Grafický editor schémat sítě pro informační systém

Graphic editor of network layouts for information system

Bc. Tomáš Kozel

Diplomová práce
2010

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2009/2010

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš KOZEL**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Grafický editor schémat sítě pro informační systém**

Zásady pro vypracování:

1. **Analyzujte informační systém FreenetIS, který vznikl v rámci diplomových prací [1] a [2]**
2. **Implementujte grafický editor schematu sítě, který umožní uživatelský přívětivou grafickou tvorbu dokumentace zapojení rozlehlé počítačové sítě přímo v prostředí informačního systému tak, aby veškeré změny schematu byly ukládány přímo do databáze.**
3. **Při implementaci využijte komponentu typu jgraph nebo mxgraph.**
4. **Výsledný systém zveřejněte pod open source licencí GPL2**

Rozsah práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **DANĚK, Petr. Informační systém pro správu a evidenci uživatelů rozsáhlých sítí. [s.l.], 2008. 71 s. UTB/UAI. Diplomová práce.**
2. **ROZEHNAL, Marek. Informační systém pro správu a evidenci uživatelů rozsáhlých sítí. [s.l.], 2008. 72 s. UTB/UAI. Diplomová práce.**
3. **JGraph and mxGraph [online]. 2001-2009 [cit. 2010-02-05]. Dostupný z WWW: http://www.jgraph.com/.**
4. **Kohana: Swift, Secure, and Small PHP 5 Framework [online]. 2007-2010 [cit. 2010-02-05]. Dostupný z WWW: http://www.kohanaphp.com/.**
5. **HORSTMANN, Cay S., CORNELL, Gary. Core Java™ 2: Volume I--Fundamentals. [s.l.] : Prentice Hall, 2000. 832 s. ISBN 0-13-089468-0.**
6. **FLANAGAN, David. JavaScript: The Definitive Guide, 5th Edition. [s.l.] : O Reilly, 2006. 1018 s. ISBN 0-596-10199-6.**
7. **ANDI STIG, Gutmans, STHER, Bakken, DERICK, Rethans. PHP 5 Power Programming. [s.l.] : PRENTICE HALL, 2005. 501 s. ISBN 0-131-47149-X.**
8. **FOWLER, Martin . Patterns of Enterprise Application Architecture. [s.l.] : Addison Wesley, 2002. 560 s. ISBN 0-321-12742-0.**

Vedoucí diplomové práce: **Ing. Tomáš Dulík**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **19. února 2010**

Termín odevzdání diplomové práce: **8. června 2010**

Ve Zlíně dne 19. února 2010

prof. Ing. Vladimír Vašek, CSc.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

ABSTRAKT

Základním cílem práce je tvorba klienta pro komunikaci s webovou aplikací FreeNetIS. V první části se analyzují možnosti komunikace desktopové aplikace s webovou. Dále se pokračuje analýzou nástroje JGraphX, který je využit pro implementaci aplikace. V poslední teoretické části jsou krátce popsány technologie, které jsou použity při vývoji. V praktické části se zabýváme rozšířením serverové části o REST rozhraní, pomocí kterého klient komunikuje s webovou aplikací. Dále je zde popsáno rozšíření původního databázového modelu. V poslední části je popsána vlastní tvorba klienta v Javě a jsou prezentovány výsledky.

Klíčová slova: HTTP, Webové služby, REST, FreeNetIS, klient FreeNetIS, JGraphX, Java, Swing

ABSTRACT

This diploma thesis describes creation of a client for communication to web application FreeNetIS. In the first part analyzes the possibility of communication desktop with web application. In the next part continue analysis of tools JGraphX, which is used to implementation of application. In the last theoretical part are briefly describes technologies, which are use in the development. In the practical part this paper solves upgrade server section of the REST interface, through client communicate with web application. In the next part is described upgrade the initial database model. In the last part is solved own creation of a client in the Java and the results are presented.

Keywords: HTTP, Web services, REST, FreenetIS, klient FreeNetIS, JgraphX, Java, Swing

Tímto bych chtěl poděkovat Ing. Tomáši Dulíkovi za ochotu, cenné rady, připomínky a vedení při tvorbě práce.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně 8.6.2010 Podpis diplomanta:

OBSAH

ÚVOD

Jelikož člověk je tvor společenský, již od počátku věku se chtěl sdružovat do komunit. Jendou z takových komunit představují i metropolitní sítě. Je to skupina lidí, která staví rozsáhlou síť na vlastní náklady sama pro sebe z toho důvodu, aby následně mohli společně všichni členové sdílet různé služby (např. internetové připojení). Záštitu nad celou touto sítí většinou tvoří nezisková organizace, která samozřejmě musí mít nějakým způsobem zmapováno, kdo všechno je do sítě připojen, jaké je schéma sítě, kdo je za konkrétní zařízení zodpovědný a v neposlední řadě i ekonomiku celého sdružení. Vidíme, že je mnoho věcí, které musí sledovat. Proto by nebylo od věci mít software, který celou agendu zastřešuje.

FeeNetIS je informační systém šířený pod licencí GNU/GPL a primárně určený pro neziskové organizace poskytující připojení k síti internet[1]. Systém obsahuje většinu informací, které pro sebe potřebuje taková organizace sledovat. FreeNetIS je celkem intuitivní, ale některé informace se v něm neupravují příliš pohodlně. Velmi špatně se v systému pracuje se schématem sítě. V současnosti je to několik HTML tabulek. Jejich procházením si lze jen těžko v hlavě člověka zmapovat nějakou topologii sítě. Proto je žádoucí vytvořit rozšíření, které by tento neduh napravilo.

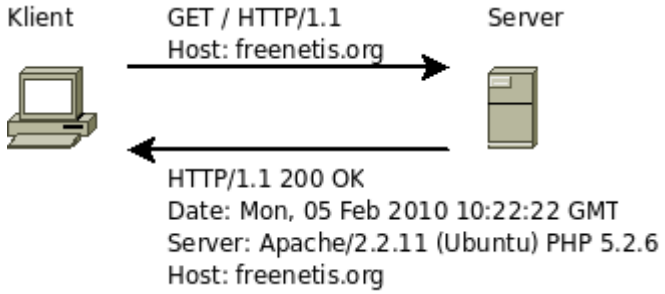
Na konci předchozího odstavce byl nastíněn cíl této diplomové práce, kterým je vytvoření rozšíření pro FreeNetIS, jenž by umožňovalo jednoduchou úpravu fyzického modelu sítě. Nejdříve se zabýváme analýzou původního řešení. Dále jsou nastíněny problémy architektury aplikace, případně technologie, které použijeme na jejich řešení. Po té rozebíráme implementační detaily a na závěr představujeme výsledky práce.

I. TEORETICKÁ ČÁST

1 KOMUNIKACE KLIENT / SERVER VE WEBOVÉM PROSTŘEDÍ

1.1 Protokol HTTP

Celý World Wide Web je založen na protokolu HTTP, což je jednoduchý bezstavový požadavek/odpověď protokol aplikační vrstvy. Protokol vychází z architektury klient/server. Klient (nejčastěji webový prohlížeč) posílá požadavek na zdroj, který by se měl nacházet na serveru. Server vytvoří odpověď a pošle ji na klienta dle toho, jestli daný zdroj našel nebo nikoliv. Přesný formát požadavků a odpovědí je obsažen v specifikaci HTTP protokolu. Nyní jsou využívány dvě verze a to 1.0 a 1.1.



Obr. 1. Ukázka požadavku a odpovědi v protokolu HTTP

Na obrázku Obr. 1 je vidět příklad základní http komunikace. Všímáme si, že požadavek je uvozen řetězcem GET, což je metoda, která udává typ požadavku. Jednoduše říká webserveru, co od něj klient očekává. Metod je několik typů (jejich počet se liší dle verze protokolu), jejich popis je uveden v tabulce Tab. 1.

Tab. 1. Tabulka metod HTTP

Metoda	Popis	Http 1.0	Http 1.1
GET	Požadavek na zdroj (html, obrázek, ...) na serveru	X	X
POST	Požadavek na zdroj na serveru s posláním dat	X	X
HEAD	Požadavek na hlavičky daného zdroje	X	X
DELETE	Smazání zdroje z webserveru		X
PUT	Uložení zdroje na webserver		X
OPTIONS	Vrací context, který sever podporuje (metody, typ obsahu, atd.).		X

TRACE	Slouží pro sledování cesty celého požadavku		X
CONNECT	Slouží k tunelování HTTP		X

Požadavek má následující formát ([] udávají volitelnost, /n nový řádek) :

```
METODA URL VERZE_HTTP/n
[1..n HLAVICEK/n]
/n
[DATA]
```

Hlavičky nám přesněji specifikují požadavek. V HTTP 1.0 byly všechny hlavičky nepovinné. Naproti tomu v HTTP 1.1 se stala hlavička Host povinná a umožnila na webserverech používat techniku virtuálních hostů. Data jsou volitelná a samozřejmě jejich přijmutí závisí na metodě. Některé důležité hlavičky, které se používají dále v práci, jsou uvedeny v tabulce Tab. 2.

Tab. 2. Hlavičky požadavku

Jméno hlavičky	**Popis**
Accept	Udává, jaké MIME (typ média) je schopen klient reprodukovat.
Host	Obsahuje jméno serveru
Content-Type	Typ zasílaných dat serveru
Content-Length	Délka zasílaných dat

Odpověď má následující formát:

```
VERZE_HTTP STAVOVY_KOD [HLASENI]/n
[1..n HLAVICEK/n]
/n
[DATA]
```

Na formátu se moc náležitostí nemění, pouze uvozující řádek. Po verzi protokolu se vrací stavový kód, který nám udává úspěšnost odpovědi (viz Tab. 3). Po verzi je umístěna libovolná hláška, která není povinná. Znovu popíšeme několik hlaviček, které se využívají v odpovědi.

Tab. 3. Přehled kódů HTTP odpovědi

Skupina kódů http odpovědi	Popis
20x	Vše proběhlo v pořádku.
30x	Zdroj již není na webserveru, klient je přesměrován na jiný.
40x	Špatný, neautorizovaný dotaz nebo neexistující zdroj
50x	Došlo k interní chybě serveru.

Tab. 4. Přehled často používaných hlaviček HTTP odpovědi

Jméno hlavičky	Popis
Content-Type	Typ zasílaných dat, který si server vybral. Vůbec není závislý na podporovaných typech v požadavku.
Expires	Server udává, jestli je daný zdroj ještě aktuální.
Location	Zdroj už se může nacházet na jiné adrese, tzn., server vrátí kód přesměrování a v Location adresu kam dál pokračovat.
Content-Length	Délka zasílaných dat

1.2 Webové služby

Je dáno několik požadavků:

- komunikace dvou různých aplikací (klient – server)
- obě aplikace nemusí být na stejném stroji
- serverová aplikace je webová

Pro splnění těchto požadavků jsou nejvhodnějším řešením webové služby. Webové služby jsou založeny na architektuře SOA a v prostředí internetu jsou to nadstavby nad protokolem

HTTP. Výhodou všech webových služeb je jejich platformní nezávislost. Například SOAP klient vytvoření v Perlu může bez problémů komunikovat se SOAP serverem, který běží na dotNet. Jelikož dále v práci bylo nutné jeden z přístupů k webovým službám vybrat, je zde popis několika nejčastěji využívaných.

1.2.1 Pure HTTP

Serverová aplikace vůbec nepotřebuje vědět, že k ní je připojen nějaký "speciální" klient. Chová se stejně, jakoby byl web procházen uživatelem pomocí webového prohlížeče. Jedním z příkladů takového přístupu je Mylyn. Jedná se o zásuvný modul do vývojové platformy Eclipse, který umožňuje správu chyb v nejrozšířenějším bugreportovacím systému Bugzilla. Je to docela triviální řešení, má ovšem docela dost nevýhod. Klient posílá na server požadavky stejně jak webový prohlížeč a po přijetí odpovědi daný výsledek parsuje. Zpracovávat odpověď je možné například xml parserem, pokud je výsledek v xhtml. Ve všech ostatních případech přijdou na řadu regulární výrazy a dolování všech potřebných informací z přijaté webové stránky. Z toho plynou i nevýhody.

Výhody

- Není třeba dělat žádné změny v serverové části aplikace
- Jednoduchost

Nevýhody

- Klient je závislý na vzhledu webové aplikace
- Klient je závislý na url stromu webové aplikace
- Řešení chybových stavů

1.2.2 XML/RPC

XML/RPC je založeno, jak už název napovídá, na RPC(Remote Process Calling). To je v základu volání rutiny jiného programu, který je v jiném adresním prostoru než program volající. Přeneseno do webového prostředí jde o poslání HTTP požadavku, kde tělo obsahuje jméno metody a její parametry. Metoda se zavolá na serveru a vrací odpověď ve stejné reprezentaci jako příchozí požadavek. Již v názvu vidíme, že základní reprezentace těla požadavku je XML. Nabízí se nám i další možnosti jako např. JSON.

Následuje ukázka požadavku [5]:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>circleArea</methodName>
  <params>
    <param>
      <value><double>2.41</double></value>
    </param>
  </params>
</methodCall>
```

Ukázka odpovědí [5]:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>18.24668429131</double></value>
    </param>
  </params>
</methodResponse>
```

Z odpovědí je jasně viditelné, že každá návratová hodnota má daný typ i celý požadavek a odpověď má jasně danou strukturu. Všechny požadavky se posílají pomocí HTTP metody POST.

Výhody

- Implementace ve všech používanějších jazycích
- Jednoduchý a jasný díky své dané struktuře (výhoda vs. nevýhoda)
- Standardizované řešení chybových stavů

Nevýhody

- Daná struktura
- Omezený počet datových typů
- Nelze přenášet objekty

1.2.3 SOAP (Simple Object Access Protocol)

SOAP je dalším z protokolů, který umožňuje standardizovanou komunikaci klienta se serverem, založeném na XML. Jsou zde ovšem ještě navíc využity jmenné prostory. Požadavek i odpověď má následující tvar:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

První jmenný prostor, který je označený soap, uvozuje tagy, které slouží k definici struktury celé zprávy. Vlastní význam základních tagů určujících strukturu je následující:

soap:envelope – je obálka celého požadavku, která musí být vždy obsažena a musí definovat dva dříve zmíněné jmenné prostory.

soap:header – nepovinný tag, který umožňuje modulárně rozšiřovat požadavky. Například je zde možné implementovat správu transakcí nebo autentifikaci.

soap:body – zde je vlastní struktura zprávy.

soap:fault – je část, kde jsou vraceny chyby, které se vyskytly při vyřizování požadavku

Požadavek může vypadat následovně [5]:

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/' xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
```

```
  <soap:Body>
    <n:sayHello xmlns:n='urn:examples:helloservice'>
      <firstName xsi:type='xsd:string'>World</firstName>
    </n:sayHello>
  </soap:Body>
</soap:Envelope>
```

Vidíme, že je volána metoda sayHello s parametrem World. Parametr tvoří řetězec, který je definovaný ve schématu XMLSchema. Toto schéma obsahuje velké množství definic různých typů, které by se mohly hodit při návrhu služby. Důležitou výhodou těchto typů je tak jednoznačné sjednocení mezi platformami v rámci SOAP komunikace. Samozřejmě návrháři služby nic nebrání zavést si vlastní typy, buď přímo v těle požadavku, nebo v připojeném schématu.

Následuje ukázka odpovědi [5]:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'>
  <SOAP-ENV:Body>
    <ns1:sayHelloResponse
      xmlns:ns1='urn:examples:helloservice'
      SOAP-ENV:encodingStyle=
        'http://schemas.xmlsoap.org/soap/encoding/'>
      <return xsi:type='xsd:string'>Hello, World!</return>
    </ns1:sayHelloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

V odpovědi od serveru je vidět, že nám vrací řetězec „Hello, World!“. Sledujeme celkové zvýšení složitosti oproti XML/RPC za cenu větší rozšiřitelnosti.

1.2.3.1 WSDL

Pokud chceme používat nějakou SOAP službu, je potřebné zjistit, co a jak je možné volat. K tomu slouží právě WSDL, které popisuje syntaxi volání metod dané služby. Generovat

WSDL není povinné, protože služba ho nevyžaduje ke svému běhu. Je to ale velmi cenná věc pro vývojáře, kteří se službou pracují. Její formát zde nebudeme popisovat, protože je poměrně složitý a z 99% případů se generuje automatickými nástroji z naprogramované služby.

Výhody:

- Rozšiřitelnost
- Implementace v nejpoužívanějších jazycích

Nevýhody:

- Komplexnost (oproti jiným jako např. REST)

1.2.4 REST přes HTTP

Representation State Transfer je architektura, kterou v roce 2000 zveřejnil Roy Fielding v dizertační práci. Proč zrovna Representation State Transfer? Klient posílá požadavek na URI zdroje, server mu vrací reprezentaci (HTML, XML, JSON). Při přijetí odpovědi se klient dostává do určitého stavu (State), dává požadavek na další zdroj a přechází (Transfer) do dalšího stavu (State) [7].

Rest architektura má několik rysů, které musí splňovat:

- **Klient – Server** – oddělení uživatelského rozhraní od úložiště dat, příkladem z webového prostředí je klasický prohlížeč VS webový server, databázový server
- **Bezstavovost –** každý požadavek musí obsahovat všechny informace, které jsou potřebné ke zpracování požadavku, nepočítá se s žádným ukládáním SESSION na serveru, atd.
- **Kešovatelnost** – požadavky by měly mít u sebe označení, jestli jsou kešovatelné nebo ne, při možnosti kešovaní by se tato věc měla využít, ve webovém prostředí například kešovací proxy
- **Jednotné rozhraní** – všechny zdroje by měly mít jednotné rozhraní, které potřebuje minimální popis

V našem případě se bavíme o REST implementované pomocí HTTP. O bezstavovost se nemusíme starat, ta je v HTTP protokolu odjakživa. Další věcí je uniformní rozhraní, které

spočívá v namapování http metod na jednoduché akce, které mohou být prováděny se zdrojem.

1.2.4.1 HTTP vs. REST

Popis namapování HTTP metod používaných na akce se zdroji:

- **GET** - tento požadavek je pouze čtecí, se zdrojem nic neprovádí, jenom vrací jeho reprezentaci.
- **PUT** – s PUT by měla na server putovat data, která budou uložena, v případě PUT jde o vytvoření nového zdroje.
- **DELETE –** zde jde pouze o vymazání zdroje, který je dán vlastností v URL
- **POST** – Tato metoda slouží pro úpravu zdroje, v těle požadavku přijde reprezentace zdroje a aplikace by měla být schopna daný zdroj dle této reprezentace zaktualizovat

1.2.4.2 Reprezentace

Pro uvození reprezentace použijeme hlavičku Content-Type, která řekne klientovi, jaký typ média má očekávat. Je na návrháři, kterou z nich vybere - zda XML, YAML nebo JSON.

1.2.4.3 Řešení chybových stavů

Jelikož REST nevyužívá HTTP protokol jen jako transportní vrstvu jako například SOAP, je pro chybový stav využita část odezvy HTTP protokolu. V tabulce Tab. 3 byly ukázány skupiny kódů, které využívá HTTP, stejně je využívá i REST. Návrháři se nabízí otázka, k čemu konkrétní kód využije. Dále je na klientovi, jak s danými návratovými kódy bude hospodařit on.

Výhody

- jednoduché
- jednotné rozhraní

Nevýhody

- na to, jak dlouho již tato technologie existuje, je docela málo ve frameworcích podporována (i když implementace není nic složitého). Pokud je naimplementována, tak nejsou dodrženy všechny náležitosti REST architektury, jak je vidět např. v Zend_Rest_Client, Zend_Rest_Server.

REST architektura implementovaná pomocí HTTP zde dostala větší prostor, protože byla vybrána pro implementaci aplikace.

2 KNIHOVNA PRO VIZUALIZACI GRAFŮ

2.1 Co by měla umět

Jelikož knihovnu budeme používat v další praktické části, tak na ni máme požadavky, které se objevily při analýze.

Knihovna by měla umět:

- zobrazit různé obrazce v různé barvě spojené do neorientovaného grafu
- lze jednoduše manipulovat s prvky grafu
- každá manipulace vyvolá událost – tzn., grafová komponenta by měla mít vlastní událostní model
- každý obrazec (uzel) grafu by měl mít možnost obsahovat poduzly
- celý model by měl jít jednoduše zvětšovat a zmenšovat
- Javovská implementace
- musí být vydána pod GPL kompatibilní licencí

2.2 Knihovna JGraph

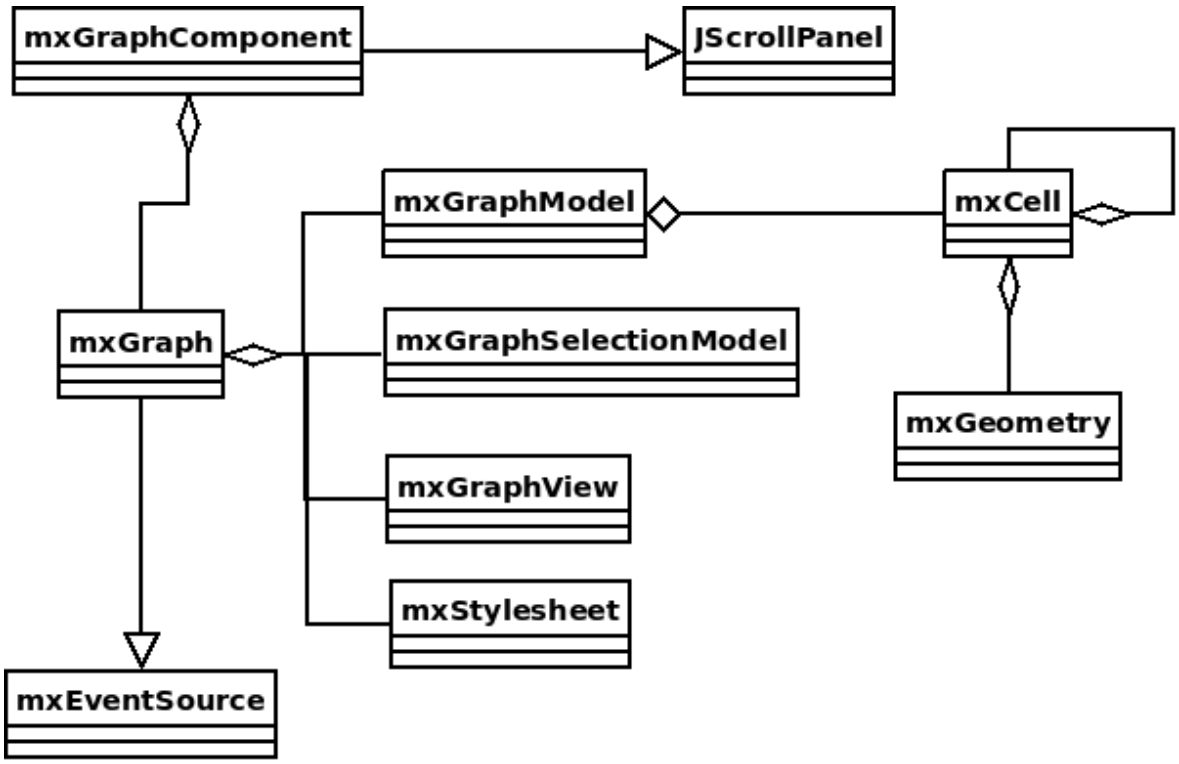
Knihovna je vyvíjena od roku 2000. Nejprve byla pouze pro Javu. V posledních letech se rozšířila o implementace v JavaScriptu a Flashi. Flash je nyní v betaverzi, takže použití tohoto modelu není dobrou volbou. Pokud jde o Javascript, sami autoři nabádají v dokumentaci, aby tato aplikace byla použita pouze pro jednodušší účely, protože Javascriptové enginy prohlížečů nejsou ještě tak výkonné, aby mohly zpracovávat nějaký složitější model. Dalším mínusem pro MxGraph je skutečnost, že je komerční.

JGraph 5 je verze, která je vyvíjena a podporována od roku 2000. Autoři se během minulého roku rozhodli, že celou architekturu celého grafového frameworku zjednoduší. Bylo to víceméně přepsání celého frameworku od počátku. Aby se úplně oddělila vývojová linie JGraph 5 a nového frameworku, neexistuje další číslo verze 6. Framework se přejmenoval na JGraphX. Sami autoři na vývojářském fóru nabádají programátory, aby nové projekty vyvíjeli v JGraphX, jelikož podpora JGraph 5 již tento rok končí.

V další části bude představena základní architektura a práce s novou knihovnou JgraphX. Jsou zde vybrány jen důležité rysy, které by se mohly hodit vývojáři pro rozhodnutí, že má vybrat právě tuto knihovnu.

Poslední věcí, na kterou je třeba upozornit, je licence knihovny. Jak původní JGraph, tak současný JGraphX jsou vydávány pod BSD licencí.

2.2.1 Základní architektura



Obr. 2 Základní zjednodušená architektura knihovny JGraph

Základem celé knihovny Jgraph je třída *mxGraph*, která nám zapouzdřuje všechny podtřídy potřebné k vykreslení grafu. Je to také jednotné rozhraní pro přístup ke všem komponentám grafu. Je děděná z *mxEventSource*, který v sobě zapouzdřuje správu událostí a jejich spouštění.

Agregované třídy v třídě *mxGraph*:

- *mxGraphModel* – modelová třída, která implementuje rozhraní *mxIGraphModel*, takže má definovány metody na úpravu uzlů jako přidávání, změna, je daný uzel otcem některého uzlu, vrácení kořenového elementu atd.

- *mxCell* – je třída základního uzlu, jak je vidět na obrázku Obr. 2, může obsahovat sama sebe, a to v četnosti 1..N, obsahuje metody pro zjištění, zda daný bod je hrana nebo uzel, polohu uzlu atd.
- *mxGeometry* – každá mxCell obsahuje i mxGeometry, která udává jeho grafické ztvárnění, tzn. polohu barvu, relativnost vůči jinému elementu atd.
- *mxGraphSelectionModel* – je kolekce vybraných uzlů
- *mxGraphView* – je to taková „cache" kolekce, kde jsou uloženy všechny uzly s jejich absolutními souřadnicemi a pokud se něco překresluje, tak v rámci optimalizace se snaží aplikace nepřepočítávat celý model, ale vykreslovat ho odsud
- *mxStylesheet* – tato třída je něco podobného jako CSS v HTML, lze si zde definovat jednotlivé styly, které je pak možné pojmenovat a použít při vykreslování
- *mxGraphComponent* – je vlastní vizuální třída, která vykresluje graf na nějaké plátno. Můžeme vidět, že třída dědí z JScrollPanel, takže je kresleno přímo na Swingovské skrolovací plátno.

2.2.2 Použití frameworku JgraphX

2.2.2.1 Přidávání uzlů

Každý by asi předpokoládal, že na obrázku Obr. 2 by měl model obsahovat dvě třídy *mxCell* a např. *MxEdge*. Ovšem v JGraph tomu tak není. Jak hrany grafu, tak uzly jsou zapouzdřeny ve třídě *mxCell* a ke zjištění, zda je daná buňka hrana nebo uzel, slouží metody *isVertex()* nebo *isEdge()*.

Následuje ukázka přidání dvou uzlů a jedné hrany [8]:

```
graph.getModel().beginUpdate();
try
{
   Object v1 = graph.addVertex(parent, null, "Hello,", 20, 20, 80, 30);
   Object v2 = graph.addVertex(parent, null, "World!", 200, 150, 80, 30);
   Object e1 = graph.addEdge(parent, null, "", v1, v2);
}
finally
```

```
{
  graph.getModel().endUpdate();
}
```

V kódu můžeme pozorovat, že vlastní přidávací příkazy jsou uzavřeny v bloku *try..finally*. Veškeré pohyby by měly být kvůli správnému vykreslování a správné funkci navracení kroků uzavřeny mezi metodami *beginUpdate()* a *endUpdate()*. Po endUpdate() je zobrazovací komponenta překreslena a jsou vyvolány události, které má definovány JGraph. Nyní tedy už k vytvoření vlastních uzlů. Jak je vidět, kód vytváří dva uzly v1 a v2. Metody pro vytváření jsou volány z třídy *mxGraph*, jak už jsme zmiňovali dříve. Jsou to jen odkazy na metody ve třídě *mxGraphModel*, jelikož třída mxGraph tvoří jakési jednotné rozhraní (proxy) pro ostatní agregované třídy.

Signatura metody insertVertex:

```
mxGraph.insertVertex(parent, id, value, x, y, width, height, style)
```

- **parent –** rodič uzlu, jelikož hrana může být klidně i mezi dvěma uzly, které jsou uvnitř nějakého jiného uzlu
- **id** – jednoznačný identifikátor uzlu, který slouží pro perzistenci grafu, jinak může být null
- **value –** uživatelsky definovaná hodnota uzlu
- **x, y –** souřadnice bodu na plátně
- **width, height** – šířka a výška uzlu
- **style** – definice stylu uzlu – popsáno níže

Signatura metody insertEdge:

```
mxGraph.insertEdge(parent, id, value, source, target, style)
```

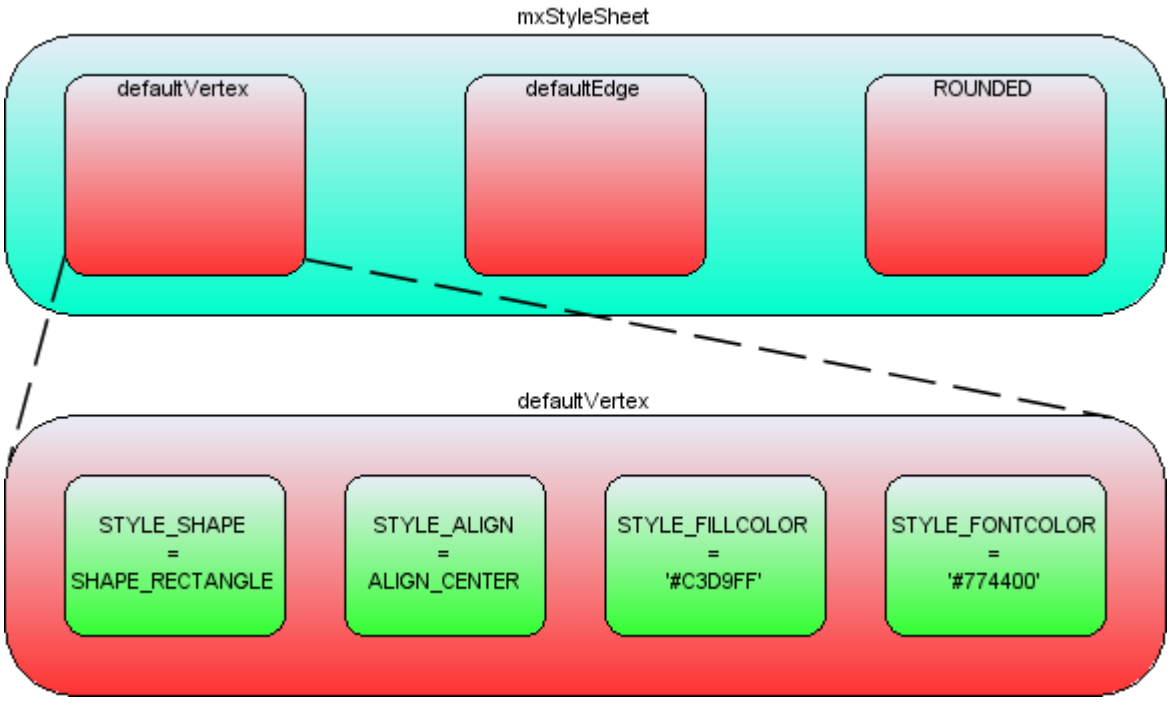
- **parent** – rodič hrany, jelikož hrana může byt i mezi uzly, které jsou uvnitř jiného uzlu
- **value** – uživatelsky definovaná hodnota hrany
- **source** – počáteční uzel
- **target** – cílový uzel
- **style** – styl hrany, který bude popsán níže

2.2.2.2 Stylování hran a uzlů

Při inicializaci JGraph jsou do *mxStylesheet* zaznamenány dva základní styly a to *defaultVertex* a *defaultEdge* (Obr. 3). Dále je možno si přidávat jakékoliv další nové styly a ukládat je do výše zmíněného objektu třídy mxStylesheet. Na obrázku vidíme přidaný styl ROUNDED, který může být definován například [8]:

```
mxStylesheet stylesheet = graph.getStylesheet();
Hashtable<String, Object> style = new Hashtable<String, Object>();
style.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_RECTANGLE);
style.put(mxConstants.STYLE_OPACITY, 50);
style.put(mxConstants.STYLE_FONTCOLOR, "#774400");
stylesheet.putCellStyle("ROUNDED", style);
```

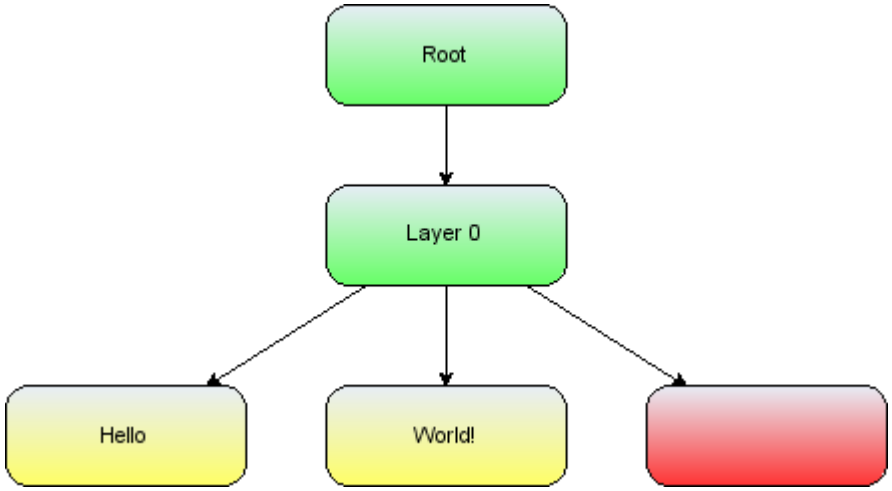
V kódu se takový styl přidává jako poslední řetězcový parametr do metody *insertVertex* nebo *insertEdge*.



Obr. 3. Stylování v JGraph

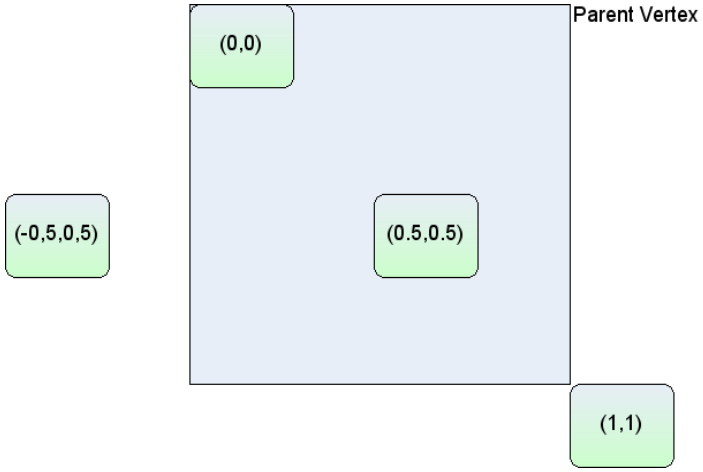
2.2.2.3 Geometrie uzlu

Geometrie uzlu nám udává jeho polohu vzhledem k jeho rodiči. Obr. 4 nám ukazuje celý model grafu, jenž vypadá tak, že máme jeden kořen (root). Pod kořenem jsou tzv. vrstvy (layer), které ještě nejsou viditelné. Následují hrany a uzly viditelné na plátně. Standardně se inicializuje pouze jedna vrstva. Při vytváření uzlu metodou insertCell se geometrie vytvaří automaticky a vývojář vůbec nepřijde do styku s třídou mxGeometry.



Obr. 4. Rozložení modelu grafu

Ovšem jsou případy, kdy je potřebné nastavit u geometrie některé další vlastnosti. Jelikož jsou standardně všechny uzly vytvářeny absolutně k hlavní vrstvě, tak je velice složité pozicovat poduzly viditelných uzlů. Proto u geometrie existuje metoda *setRelative*, která udává, že se bude objekt pozicovat relativně k jeho rodiči v souřadném systému naznačeném na obrázku.



Obr. 5. Relativní pozicování uvnitř elementu

2.2.2.4 Value objekty

Každému uzlu je možné přiřadit nějaký uživatelský objekt, který přidá danému uzlu význam. Od toho jsou zde právě Value objekty a dá se říci, že hrají roli jakési business vrstvy celého grafu.

3 VYBRANÉ TECHNOLOGIE

3.1 Serverová část

Na straně serveru nebyla volba možná, jelikož je vyvíjen klient k hotové aplikaci.

3.1.1 PHP

PHP je interpretovaný slabě typový skriptovací jazyk, používaný hlavně v oblasti webu na straně serveru. Je většinou používán jako modul do webovéhu serveru Apache. Apache při HTTP požadavku klienta na zdroj zkontroluje, jestli daný zdroj nemá příponu, kterou by měl zpracovat pomocí modulu PHP. Pokud tomu tak je, skript se předá modulu PHP, který příkazy vykoná nejčastěji mezi značkami *<?php ?>* a výstup vrací Apachi zpět. Ten ho posílá zpět jako HTTP odpověď. Nejpoužívanější verzi v dnešní době obstarává 5.2.

3.1.2 Kohana

Dnes se v PHP nezačíná s žádným projektem na zelené louce. Pro urychlení vývoje se používají tzv. frameworky většinou založené na návrhovém vzoru MVC. Jedním takovým je i Kohana, která je blíže popsána v teoretické části prací [1] a [2].

3.1.3 MySQL

Databázový stroj MySQL je v dnešní době jedním z nejvíce používaných serveru na střední a malé aplikace. Dříve byl velmi jednoduchý, ale dnes obsahuje veškeré vymoženosti robustnějších databázových strojů jako je transakční zpracování, uložené procedury. S PHP drží u webhosterů největší zastoupení na světě, proto je také používán u velkého množství webových open source projektů. Nyní jsou nejpoužívanější MySQL verze 5 a 5.1.

3.2 Klientská část

3.2.1 Java

Javu můžeme označit za velmi používaný silně typový objektový multiplatformní jazyk. Každý program je přeložen do tzv. bytekódu, který pak můžeme interpretovat na kterémkoliv operačním systému. Takový systém má implementováno běhové prostředí,

které se nazývá JRE a v sobě má zahrnutu Java Virtual Machine (JVM). Ta sama se stará o spuštění kódu. Interpretace bytekódu by určitě nebyla nijak rychlá, ale dnes již JVM používá tzv. Just in time kompilaci, která nejpoužívanější částí kódu kompiluje do strojového kódu dané platformy za běhu.

Java se dělí do tří balíčků:

- Java ME – pro vývoj pro mobilní zařízení
- Java SE – pro vývoj desktopových aplikací
- Java EE – určená pro enterprise aplikace

3.2.2 JFC/Swing

Java Foundation Classes je knihovna Javovských tříd, která zahrnuje i vizuální komponenty které se souhrnně nazývají Swing. Swing je grafický toolkit, který obsahuje jak většinu známých GUI komponent (tlačítka, editační pole, listy, stromy), tak rozvrhovače (Layout managery). Swing dále umožňuje různé tzv. Look & Feel, tzn., že aplikace může vypadat na různých platformách relativně podobně.

3.2.3 Jersey API

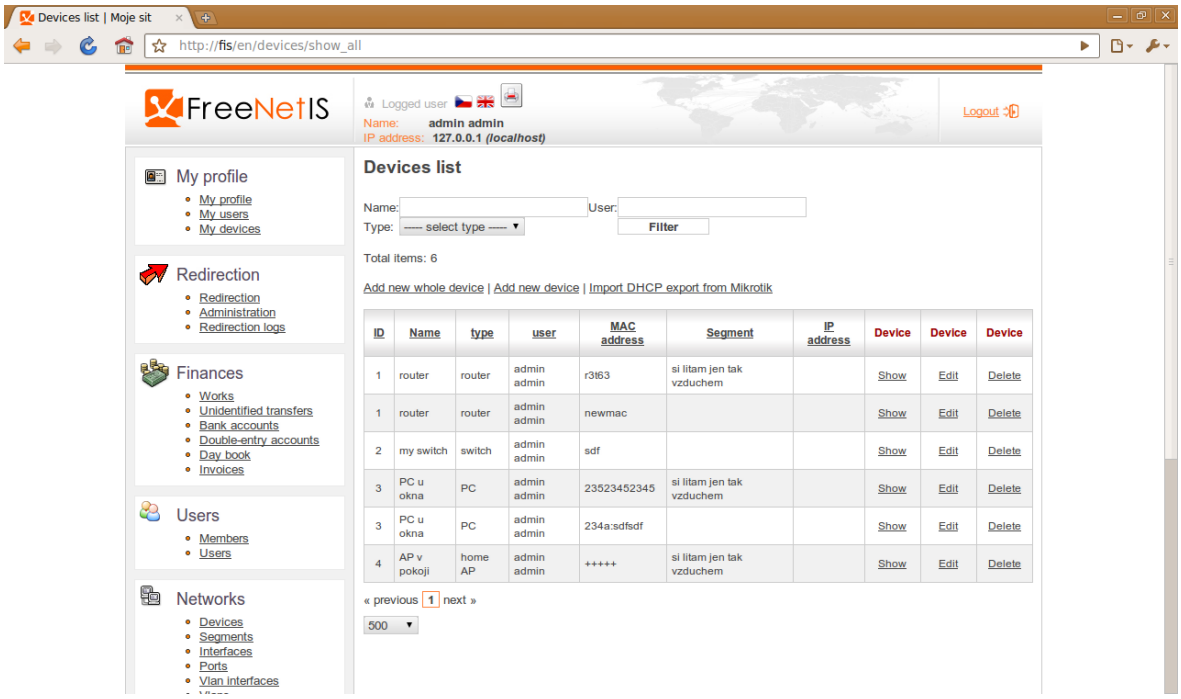
Jersey API je referenční implementace RESTových webových služeb v Javě. Definuje jak podporu pro webové služby, tak pro klientské aplikace. Hlavní výhodou je podpora anotací, pomocí kterých lze jednoduše uvozovat Javové třídy, atributy a metody. Těmito anotacemi definujeme REST rozhraní u serverů. U klientů se anotace používají hlavně pro mapování reprezentací, které přijdou jako odpovědi REST služby, na konkrétní Javové třídy.

II. PRAKTICKÁ ČÁST

4 ANALÝZA

4.1 Problém stávajícího řešení

Problémem stávajícího řešení, jak už bylo naznačeno v úvodu, je nepohodlná editace schématu sítě, ve kterém je i velmi složité se orientovat (Obr. 6). Proto je žádoucí vyvinout klienta, který by dokázal komunikovat s webovou aplikací a při klikání v grafickém prostředí rovnou ukládat data do databáze na serveru.



Obr. 6 Ukázka webového rozhraní FreeNetIS

FreeNetIS nyní pracuje s několika entitami, které jsou důležité pro klientskou aplikaci:

- zařízení – jakékoliv zařízení (PC, router, AP), které může obsahovat porty a rozhraní
- segment – je fyzická technologie spojení, tzn., že vyjadřuje spojení mezi porty nebo rozhraními (UTP, optika, WIFI)
- port – je fyzická zdířka na switchy, která nemá MAC adresu
- rozhraní - je fyzická zdířka na jakémkoliv zařízení, která má svoji MAC adresu
- uživatel – každé zařízení je u uživatele doma nebo má za něj uživatel zodpovědnost

Dalším důležitou věcí, s kterou je třeba počítat, je zapojení více rozhraní do jednoho rozhraní. Tato situace nastává například při bezdrátovém spojení, kdy je více klientů připojeno k jednomu bezdrátovému rozhraní

Dalším problémem je zobrazení rozlehlejší sítě. Ta by byla dosti nepřehledná, kdyby byly vidět všechny detaily. Proto by bylo žádoucí dělit zařízení do skupin, které jsou strukturovány do stromu. Skupinu si lze představit jednoduše, jako když v emailovém klientu uživatel dává e-mailům vlajky. Zde je ovšem rozdíl v tom, že každé zařízení by bylo pouze v jedné skupině. Ve více by to ostatně nemělo ani smysl.

5 ÚPRAVY NA STRANĚ SERVERU

5.1 Rozšíření databáze

Úpravy databázového schématu plynou z potřeb klientské aplikace. Základním stavebním kamenem sítě jsou zařízení (routery, switche, PC atd.), které si uživatel v klientu nějakým způsobem rozmístí po obrazovce. Z toho plyne, že bude nutné držet jejich polohu v tabulce Devices. Více úprav pro základní funkčnost není třeba.

5.2 Návrh REST rozhraní

Při návrhu rozhrání služby založené na REST architektuře se postupuje podobně jako při návrhu jakéhokoliv jiného softwaru. Platí, že prvním, čím začneme, je definice funkčních požadavků. [9]

Požadavky na službu:

- systém obsahuje množství typů zařízení, které do kterých lze zařízení zařazovat
- systém obsahuje druhy přenosových medií, která lze vybírat při vytváření a editaci zařízení
- každé zařízení je přiřazeno nějakému uživateli
- každé zařízení se nachází na nějakém místě, které je dáno ulicí a městem
- v systému jsou zařízení (device), která by měla jít přidávat, odebírat nebo měnit
- v systému jsou spoje (segment) mezi zařízeními, která lze přidat, odebrat, nebo měnit jejich charakter
- zařízení může být zařazeno do skupiny, která nám umožňuje strukturovat model sítě
- zařízení lze ze skupiny vyjmout
- systém musí umět vrátit pouze zařízení z dané skupiny

V požadavcích jsou podtržena podstatná jména, která by mohla být zdroji. Dále jsou v požadavcích slovesa, což jsou nejspíše operace s danými zdroji.

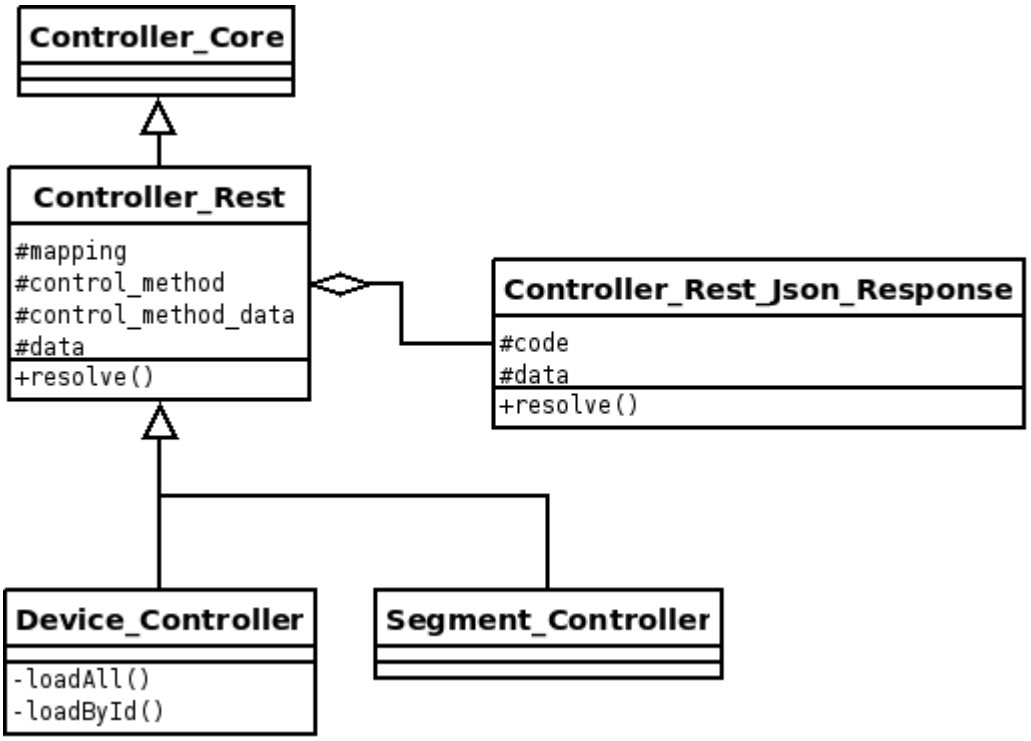
Dalším krokem je mapování zdrojů a operací na URL a HTTP metody, jelikož používáme implementaci REST architektury pomocí HTTP protokolu. Zdroje budeme pojmenovávat anglicky dle dříve dané terminologie v databázovém modelu.

Tab. 5. Mapování metod na URL pro REST zdroje

Metoda HTTP	URL	Popis
GET	/device-type	Vrátí seznam všech typů zařízení
PUT	/device-type/{id}	Úprava typu zařízení
POST	/device-type	Vytvoření typu zařízení
DELETE	/device-type/{id}	Smazání typu zařízení
GET	/media-type	Vrátí seznam všech typů médií
PUT	/media-type/{id}	Úprava typu média
POST	/media-type	Vytvoření typu média
DELETE	/media-type/{id}	Smazání typu média
GET	/device	Vrátí seznam všech zařízení
PUT	/device/{id}	Úprava zařízení
POST	/device	Vytvoření zařízení
DELETE	/device/{id}	Smazání zařízení
GET	/group	Vrátí seznam skupin
GET	/group/{id}	Vrátí všechny média, která se nachází v dané skupině
GET	/street	Vrátí všechny ulice v aplikaci
GET	/town	Vráti všechny města v aplikaci

5.2.1 Vlastní implementace REST rozhraní v KOHANě

Bohužel Kohana neobsahuje implementaci REST rozhraní, proto nezbylo než si udělat implementaci vlastní. Velmi pěkné řešení má například Java, kde můžeme každou metodu uvozovat anotacemi. Označí se, na jaký typ HTTP metody bude aplikace reagovat, v jaké reprezentaci budou posílána data a na jaké URL je daná metoda namapována. To ovšem v PHP není možné. Proto, abychom mohli zpracovávat REST požadavky, je nutné si nejdříve vytvořit vlastní kontroler. V praxi to znamená, že definujeme vlastní kontroler(*Controller_Rest*), ze kterého budou dědit všechny kontrolery, které chtějí mít REST rozhraní (Obr. 7).



Obr. 7. Architektura REST na serveru

Pole *mapping* v *Controller_Rest* představuje mapování metod na URL. Následuje příklad pole *mapping*:

```
protected $mapping = array(
        array(
                'method' => 'GET',
                'path' => '^(?P<id>\d+)$',
```

```
                'control_method' => 'loadById'
        ),
        array(
                'method' => 'POST',
                'path' => '^$',
                'control_method' => 'save'
        );
```

Vidíme, že první pole obsahuje mapování HTTP metody GET na metodu kontroleru *loadById*. Položka path v poli nám udává regulární výraz, kterému bude vystavena URL. Pokud URL odpovídá regulárnímu výrazu, bude uložena do control_method_data položka *id*.

Nedůležitější je metoda resolve, kterou dále popisuji.

```
try {
        $method = strtoupper(request::method());
```

Celá metoda je uzavřena do bloku try..catch, protože při každém vyhození výjimky se musí vrátit návratový kód 500 HTTP protokolu.

```
        $data_start = strpos(Router_Core::$current_uri, Router_Core::$controller) +

                        strlen(Router_Core::$controller);
        $data_path = trim(substr(Router_Core::$current_uri, $data_start), '/');
```

Zde se vytahuje část URL, která se bude stavět proti regulárnímu výrazu.

```
        foreach($this->mapping as $map)
        {
                $matches = array();
                if(strcmp($map['method'], $method) == 0 &&
                preg_match("#".$map['path'].'#', $data_path, $matches))
```

Pro každý element v poli *mapping* se provede, jestli HTTP metoda odpovídá metodě, kterou se dotazoval klient. A dále se porovná URL s regulárním výrazem uvedeným v poli pod klíčem *path*.

```
                {
                        $this->control_method_data = $matches;
                        $this->control_method = $map['control_method'];
                        break;
```

Pokud dojde k nalezení metody, tak se uloží data z URL do control_method_data. Pokud nebude nalezena žádné odpovídající mapování, je vyhozena výjimka hned po cyklu.

```
                }
        }
        if($this->control_method === NULL) {
                throw new Kohana_Exception('No rest method found!');
        } else {
                if(strcmp($method, 'PUT') == 0 ||
                strcmp($method, 'POST')== 0)
                {
                        $data = file_get_contents("php://input");
                        $this->data = json_decode($data);
```

Pokud je požadavek proveden metodou PUT nebo POST, tak data která přijdou, se musí číst ze standardního vstupu. Očekává se, že jsou ve formátu JSON.

```
                }
                $this->{$this->control_method}($this->control_method_data);
```

Vlastní zavolání metody controlleru.

```
        }
} catch(Exception $e) {
        $this->response->setCode(500);
        $this->response->resolve();
}
```

Zpracování REST požadavku:

1. Uživatel provede REST požadavek například na /rest/media-type/10.
2. Kohana přijme požadavek souborem index.php a provede všechny důležité inicializace.
3. Zinicializuje Router a podívá se na pravidla v souboru application/config/routes.php.
4. V souboru najde pravidlo, které odpovídá naší cestě a převede ho na formát /rest/media-type/resolve/10.
5. Router najde v adresáři application/controllers/rest/ soubor, který se jmenuje media_type.php naincluduje ho a provede metodu resolve.

6. Metoda resolve provede vše, jak je uvedeno výše a zavolá konkrétní metodu kontroleru, v tomto případě *Media_Type_Controller*.
7. Sám kontroler již přebírá zodpovědnost za odpověď a posílá data zakódována v JSONu pomocí objektu třídy Controller_Rest_Json_Response. Controller_Rest_Json_Response je třída, která vrací validní HTTP odpověď s daty kódovanými v JSONu.

6 KLIENTSKÁ APLIKACE

Aplikace na klientovi je tvořena jako Javová desktopová. Je založena na GUI toolkitu Swing, tudíž by neměl být problém ji zveřejnit jako applet, v informačním systému FreeNetIS.

6.1 Architektura aplikace

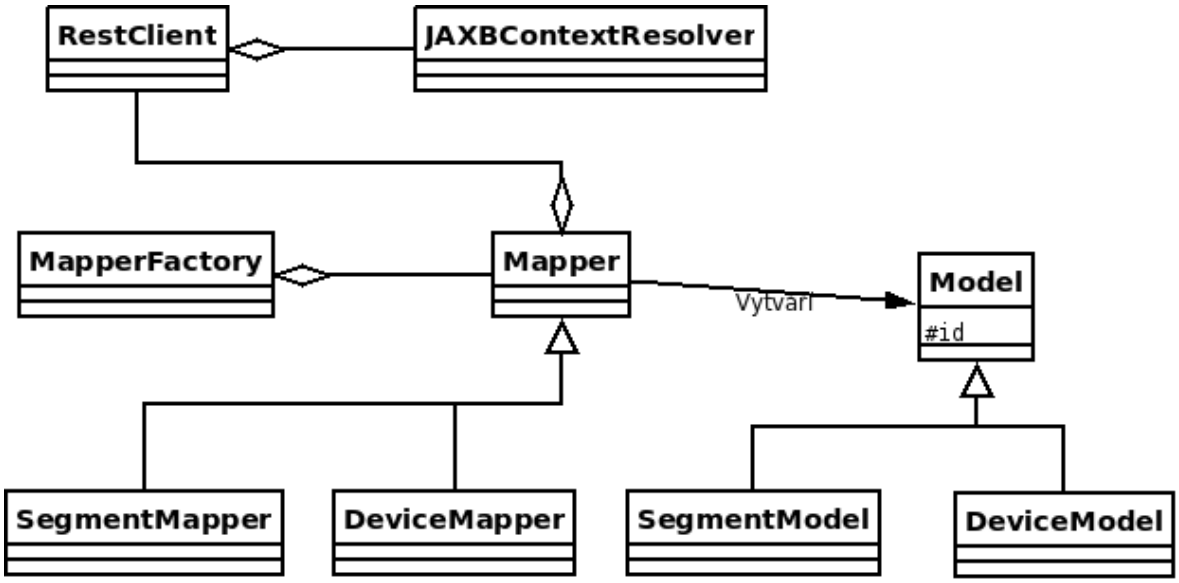
Aplikace je rozdělena do balíčků:

- **org.freenetis.editor –** obsahuje třídy, které se týkají GUI a je zde i třída FISClient, která obsahuje metodu main, tedy vstupní bod aplikace
- **org.freenetis.editor.editpanel** – třídy týkající se hlavního editačního panelu aplikace, tedy vlastního editoru
- **org.freenetis.editor.ui** – některé rozšířené gui elementy
- **org.freenetis.mappers** – třídy, které mapují příchozí JSON požadavky na objektové modely
- **org.freenetis.model** – modelové třídy, na které se mapují requesty
- **org.freenetis.rest** – třídy, které se starají o REST komunikaci a využívají Jersey API

6.2 Implementace klientského REST rozhraní

Pro REST komunikaci je v aplikaci využito Jersey API. Komunikační rozhraní využívá dvou tříd *JAXBContextResolver* a *RestClient*, (Obr. 8). Rest Client zapouzdřuje Jersey API, tzn., že v aplikaci se pracuje již jen minimálně přímo s Jersey API. RestClient je singleton, jelikož instanciovat klienta Jersey je velmi drahá záležitost. Když RestClient provádí mapování, samozřejmě musí vědět, kde má třídy, na které mapuje, hledat. Od toho je zde JAXBContextResolver, který se využívá při inicializaci klientského Jersey API a zjednodušeně vrací kontext se všemi třídami, které by mohl použít někdy při mapování (v naší aplikaci je to celý balíček org.freenetis.model). Dále se ještě v JAXBContextResolveru nastavuje, z jaké reprezentace se bude mapovat. Je zde použit JSON a notace natural.

RestClient je napojen na rodiče všech tříd Mapper (Obr. 8.). Při první incializaci kteréhokoliv z mapperů je *RestClient* vytvořen a uložen do statické vlastnosti client. Od té chvíle již všechny Mappery využívají stejnou instanci třídy *RestClient*.



Obr. 8. Část modelu tříd zajišťující REST

Dále je zde vidět třída MapperFactory, ta v sobě agreguje všechny již vytvořené mappery. To znamená, že pokud je potřeba nějaký mapper, tak se zavolá:

```
(DeviceMapper) dm = (DeviceMapper) MapperFactory.get(DeviceMapper.class)
```

Tím je zajištěné, že všechny třídy mapper jsou v aplikaci také pouze jednou.

Všechny modely mají společného rodiče, který má v sobě atribut id. Je to primární klíč daného modelu v databázi na serveru. Podle něho se také poznává, jestli je daný model nově ukládaná entita, tzn. ukládaná pomocí HTTP metody PUT nebo jen upravovaná nějaká již dříve vytvořená, tzn. ukládaná pomocí metody HTTP POST.

Dále se zaměříme na průběh vlastního mapování, které je vyznačeno vazbou zprávy s anotací „Vytváří“ (Obr. 8). Máme následující třídu *Device*.

```
@XmlRootElement(name="Device")
@XmlAccessorType(XmlAccessType.NONE)
public class Device extends Model {
        private User user;
        private DeviceType type;
        private AddressPoint addressPoint;
        @XmlElement(name="user_id") private int userId;
```

```
@XmlElement(name="address_point_id") private int addressPointId;
@XmlElement(name="ifaces", nillable=true) private List<Iface> ifaces;
@XmlElement(name="ports", nillable=true) private List<Port> ports;
@XmlElement(name="name") private String name;
@XmlElement(name="comment") private String comment;
@XmlElement(name="trade_name") private String tradeName;
@XmlElement(name="login") private String login;
@XmlElement(name="type") private Integer typeId;
@XmlElement(name="posx") private Integer posx;
@XmlElement(name="posy") private Integer posy;
```

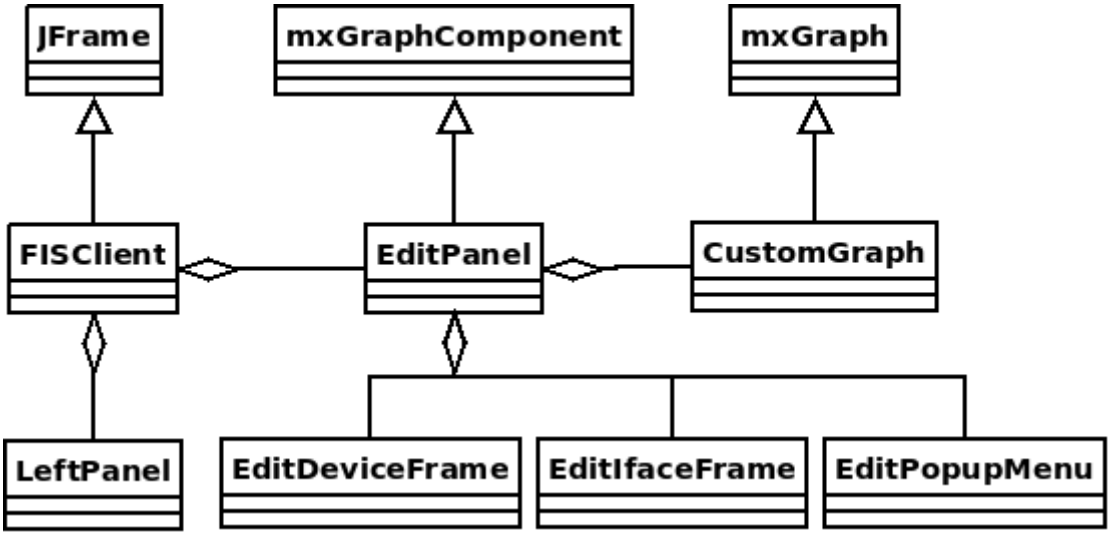
Každá odpověď, která přijde ze serveru, obsahuje JSON reprezentaci takové třídy. Dotaz na JSON, který bude možné mapovat na třídu výše, je možné provést například takto:

```
Device d = dm.getObjectById(10);
```

Při zavolání toho příkazu se provedou příslušná mapování. Například mapování type položky z JSONu na *typeId* atribut třídy *Device*. Pozorujeme, že je možné mapovat celé kolekce objektů, jak je například vidět u vlastnosti *ifaces*. Zde nám vznikne po mapování kolekce ArrayList, která bude obsahovat objekty třídy *Iface*. Toto mapování zabezpečuje Jersey a atributy, které má mapovat pozná podle @XmlElement anotací. Na začátku třídy je anotace @XmlRootElement, ta ukazuje, kde daná reprezentace začíná. Anotace @XmlAccessorType(XmlAccessType.NONE) označuje, že se nemají mapovat všechny atributy třídy, ale pouze ty, které jsou uvozené anotací. Je trošku nesmyslné, že anotace mají prefix Xml, ale pracuje se s JSONem, To, jak už bylo dříve řečeno, se nastavuje v JAXBContextResolver.

6.3 GUI a implementace JGraph v aplikaci

Vstupním bodem aplikace je třída FISClient, která vytváří grafický layout celého GUI. LeftPanel je potomkem JPanel a obsahuje levý panel s vlastnostmi objektu, když na něj klikneme.



Obr. 9. Zjednodušený diagram GUI komponent a komponent grafu

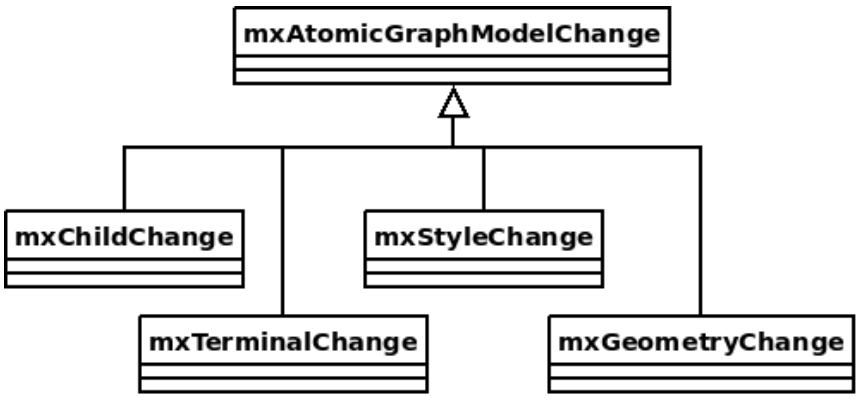
Další komponentou, kterou FISClient agreguje je EditPanel, což je vlastní kreslící plátno celé aplikace dědící z mxGraphComponent. V EditPanelu se reaguje na většinu událostí, které generuje komponenta modelu JGraphu. V konstruktoru EditPanelu se připojují obsluhy událostí následovně:

```
graph.getModel().addListener(mxEvent.CHANGE, new mxIEventListener());
```

mxEventListener obsahuje metodu invoke, která musí být definovaná. Její signatura vypadá následovně:

```
void invoke(Object sender, mxEventObject evt);
```

V objektu evt třídy *mxEventObject* jsou předány všechny potřebné informace proto, aby byl vývojář schopen zjistit, co se vlastně stalo. V objektu *evt* proto využíváme metodu *getProperties,* která nám vrací HashMapu s vlastnostmi událost. Pod klíčem changes je uložena kolekce objektů *mxAtomicGraphModelChange*.



Obr. 10. Třídy, které značí změny v grafu

Každá podtřída mxAtomicGraphModelChange značí nějakou změnu v modelu. Na obrázku Obr. 10 je ukázáno jen několik z možných objektů změn.

- **mxChildChange –** byl přidán uzel nebo hrana
- **mxTerminalChange –** byl změněn jeden z koncových uzlů hrany
- **mxStyleChange** – došlo ke změně stylu uzlu nebo hrany
- **mxGeometryChange –** změna polohy uzlu

Zpracování takové události probíhá iterací přes pole změn. Příkladem může být změna polohy zařízení v klientu:

```
Map<String, Object> m = evt.getProperties();
for(mxAtomicGraphModelChange change : (ArrayList<mxAtomicGraphModelChange>)
m.get("changes")) {
        if(change instanceof mxGeometryChange) {
                mxCell c = (mxCell) ((mxGeometryChange) change).getCell();
                Device d = (Device) c.getValue();
                d.setPosx((int) c.getGeometry().getX());
                d.setPosy((int) c.getGeometry().getY());
        }
}
```

6.3.1 Vykreslování grafu sítě

Vykreslování celého grafu provádí metoda *drawGroup(Group g),* která je součástí třídy *customGraph.* Metoda přebírá model *Group*, který obsahuje souhrnná data o zařízeních, segmentech, portech a rozhraních. Následuje popis metody:

```
public void drawGroup(Group g) {
```

Vytvoření *HashMap*, které nám budou sloužit k rychlejšímu přístupu k jednotlivým objektům dle databázového id.

```
        Map<String, mxCell> devicesInCells = new HashMap<String, mxCell>();
        Map<String, mxCell> ifacesInCells = new HashMap<String, mxCell>();
        Map<String, mxCell> portsInCells = new HashMap<String, mxCell>();
        Map<String, Segment> segments = new HashMap<String, Segment>();
        for(Segment s : g.getSegments()) {
                segments.put(s.getId().toString(), s);
```

```
        }
        getModel().beginUpdate();
        try {
                mxCell deviceVertex, innerVertex;
                Object parent = getDefaultParent();
```

Nejprve jsou vykreslena zařízení a reference jejich uzlů jsou uloženy do *devicesInCells*.

```
                for(Device device : g.getDevices()) {
                        deviceVertex = (mxCell) insertVertex(parent, null, device,
                                        device.getPosx(), device.getPosy(), 100, 30);
                        devicesInCells.put(device.getId().toString(), deviceVertex);
                }
```

Dále jsou vykreslována rozhraní, kde je jako druhý parametr v metodě addCell předáno rodičovské zařízení a dané geometrie jsou označeny jako relativní, tudíž se vykreslují relativně ke svému rodičovskému zařízení.

```
                mxGeometry vGeo;
                for(Iface iface : g.getIfaces()) {
                        vGeo = new mxGeometry(0, 0.3, 10, 10);
                        vGeo.setRelative(true);
                        innerVertex = new mxCell(iface, vGeo, "fillColor=white");
                        innerVertex.setVertex(true);
                        innerVertex = (mxCell) addCell(innerVertex,
                                        devicesInCells.get(iface.getDeviceId().toString()));
                        ifacesInCells.put(iface.getId().toString(), innerVertex);
                }
```

Zde se stejně jako rozhraní přidávají porty.

```
                for(Port port : g.getPorts()) {
                        vGeo = new mxGeometry(0, 0.3, 10, 10);
                        vGeo.setRelative(true);
                        innerVertex = new mxCell(port, vGeo, "fillColor=black");
                        innerVertex.setVertex(true);
                        innerVertex = (mxCell) addCell(innerVertex,
                                        devicesInCells.get(port.getDeviceId().toString()));
                        portsInCells.put(port.getId().toString(), innerVertex);
                }
```

Následuje uspořádání portu do layoutu, tzn. v řadě vedle sebe, podle přidání posloupnosti přidání zařízení.

```
                for(String keyId : devicesInCells.keySet()) {
                        processLayoutPorts(devicesInCells.get(keyId));
                }
```

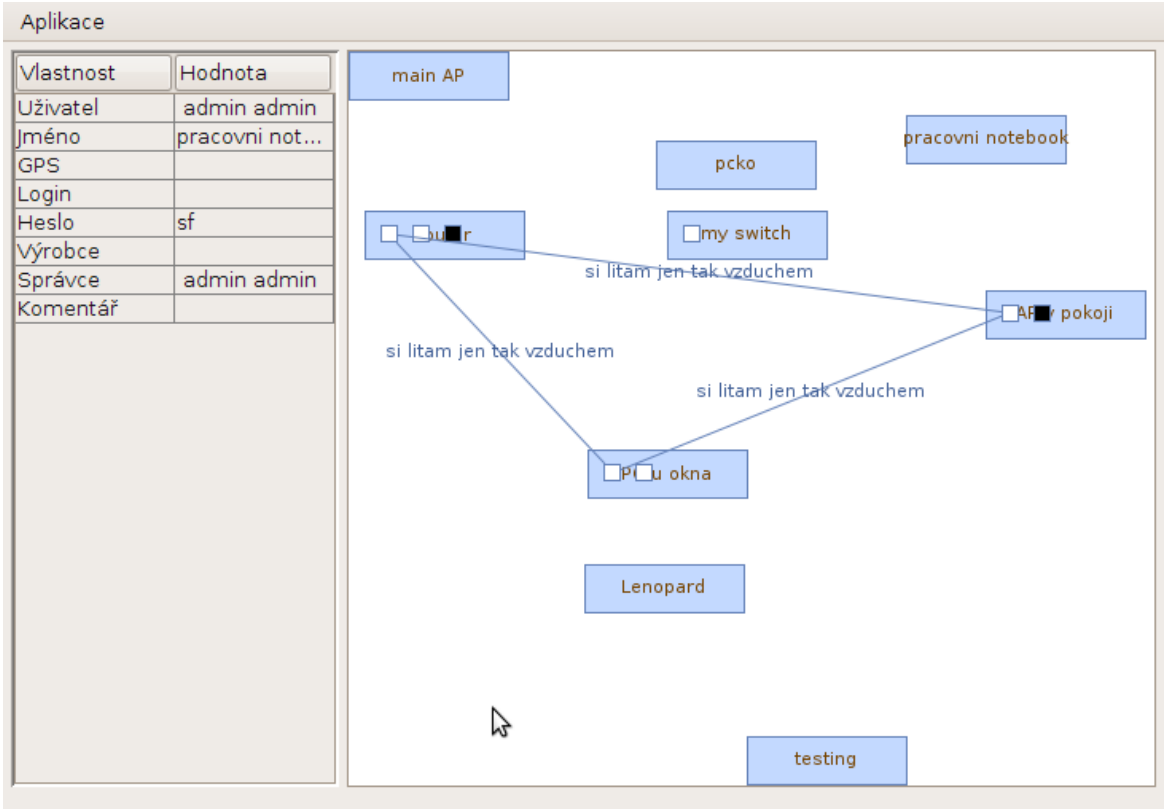
Zde jsou vykreslovány segmenty, které jsou rozděleny do tří skupin:

- P_P – segment mezi dvěma porty
- I_P – segment mezi rozhraním a portem
- I_I – segment mezi dvěma rozhraními

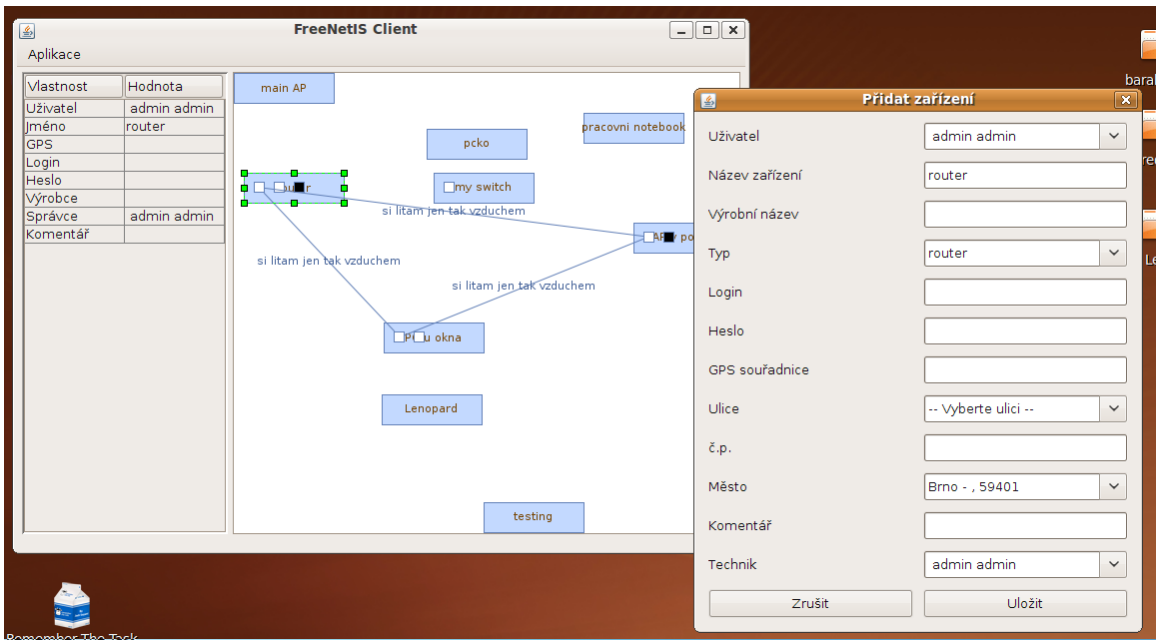
```
                for(Connection c : g.getConnections()) {
                        if (c.getType().equals("P_P")) {
                                insertEdge(parent, null,
segments.get(c.getSegment().toString()), portsInCells.get(c.getPort1().toString()),
portsInCells.get(c.getPort2().toString()));
                        } else if (c.getType().equals("I_P")) {
                                insertEdge(parent, null,
segments.get(c.getSegment().toString()), ifacesInCells.get(c.getPort1().toString()),
portsInCells.get(c.getPort2().toString()));
                        } else if (c.getType().equals("I_I")) {
                                        insertEdge(parent, null,
segments.get(c.getSegment().toString()), ifacesInCells.get(c.getPort1().toString()),
ifacesInCells.get(c.getPort2().toString()));
                        }
                }
        } finally {
                getModel().endUpdate();
        }
}
```

Po dokončení této funkce je celý graf definovaný v databázovém modelu FreeNetISu vykreslen na plátno komponenty EditPanel.

7 VÝSLEDKY



Obr. 11. Základní pracovní plocha klienta, na levé straně informace o posledním vybraném objektu, na pravé straně plátno, kde se kreslí vlastní schéma



Obr. 12. Přidávání zařízení

ZÁVĚR

Práce se zabývá tvorbou jednoduchého desktopového klienta, který umí upravovat schéma počítačové sítě nad již existujícím webovým informačním systémem FreeNetIS. Nejprve jsme museli řešit komunikaci klientské a serverové aplikace. Tu jsme rozebírali v první části práce. Cílem bylo vybrat jednoduché rozhraní, které by se jevilo jako lehce rozšiřitelné. Volba padla na REST a jeho referenční implementaci v JAVě, Jersey API, které splnilo kladené požadavky.

Schéma sítě je z matematického hlediska graf, který se skládá z hran a vrcholů. Proto se přímo nabízelo využití nějakého již funkčního frameworku, který velmi ulehčí kreslení grafů. Z toho důvodu se druhá část této práce zaobírá analýzou grafové knihovny JGraphX.

Následující část se soustředí přímo na implementaci serverového REST rozhraní v PHP a frameworku Kohana.

Poslední fáze práce obsahuje detaily implementace klientské aplikace v jazyce JAVA.

Kódy obou aplikací jsou dostupné na adrese www.bitbucket.org/tomina a jsou šířeny pod licencí GNU/GPL. Aplikace setrvávají stále ve vývoji a měly by směřovat k jasnému cíli, tedy reálnému používání. V této fázi aplikace umí vykreslovat schéma sítě, přidávat zařízení na určitou pozici, přidávat porty, rozhraní a segmenty.

CONCLUSION

This diploma thesis deals with a creation of simple desktop client, which is used to modify scheme of computer network above already existing information system FreeNetIS. At first we have to solve communication between client and server application. We solved this task in the first part of the thesis. The aim was to choose a simple interface that would appear to be easily expandable. The choice fell on REST and its reference implementation in JAVA Jersey API, that meets the imposed requirements.

Network diagram is from mathematical point of view a graph, which consists of edges and vertices. Therefore, there was directly offered to use some already functional framework, which greatly facilitate plotting graphs. Therefore, the second part of this thesis deals with the analysis graph library JGraphX.

The next part is focused on implementation the server interface REST in PHP and framework Kohana.

The final part includes the implementation details a client application in JAVA

Both source codes of application are available at www.bitbucket.org/tomina and are released under GNU/GPL license. Application remains still in development and should aim at obvious goal, it means to real using. In current phase, we can add devices on assigned position, add ports, interfaces and segments.

SEZNAM POUŽITÉ LITERATURY

[1] DANĚK, Petr. *Informační systém pro správu a evidenci uživatelů rozsáhlých sítí.* [s.l.], 2008. 71 s. UTB/UAI. Diplomová práce.

[2] ROZEHNAL, Marek. *Informační systém pro správu a evidenci uživatelů rozsáhlých sítí*. [s.l.], 2008. 72 s. UTB/UAI. Diplomová práce.

[3] JGraph and mxGraph [online]. 2001-2009 [cit. 2010-02-05]. Dostupný z WWW: <http://www.jgraph.com/>.

[4] Kohana: Swift, Secure, and Small PHP 5 Framework [online]. 2007-2010 [cit. 2010-02-05]. Dostupný z WWW: <http://www.kohanaphp.com/>.

[5] CERAMI, Ethan. *Web Services Essentials*. [s.l.] : O'Reilly, 2002. 304 s. ISBN 0-596-00224-6.

[6] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. IRVINE, 2000. 180 s. Dizertační práce. UNIVERSITY OF CALIFORNIA. Dostupné z WWW: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

[7] CHODIL, Ladislav. *Budování REST aplikací v Javě* . Brno, 2007. 72 s. Diplomová práce. Masarykova Univerzita.

[8] JGraph [online]. 2010, 2010-06-01 [cit. 2010-06-04]. *MxGraph User Manual*. Dostupné z WWW: <http://jgraph.com/doc/mxgraph/>.

[9] *RESTful PHP Web Services*. Birmingham : Packt Publishing, 2008. 220 s. ISBN 978-1-847195-52-4.

[10] *The Java Tutorial* [online]. 2010, 2010-01-10 [cit. 2010-05-27]. Dostupné z WWW: <http://java.sun.com/docs/books/tutorial/>.

[11] Sun Microsystems. *Writing Java Application Using Jersey* : INTEROPERATE WITH RESTful WEB SERVICES [online]. [s.l.] : [s.n.], 2010 [cit. 2010-06-02]. Dostupné z WWW: <http://docs.sun.com/app/docs/doc/820-4867?l=en>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

HTTP Hypertext Transfer Protocol.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

REST Resource State Transfer

XML Extensible Markup Language

RPC Remote Procedure Call

SOAP Simple Object Access Protocol

WSDL Web Service Definition Language

JSON Javascript Object Notation

MAC Media Access Control Address

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PI: Disk CD s oběma aplikacemi a diplomovou prací.

PŘÍLOHA P I: DISK CD S OBĚMA APLIKACEMI A DIPLOMOVOU PRACÍ.