identického modelu je následující:

1. Nejprve je nutné vytvořit novou simulaci. Toho docílíme kliknutím na tlačítko *Add Simulation* v prohlížeči repozitáře. Nová položka by se měla objevit ve složce *Simulations.*

2. Po vytvoření simulace je nutné vytvořit submodely. Poklepáním otevřeme nově vytvořený složený model a kliknutím na tlačítko *Add Atomic DEVS* vytvoříme submodely *generator* a *processor*. Submodely by se měly opět objevit v prohlížeči repozitáře.

3. Poklepáním na položku v prohlížeči repozitáře otevřeme atomický model *generator*. Nyní vytvoříme pomocí tlačítka *Add Output* výstupní port *out*. Stejný postup opakujeme i pro vstupní a výstupní porty atomického modelu *processor*.

4. V inspektoru složeného modelu propojíme porty.

5. Nyní vytvoříme všechny sloty pomocí tlačítka *Add Slot*. Pro přepnutí zpět do inspektoru atomického modelu *generator* můžeme použít horní lištu se seznamem oken. Pro správnou funkčnost je nutné vložit do slotů správné hodnoty. Toho docílíme označením slotu a klepnutím na tlačítko *Inject Value*. Aktuální verze klientské aplikace zatím bohužel nepodporuje tvorbu a úpravu prototypových objektů ve slotech, tak je nutné objekt *aJob* vytvořit pomocí původního grafického uživatelského rozhraní.

6. Nakonec zbývá přidat kód metod modelu. Pomocí tlačítku *Add Method* vždy přidáme metodu, do označeného editoru zapíšeme zdrojový kód a vše potvrdíme tlačítkem *Accept*.

V tomto okamžiku by vytvořená simulace měla být funkční. Funkčnost lze ověřit spuštěním simulace pomocí tlačítka *Start* v dolní části okna s modelem. Stav simulace lze pozorovat např. sledováním hodnot slotů. Hodnoty lze aktualizovat tlačítkem v levé horní části okna s modelem.

Kapitola 7

Závěr

V rámci bakalářské práce se mi podařilo navrhnout a implementovat webovou aplikaci pro přístup k jádru nástroje SmallDEVS. Webová aplikace umožňuje manipulovat s modely podobně jako původní grafické uživatelské rozhraní. Pro zpřístupnění služeb jádra SmallDEVS jsem navrhl server, který s webovou aplikací komunikuje pomocí RESTful protokolu.

Realizovaná aplikace umožňuje nasadit SmallDEVS v prostředí, kde lze počítač ovládat pouze vzdáleně. Navržený komunikační protokol dále umožňuje vytvářit alternativní klienty pro různé účely.

Zdrojové kódy serveru i webové aplikace byly zveřejněny jako open source software[1]. Použitý software (Squeak, AngularJS, Codemirror) je dostupný pod licencí kompatibilní s MIT licencí, proto jsem svůj zdrojový kód zveřejnil také s MIT licencí.

Součástí implementace klientské aplikace jsou i automatické testy. Pokrytí kódu je asi 97%. Automatické testy se prokázaly jako velmi užitečné při refaktoringu, ale i při vývoji nové funkcionality. Testy lze využít také jako formu dokumentace. Další dokumentace k modelu klientské aplikace je dostupná ve formě výstupu nástroje JSDoc.

7.1 Možnosti rozšíření

7.1.1 Podpora novějších implementací Smalltalku

Poslední stabilní verze SmallDEVS byla vydána pro implementaci Smalltalku 4.3. Byla rovněž vydána experimentální verze pro Pharo, další open source implementaci Smalltalku. Tato však není plně funkční, protože všechny knihovny, na kterých SmallDEVS závisí nejsou k dispozici pro Pharo. Tyto závislosti SmallDEVS jsou však nutné pouze pro práci s původním grafickým uživatelským rozhraním.

Jelikož server, který byl v rámci této práce implementován, nezávisí na původním grafickém uživatelském rozhraní, je možné vydat verzi SmallDEVS pro novější implementace Smalltalku, která bude zahrnovat server a klient implementovaný v této práci.

7.1.2 Rozšíření inspektoru atomických modelů o editaci Petriho sítí

Při použití Petriho sítí pro popis atomických modelů by bylo možné použít upravený inspektor složených modelů. Petriho sítě je vhodné zobrazit jako vektorovou grafiku podobně jako graf submodelů v inspektoru složených modelů.

[1] Projekt je zveřejněn na adrese https://github.com/PavelGavlik/OpenDEVSClient

Pro přenos reprezentace Petriho sítě by bylo možné použít rovněž RESTful protokol, který by obsahoval zdroje pro jednotlivá místa a přechody. S místy a přechody by bylo možné manipulovat podobně jako např. s porty a sloty v aktuální verzi protokolu.

7.1.3 Bezpečnostní prvky

SmallDEVS umožňuje při modelování využít plnou sílu programovacího jazyka Smalltalk. To je výhodné při modelování, ale možnost vyhodnocení libovolného výrazu a schopnost vzdáleně upravovat modely může potencionálnímu útočníkovi umožnit získat přístup k počítači, na kterém je spuštěn SmallDEVS. Proto je vhodné před použitím projektu mimo lokální síť učinit několik změn.

Aby se předešlo přístupu neoprávněných osob, je nutné zavést autentizaci (např. pomocí uživatelského jména a hesla).

Některé modelované systémy mohou obsahovat citlivá data. Tato data se při použití vzdáleného rozhraní ke SmallDEVS přenášejí po síti. Aby se předešlo získání dat cizí osobou, je vhodné data šifrovat. Šifrování dat by mohlo být realizováno např. pomocí protokolu SSL.

Literatura

[Ang12] Angularjs [online]. `http://www.angularjs.org/`, 2012 [cit. 14. 12. 2012].

[BDNP09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, and Damien Pollet. *Squeak by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-0-2.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002. ISBN 978-0321146533.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. ISBN 0-599-87118-0.

[git13] Github [online]. `http://github.com`, 2013 [cit. 21. 4. 2013].

[gru13] Grunt [online]. `http://gruntjs.com/`, 2013 [cit. 11. 5. 2013].

[GS13] Brad Green and Shyam Seshadri. *AngularJS*. O'Reilly Media, 2013. ISBN 978-1-449-34485-6.

[Jan11] Vladimír Janoušek. *Simulace a návrh vyvíjejících se systémů*. Faculty of Information Technology BUT, Brno, 2011. ISBN 978-80-214-4414-0.

[Jan12] Vladimír Janoušek. Smalldevs [online]. `http://www.fit.vutbr.cz/~janousek/smalldevs/`, 2012 [cit. 26. 11. 2012].

[Joh10] Christian Johansen. *Test-Driven JavaScript Development (Developer's Library)*. Addison-Wesley Professional, 2010. ISBN 978-0321683915.

[RR07] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 2007. ISBN 978-0-596-52926-0.

[SAS13] Sass [online]. `http://sass-lang.com/`, 2013 [cit. 10. 5. 2013].

[SO13] Stack overflow [online]. `http://stackoverflow.com`, 2013 [cit. 21. 4. 2013].

[Tod13] Todomvc [online]. `http://todomvc.com/`, 2013 [cit. 14. 4. 2013].

[ugl13] Uglifyjs [online]. `https://github.com/mishoo/UglifyJS/`, 2013 [cit. 11. 5. 2013].

[ZPK00] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation, Second Edition*. Academic Press Inc., 2000. ISBN 0127784551.

Příloha A

Obsah CD

Přiložené CD obsahuje tyto adresáře:

- client/app/ – zdrojové kódy klientské aplikace
- client/dist/ – produkční verze klientské aplikace
- client/test/spec/ – specifikace testů
- client/test/coverage/ – code coverage
- doc/ – vygenerovaná dokumentace pro model klientské aplikace
- server/ – zdrojové kódu serveru pro SmallDEVS
- tex/ – zdrojové kódy textové části
- tex/projekt.pdf – výsledná PDF verze textové části