VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRACE VYBRANÝCH GRAFOVÝCH ALGORITMŮ

DEMONSTRATION OF SELECTED GRAPH ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **KATARÍNA GALANSKÁ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20410/2017/xgalan02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Galanská Katarína**
Obor: Informační technologie
Téma: **Demonstrace vybraných grafových algoritmů**
Demonstration of Selected Graph Algorithms
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Po konzultaci s vedoucím se seznamte s vybranými grafovými algoritmy (např. toky v síti, barvení grafů, hledání eulerovské kružnice). Dále se seznamte s architekturou aplikace od pana Varadinka z roku 2013.
2. Navrhněte úpravu architektury existující aplikace, aby umožňovala přidání dalších algoritmů a vícejazyčné uživatelské rozhraní. Dle konzultace s vedoucím rozhodněte o způsobu rozšíření existující aplikace nebo vývoji nové.
3. Navrhněte a implementujte didaktickou demonstraci algoritmů vybraných dle pokynů vedoucího na základě návrhu a rozhodnutí z předchozího bodu.
4. Implementaci testujte na minimálně 10 různých úlohách a zhodnoťte její negativa, pozitiva a další možný vývoj.

Literatura:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest: Introduction to Algorithms. McGraw-Hill, 2002
- J. Varadinek: Demonstrace grafových algoritmů, bakalářská práce, Brno, FIT VUT v Brně, 2013
- Materiály k předmětu Grafové algoritmy

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a částečně bod 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S. VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ Fakulta informačních technologií Ústav informačních systémů 612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto práca sa zaoberá úpravou architektúry existujúcej aplikácie pre demonštráciu a vizualizáciu vybraných grafových algoritmov. Cieľom práce je rozdeliť aplikáciu do viacerých modulov a umožniť ľahšiu rozšíriteľnosť. K modularizácií je využívaná technológia OSGi. Vysvetlené sú jej princípy, ktoré sú následne využité k návrhu a implementácií modulov. V rámci novej architektúry je implementovaná podpora viacjazyčnosti programu využitím Eclipse Plugin internacionalizácie. Pri tvorbe modulov grafových algoritmov sú využívané OSGi služby, pomocou ktorých je možné do programu dynamicky pridávať moduly a registrovať ich služby počas behu programu. Implementácia zahŕňa aj tvorbu nových modulov pre Primov, Kruskalov, Edmondsov Karpov a upravený Hierholzerov algoritmus. Beh simulácií nových algoritmov je otestovaný na sade vytvorených grafov.

Abstract

This thesis deals with re-architecture of an existing application for the demonstration and visualization of selected graph algorithms. The goal of this thesis is to convert the application into multiple modules and make it easier to extend. OSGi technology helps to achieve the modularity. Its principles are explained and used to design and implementation of modules. With new architecture is also implemented the support for multilingual user interface using Eclipse Plugin internationalization. Graph algorithms represented as modules use OSGi Services. Dynamicity is ensured by installing the module and registering its service during the program's run time. The implementation also includes the creation of new modules for Prim's, Kruskal's, Edmond Karp's and modified Hierholzer's algorithm.

Kľúčové slová

grafy, grafové algoritmy, MST, Primov algoritmus, Kruskalov algoritmus, Hierholzerov algorithmus, Eulerov ťah, Ford-Fulkersonova metoda, Edmond Karpov algoritmus, OSGi, modularizácia, rozšíriteľnosť, OSGi služby, viacjazyčnosť aplikácie

Keywords

graphs, graph algorithms, MST, Prim's algorithm, Kruskal's algorithm, Hierholzer's algorithm, Euler path, Ford-Fulkerson method, Edmond-Karp algorithm, OSGi, extensibility, OSGi services, multi-language application

Citácia

GALANSKÁ, Katarína. *Demonstrace vybraných grafových algoritmů*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Demonstrace vybraných grafových algoritmů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pána Ing.Zbyňka Křivku,Ph.D. Uviedla som všetky literárne pramene a publikácie, z ktorých som čerpala.

.....
Katarína Galanská
15. mája 2018

Poďakovanie

Chcela by som poďakovať Ing. Zbyňkovi Křivkovi Ph.D. za odborné vedenie a konštruktívne pripomienky, ktoré mi pri tvorbe tejto práce pomohli.

Obsah

Kapitola 1

Úvod

Pri budovaní softvéru je jedným z najvýznamnejších problémov pochopenie veľkých projektov so zložitými väzbami. Vývojár trávi bežne viac času zisťovaním toho, čo kód robí, namiesto toho, aby písal nový. Spôsob, akým je možné sa problému vyhnúť, je dbať na to, aby bola výsledná aplikácia modulárna a rozšíriteľná bez väčšiej námahy pochopenia fungovania celej aplikácie. Modulárna aplikácia sa zvonku stále tvári ako jeden celok, no pri bližšiom nahliadnutí možno identifikovať viacero menších komponentov s určitými závislosťami. Táto bakalárska práca je zameraná na implementáciu práve takejto aplikácie. Konkrétne na úpravu architektúry existujúcej Java aplikácie [11] demonštrujúcej vybrané grafové algoritmy.

Kapitole 2 sa zameriava na analýzu architektúry existujúcej aplikácie. Znázorňuje rozdelenie aplikácie do balíkov tried a popisuje riadiacu jednotku okna simulátora.

V kapitole 3 sú vysvetlené základné princípy potrebné k porozumeniu danej problematiky. Venuje sa tiež základným stavebným kameňom rozšíriteľných aplikácií. Rovnako sú definované aj technológie, ktorými je možno modularitu dosiahnuť, ich porovnanie, výhody aj nevýhody. Vysvetlený je výber najvhodnejšieho spôsobu s ohľadom na budúce využívanie aplikácie.

Kapitola 4 sa zaoberá návrhom novej architektúry a spôsobom rozdelenia existujúcej aplikácie na menšie časti s úmyslom minimalizácie závislosti jednotlivých modulov. Rozdelenie do jednotlivých modulov sa riadi štandardom OSGi. V tejto práci je označovaná najmenšia jednotka modularizácie podľa OSGi štandardu slovom modul (angl. bundle). Ďalším krokom vývoja je návrh spôsobu podpory viacjazyčnosti s dôrazom na zmenu jazyka počas behu aplikácie. Po dosiahnutí modularity sú navrhnuté nové moduly grafových algoritmov, ktoré sú vyberané na základe predmetu Grafové algoritmy (GAL), ktorý je vyučovaný v magisterskom štúdiu na Fakulte informačných technológií VUT v Brne. Program slúži práve ako pomôcka k výuke v tomto predmete.

Kapitola 5 predstavuje samotnú implementáciu navrhnutej aplikácie. Vysvetlené sú základné vzťahy medzi modulmi a novo implementovaná funkcionalita.

Predposledná kapitola 7 je venovaná testovaniu, kde je overená funkčnosť novo implementovaných modulov. Každá nová implementácia simulácie algoritmu je testovaná na viacerých grafoch.

Záverečná kapitola 8 zhŕňa celú prácu. Rekapituluje použité mechanizmy, implementovanú funkcionalitu a zhodnocuje aplikáciu ako celok.

Kapitola 2

Analýza súčasnej aplikácie

V tejto časti sa zameriavam na analýzu existujúcej aplikácie [11], ktorá demonštruje vybrané grafové algoritmy. Ako bolo zmienené v úvode, cieľom práce je aplikáciu modularizovať. Aplikácia využíva platformu Java SE (z anglického Standard Edition) s užívateľským rozhraním využívajúcim štandardizovanú knižnicu Swing. Rozsiahle možnosti prezentácie a vizualizácie grafov poskytuje práve knižnica JGraphX [1]. Je navrhnutá a implementovaná do dvoch celkov. Prvým je editor grafu, ktorý tvorí hlavné okno programu. Jej v ňom možné vytvoriť nový graf pridávaním vrcholov a hrán. Rovnako užívateľovi dáva možnosť načítať graf zo súboru v podobe XML, v ktorom je zapísaná štruktúra a štýly pre vizualizáciu grafu. Uzly a hrany sú reprezentované objektami rovnakého typu obsahujúce príznak určujúci, či sa jedná o hranu alebo vrchol. Grafu je možné nastaviť počiatočný uzol. Pomocou grafické užívateľského rozhrania editoru je možné spustiť druhý celok, ktorým je samotný simulátor užívateľom určeného algoritmu. Panel simulátora pozostáva z troch hlavých častí. Prvou je vizualizačný panel znázorňujúci priebeh algoritmu. Táto komponenta je riadená panelom s pseudo-kódom pozostávajúca z textového panelu s očíslovanými riadkami. Panel s pseudo-kódom navyše ponúka možnosť riadenia simulácie postupným krokovaním, nastavením rýchlosti behu, zastavením, opätovných spustením alebo reštartovaním. Ďalším rozšírením je možnosť určenia bodov prerušenia (angl. breakpoints), ktoré zastavia beh simulácie v užívateľom zvolenej časti algoritimu. Poslednou časťou simulátora je panel s premennými, v ktorom je možné sledovať aktuálny stav premenných počas simulácie.

Naimplementované sú algoritmy prehľadávania do šírky (BFS), prehľadávanie do hĺbky (DFS), algoritmus topologického usporiadania, vyhľadávanie silne súvislých komponentov, Bellman-Fordov algoritmus a Dijkstrov algoritmus pre vyhľadávanie najkratšej cesty. Simulácie jednotlivých algoritmov poskytujú užívateľovi možnosť aktivácie interaktívneho módu, ktorý pomáha pri otestovaní si vlastného pochopenia daného algoritmu. Interaktívny mód dodáva aplikácií edukatívnu funkciu, pričom v rámci simulácie je možné mód viacnásobne zapínať a vypínať.

2.1 Analýza architektúry

Súčasná aplikácia je rozdelená do viacerých balíkov tried. Balík tried *editor* obsahuje triedu rezprezentujúcu graf, upravenú tak, aby bol vhodný pre okno editoru, triedu manažéra pre návrat akcií, pre ukladanie a načítavanie grafu, pre komponentu editoru grafu, informačný panel, zmenu štýlu a vytvorenie dialógu.

Balík tried *program* obsahuje triedy pre okno s informáciami o programe, hlavný panel editoru, dialóg s nápovedou, hlavné menu programu, hlavnú spúšťaciu triedu aplikácie pre simulácie a panel nástrojov. V balíku *program.actions* sa nachádzajú akcie pre spúšťanie simulácie, pre vytvorenie nového grafu, logovanie akcií, a akcie pre ovládanie grafu v editore.

Triedy v balíku *simulator.algorithms* definujú abstraktnú triedu pre algoritmus, ktorý poskytuje základné metódy pre prevod do podoby pre upravený panel pseudo-kódu v simulátore. V tomto balíku sa nachádza zápis presudokódov všetkých implementovaných algoritmov.

Abstraktná trieda pre riadenie simulátora sa vyskytuje v balíku *simulator.controllers* spolu s riadiacimi jednotkami jednotlivých algoritmov.

V balíku *simulator.graph* sa nachádzajú triedy rozhrania pre štýl vizualizácie grafu pre algoritmy, rozhranie pre objekt, ktorý sa vkladá do hrán a uzlov. Obsahuje aj triedu pre graf špečiálne určený pre simulátor algoritmu. Obsahom sú aj balíky tried určené pre jednotlivé algoritmy, obsahujúce operácie na objektami.

Posledný balík *simulator.gui* pozostáva z panelu ovládania simulátora, panelu pre pseudokód, panelu premenných a všeobecného panelu prezentovaného pomocou abstraktnej triedy, ktorá je rozšíriteľná panelom konkrétneho algoritmu. Súčasťou je aj definícia tried pre konkrétne algoritmy.

2.2 Riadiaca jednotka okna simulácie

Riadiaca jednotka je implementovaná pre jednotlivé algoritmy na mieru. Hlavnou súčasťou je páska inštrukcií, ktorá reprezentuje zápis algoritmu. Inštrukcie slúžia k samotnému vykonávaniu daného algoritimu. Vďaka páske inštrukcií môže užívateľ ľahšie ovládať beh simulácie pomocou užívateľského rozhrania. Riadiaca jednotka je schopná ovládať panel premenných, panel s pseudo-kódom algoritmu a jeden či viac panelov určených pre vizualizáciu grafu.

Interaktívny mód Tak ako s bežnými inštrukciami, pracuje riadiaca jednotka aj s interaktívnymi inštrukciami, ktoré sú definované v dvoch stavoch. Pokiaľ je v neaktívnom stave, čaká na aktiváciu riadiacou jednotkou. Po aktivácií využívania daného módu čaká simulácia algoritmu na interakciu užívateľa. Napríklad výber ďalšieho uzlu alebo nastavenie novej vzdialenosti podľa vybraného algoritmu.

Kapitola 3

Prehľad základných princípov a použitých technológií

3.1 Modularita

Modulárna aplikácia je softvér, ktorého implementácia je rozdelená do jednotlivých častí, do takzvaných modulov. Jedná sa o technológiu vývoja softvéru tak, že sa oddeľuje funkčnosť programu do nezávislých, vymeniteľných modulov, ktoré sú spojené za účelom získania konečnej aplikácie. Tým je možné zaistiť prehľadnosť a ľahkú manipuláciu s aplikáciou. Jej moduly majú medzi sebou určité závislosti a jej funkcionalita je rozšíriteľná bez nutnosti kompilácie. Modularita prináša nespočetné výhody medzi ktoré patrí hlavne jednoduchá rozšíriteľnosť a ľahká implementácia nových funkcií. Niektoré moduly môžu byť definované štandardnými metódami, ktoré sú znovupoužiteľné v iných programoch. Rozsah premenných sa dá jednoducho kontrolovať. Patrí sem aj výhoda jednoduchšieho pochopenia, pretože každý modul funguje nezávisle na inom module. Pri vývoji modulárnej aplikácie môže pracovať viacero programátorov súčasne a vďaka využívaniu rôznych modulov často stačí napísať menej kódu.

3.2 Možné riešenia modularizácie

Modularizácia zdrojového kódu aplikácií je dôležitou súčasťou vývoja softvéru. Pri návrhu je potrebný výber správnej technológie. V tejto podkapitole sú predstavené možnosti implementácie rozšíriteľnej aplikácie. Vysvetlené sú základné princípy a porovnané možné spôsoby dosiahnutia modularity.

3.2.1 Service Provider Interface

Jedným z riešení modularizácie od firmy Oracle je využitie rozhrania Service Provider Interface (SPI) [6]. Jedná sa o množinu verejných rozhraní a abstraktných tried. Definuje triedy a metódy, ktoré sú dostupné v aplikácií. SPI môže byť reprezentované rozhraním, abstraktnou triedou alebo súborom rozhraní či abstraktných tried. Tie sú určené pre implementáciu alebo rozšírenie treťou stranou. SPI je teda používané na umožnenie rozšírení a vymeniteľných komponentov. Trieda Service Provider implementuje SPI a umožňuje pridávanie služieb bez zmeny pôvodnej aplikácie a je dostupná v platforme Java SE 8.

3.2.2 Trieda ServiceLoader a limity jej využitia

Trieda ServiceLoader [7] pomáha nájsť, načítať a používať nových poskytovateľov služieb. Ich inštalácia funguje na princípe pridania nového java JAR do classpath aplikácie alebo do adresára rozšírení. Service Loader načíta rozšírenia a povolí aplikácií použiť poskytovateľove API (skratka pro Application Programming Interface). API je zbierka funkcií a tried, ale aj iných programov, ktoré určujú, akým spôsobom sa majú funkcie knižníc volať zo zdrojového kódu programu. Sú to programové celky, ktoré programátor volá namiesto vlastného naprogramovania.

Pri pridaní nového poskytovateľa do classpath, parametru v JVM, Java kompilátoru alebo do adresára rozšírenia ho ServiceLoader nájde. Trieda je finálna, čo znamená, že nie je možné vytvárať podtriedy alebo prepisovať načítavací algoritmus. Teda nie je možné načítavať služby z iných priečinkov. API triedy ServiceLoader sú užitočné, no majú svoje obmedzenia. Súčasná trieda *ServiceLoader* nedokáže oznámiť aplikácií skutočnosť, že je dostupný nový poskytovateľ počas behu programu.

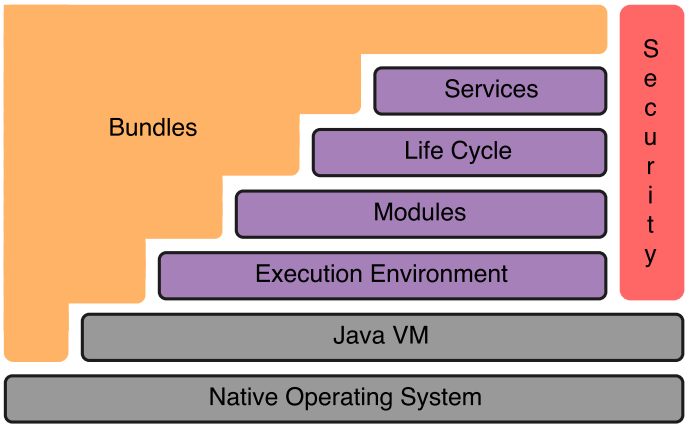
3.2.3 Open Service Gateway Initiative

Open Service Gateway Initiative (OSGi) [10, 4] je súbor špecifikácií definujúcich dynamický systém v programovacom jazyku Java. Tieto špecifikácie umožňujú vývoj aplikácií, ktoré sú zložené z rôznych komponentov. OSGi je vrstevný model [2], ktorý poskytuje niekoľko konceptov.

Praktickou výhodou OSGi je, že každá softvérová súčasť môže definovať API prostredníctvom množiny exportovaných modulov. Každá zložka môže špecifikovať jej požadované závislosti. Špecifikácie OSGi umožňujú jednotlivým komponentom skryť svoju implementáciu pred inými komponentami počas ich komunikácie prostredníctvom služieb, ktoré sú zdielané objektami medzi komponentami.

3.3 OSGi

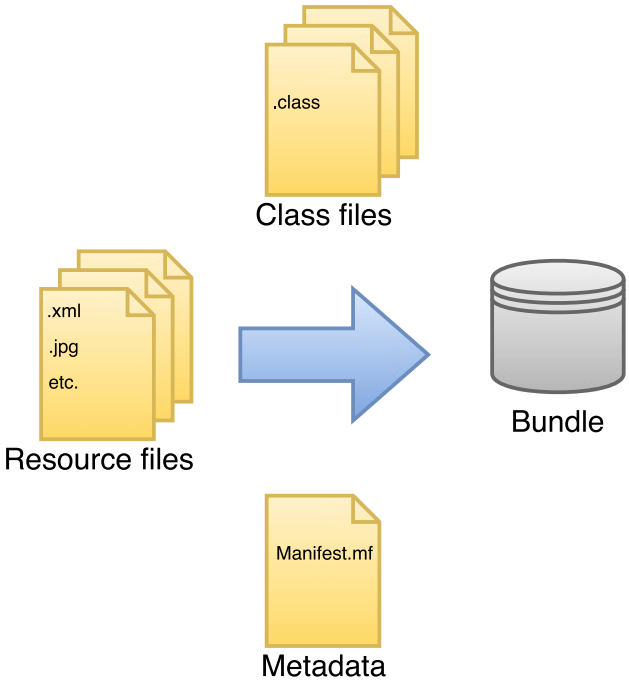
V tejto sekcií je popísaná OSGi architektúra. Funkčnosť aplikačného rámca je rozdelená do niekoľkých vrstiev, ktoré su zobrazené za obrázku 3.1



Obr. 3.1: Architektúra OSGi

Prvou vrstvou je `Behové prostredie` (Execution Environment), ktoré poskytuje nezávislosť od operačného systému, čo znamená, že tie isté OSGi kompatibilné služby môžu byť spravované na akejkoľvek platforme podporujúcej Javu. Behové prostredie definuje metódy a triedy dostupné v danej platforme. Vrstva s názvom Security sa stará o bežpečnosť odmedzovaním funkcionality modulov na preddefinované možnosti.

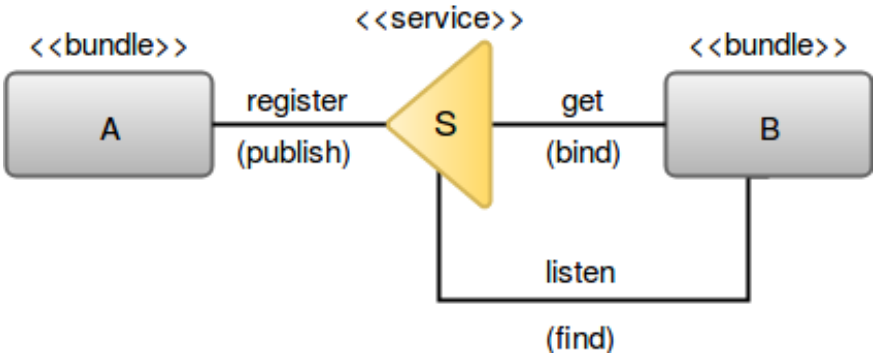
Vrstva `Modulov` (Modules) definuje základný koncept OSGi modulu tzv. bundle. Obsahuje zdrojové súbory, zdroje k nim pripojené a metadáta v súbore Manifest.mf viď obrázok 3.2. Ich kombináciou vznikne aplikácia. Vďaka možnosti definovať, ktoré balíčky tried modulov budú viditeľné atribútom sú efektívnejšie ako klasické JAR archívy. OSGi skryje všetko v súbore JAR, až pokiaľ nie je explicitne exportovaný. Modul, ktorý chce použiť iný JAR, musí explicitne importovať časti, ktoré potrebuje. V štandardnom nastavení nie je povolené žiadne zdieľanie. Ďalšou výhodou je možnosť definovať závislosť na iných moduloch. Výhodou je možnosť automatickej verifikácie konzistentnosti OSGi aplikačným rámcom. Tento proces sa nazýva *bundle resolution*.



Obr. 3.2: Štruktúra modulu

Vrstva `Služieb` (Services) prepája moduly dynamickou cestou. Služba môže byť zaregistrovaná kedykoľvek je modul v stave *starting* alebo *active*. Registrácia sa uskutočňuje pod jedným alebo viacerými rozhraniami, ktoré daný objekt implementuje. Registrácia môže byť ľubovoľného počtu atribútov kľúč-hodnota pre odlíšenie služieb rovnakého typu. Po registrácií, ako je možné vidieť na obrázku 3.3, môže byť služba nájdená ostatnými modulmi v registri na základe názvu rozhrania. Modul môže zaregistrovať poslucháča *ServiceListener* na službu, ktorá ešte nie je dostupná a čakať na jej registráciu. Pri inštalácií nových modulov môžu prichádzať nové služby. Rovnako môžu odchádzať a to programovo alebo automaticky pri zastavení modulu, ktorý bol registrovaný.

Modul A môže poskytnúť implementáciu služby S a iný modul B ho môže spotrebovať. To sa dosiahne nájdením služby a väzbou na ňu v okamihu nájdenia. Súčasne môžu byť implementované viaceré služby pre rovnaký typ. Prichádzajú a odchádzajú počas behu pričom spotrebiteľ je stále schopný reagovať. Je to implementované službou OSGi *ServiceRegistry*.



Obr. 3.3: OSGi Služby

3.3.1 OSGi metadáta

Moduly OSGi sú technicky kompresné súbory .jar s dodatočnými meta informáciami [9], ktoré slúžia na popis modularity a presnú špecifikáciu. Sú uložené v súbore Manifest.mf. Súbor nazývame *manifest* a je časťou štandardu Java špecifikácie. OSGi pridáva súboru ďalšie metadáta. Vďaka nim dokáže aplikačný rámec rozpoznať závislosti daného modulu. Definuje API daného modulu.

`Nasledujúci zoznam definuje štruktúru metadát modulov:`

- Čitateľné informácie
- Identifikácia zväzkov
- Viditeľnosť kódu

Nasledujúca tabuľka 3.1 vysvetľuje identifikátory v manifeste. *Bundle-Name* symbolizuje krátky popis modulu. *Bundle-SymbolicName* je unikátny identifikátor modulu. Ak daný modul používa funkciu rozšírenia Eclipse, musí byť označený ako *singleton* pridaním údaju *singleton:=true*. *Bundle-Version* definuje verziu modulu. *Bundle-Activator* špecifikuje nepovinnú triedu aktivátoru, ktorá implementuje rozhranie *BundleActivator*. Inštancia triedy je vytvorená v momente, kedy je modul aktivovaný. Aktivátor OSGi môže byť použitý na konfiguráciu modulu počas štartu. Jeho vykonanie však zvyšuje čas spustenia aplikácie.

3.3.2 Čitateľné informácie

Tieto informácie nie sú spracované aplikačným rámcom. Slúžia ako pomocné informácie pre človeka. Špecifikácia definuje niekoľko informácií, použitých pre tento účel a to názov a popis modulu. Atribút *Bundle-SymbolicName* slúži na jednoznačnú identifikáciu daného modulu v rámci OSGi. Rovnako ako názov sa identifikáciu využíva aj verzia modulu určená atribútom *Bundle-Version* a *Bundle-ManifestVerion*. Definované tri atribúty jednoznačne určujú konkrétny modul.

3.3.3 Viditeľnosť kódu

Viditeľnosť kódu špecifikuje konkrétne moduly viditeľné z každého modulu, ktoré potrebuje importovať a tie ktoré budú obsiahnuté. OSGi rámec využíva špecifický prístup zvaný *classpath modulu*. Ide o zoznam adresárov, v ktorých rámec hľadá nové moduly. Rozdiel od klasickej classpath v Jave je v tom, že classpath modulu odkazuje na umiestnenie vo vnútri JAR archívu v samotnom module. Pre definovanie zoznamu adresárov sa využíva atribút *Bundle-ClassPath*. Modul môže mať množstvo požiadaviek a možností, ktoré sú vyjadrené v manifest súbore. Pre poskytnutie prístupu k triedam daného modulu musí modul exportovať balík obsahujúci dané triedy. Exportovaných balíkov môže byť ľubovoľný počet a definujeme ich pomocou atribútu *Export-Package*. Balíky, ktoré nie sú exportované sú považované za súkromné a nie sú viditeľné ostatným modulom. V prípade závislosti modulu na iných je potrebné špecifikovať závislosť pomocou atribútu *Import-Package*.

Tabuľka 3.1: Súbor MANIFEST.mf

Manifest-Version:	1.0
Bundle-ManifestVersion:	2
Bundle-Name:	euler
Bundle-SymbolicName:	euler
Bundle-RequiredExecutionEnvironment:	JavaSE-1.8
Bundle-Version:	1.0.0.qualifier
Import-Package:	algorithm.service, simulator.graphs
Export-Package:	simulator.gui.panels.euler
Bundle-Activator:	algorithm.Activator

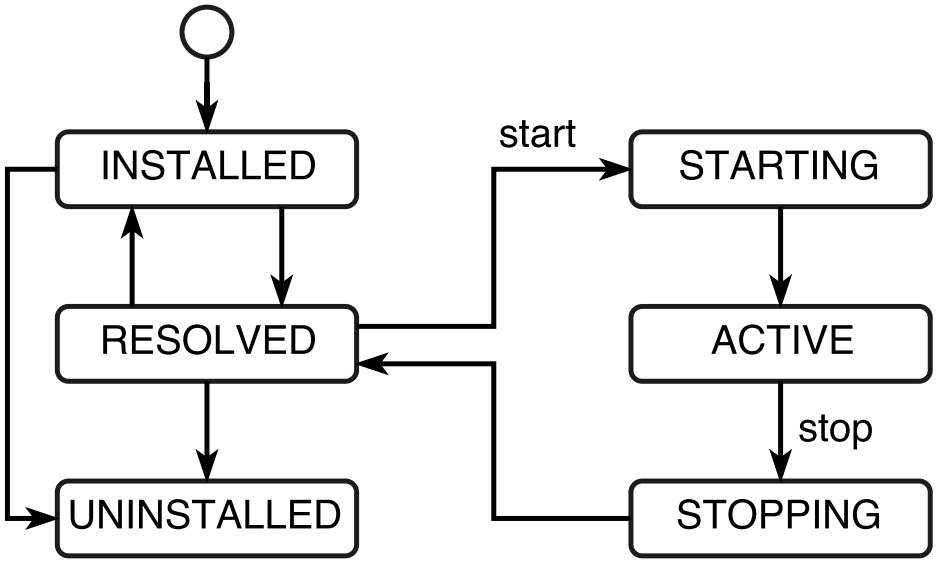
3.4 OSGi modul a jeho životný cyklus

Modul OSGi poskytuje jasnú definíciu modulárnosti. Zahŕňa to jeho identitu, požiadavky a možnosti. Hlavičky manifest súborov sa starajú o definovanie identity. OSGi špecifikácia definuje modul ako najmenšiu jednotku modularizácie. V prostredí Eclipse je táto softvérová komponenta nazývaná *plug-in*, tieto pojmy sú však zameniteľné. Jedná sa o samostatnú jednotku, ktorá explicitne definuje jej závislosti na iných moduloch a službách. Rovnako definuje API prostredníctvom balíkov Java.

Pri inštalácií modulu počas behu aplikácie OSGi pretrváva modul v lokálnej vyrovnávacej pamäti. Ak sa vyriešia všetky požadované závislosti, potom prechádza do stavu *resolved*. V prípade nevyriešených závislostí zostáva v stave *installed*. V prípade, že existuje niekoľko modulov, ktoré môžu vyriešiť závislosť, použije sa modul s najvyššou validnou verziou. Ak sú verzie rovnaké, použije sa modul s najmenším unikátnym identifikátorom. Každému je počas inštalácie pridelené unikátne identifikačné číslo. Pri spúšťaní modulu prechádza stavom *started* a po úspešnom štarte prechádza do stavu *active*. Tento životný cyklus je znázornený na obrázku 3.4.

Modul je v stave `installed` ak bol nainštalovaný do kontajneru OSGi, jeho závislosti však ešte neboli splnené. Modul vyžaduje iné moduly, ktoré neboli exportované žiadnym aktuálne inštalovaným modulom. Ak bol modul nainštalovaný a systém OSGi pripojil všetky závislosti na úrovni triedy a uistil sa, že sú všetky vyriešené a je pripravený na spustenie. Ak sa spustí a všetky závislosti sú splnené, prejde na stav `resolved`. Dočasný stav `starting`, definuje stav,ktorým modul prechádza počas spustenia modulu po vyriešení všet-

kých závislostí. Spring kontroluje konfiguráciu a vytvára kontext. Stav `active` nastáva keď Spring podá kontext modulu. Dočasným stavom `stopping` modul prechádza počas zastavovania modulu. Ak bol modul Modul zmazaný z OSGi kontajneru prechádza do stavu `uninstalled`. Celý životný cyklus znázorňuje obrázok 3.4



Obr. 3.4: OSGi Bundles

3.5 Služby v OSGi

OSGi služba je inštancia objektu Java, registrovaná v prostredí OSGi s určitou sadou vlastností. Akýkoľvek Java objekt môže byť registrovaný ako služba väčšinou implementujúca známe rozhranie. Každý modul môže registrovať ľubovoľný počet služieb. A rovnako môže využívať nula alebo viac služieb. V štandarde neexistuje limit určujúci ich počet.

3.6 Equinox

Využitie aplikačného rámca OSGi je možné pomocou viacerých implementácií. Equinox [5] patrí medzi jednu z nich. Implementácia tohto OSGi kontaineru tvorí základy platformy Eclipse RCP a IDE. Je implementáciou základnej špecifikácie rámca OSGi, súboru balíkov, ktoré implementujú voliteľné služby OSGi a ďalšiu infraštruktúru na prevádzku systémov založených na OSGi. Základná implementácia rámca Equinox OSGi sa používa ako referenčná a ako taká implementuje všetky požadované funkcie najnovšej špecifikácie základného rámca OSGi. Ak chceme spúšťať Equinox samostatne, je potrebné stiahnuť súbor JAR org.eclipse.osgi. V novšej verzií je potreba viacerých JAR súborov.Tými sú:

- org.eclipse.osgi
- org.apache.felix.gogo.runtime
- org.apache.felix.gogo.command

- org.apache.felix.gogo.shell
- org.eclipse.equinox.console

V prípade prítomnosti týchto súborov v priečinku sa Equinox spustí príkazom:

```
java -jar org.eclipse.osgi.jar -console
```

Po spustení sa užívateľovi zobrazí konzola OSGi, ktorá čaká na zadávanie príkazov. Medzi príkazy, ktoré sú potrebné patria príkaz *ss* zistíme aktuálny stav všetkých nainštalovaných modulov. Pomocou príkazu *install* nainštalujeme modul z danej URL. Po inštalácií modulu sa mu pridelí číselný identifikátor, ktorý je možno použiť ako parameter pri práci s príkazom *start* . Tým sa modul spustí svoje chovanie. Nainštalovaný modul je možno príkazom *stop* zastaviť. Dôsledkom príkazu sa balík následne dostane do stavu resolved. Príkaz *uninstall* úplne odinštaluje modul z OSGi konzoly. Príkazom *diag* môžeme detegovať problémy určitého modulu. Príkazom *exit* zastavíme beh modulov a opustíme konzolu.

```
ss
diag <názov modulu alebo jeho číselný identifikátor>
start <názov modulu alebo jeho číselný identifikátor>
stop <názov modulu alebo jeho číselný identifikátor>
install <bundle URL>
uninstall <bundle URL>
exit
```

3.6.1 Konfigurácia implementácie Equinox

V prípade spustenia aplikácie pomocou implementácie Equinox ako samostatnú aplikáciu je potrebné vytvoriť konfiguračný súbor uložený v priečinku *configuration* nasledovným spôsobom:

```
application/
  configuration/
    config.ini
  org.eclipse.osgi.jar
  module1.jar
  module2.jar
```

Ak sa nachádzajú v priečinku vlastné moduly, pri spustení konzoly predošlím spôsobom zostanú moduly nenainštalované. Je možné ich manuálne nainštalovať alebo upraviť konfiguračný súbor nasledovne:

```
osgi.bundles=./org.apache.felix.gogo.command@start,\
./org.apache.felix.gogo.runtime@start,\
./org.apache.felix.gogo.shell@start,\
./org.eclipse.equinox.console@start,\
./module1.jar@start,\
./module2.jar@start,\
eclipse.ignoreApp=true
```

Týmto spôsobom sa pri každom spustení konzoly automaticky nainštalujú a spustia aj moduly *module*1, *module*2.

Kapitola 4

Návrh

Z predchádzajúcej kapitoly možno získať dostatočný prehľad o OSGi technológii. V tejto kapitole sa budem venovať samotnému návrhu aplikácie. Priblížim detailnejší cieľ práce. Rozoberiem jednotlivé kroky návrhu aplikácie a postup práce. Predstavím svoj návrh riešenia problému. Samotný návrh aplikácie je rozdelený do viacerých časti.

4.1 Rozdelenie aplikácie na OSGi zväzky

Pri modularizácií aplikácie neexistuje konkrétna správna odpoveď ako presne aplikáciu rozdeliť na viacero modulov. Modulová súdržnosť však zohráva dôležitú rolu pri navrhovaní rozdelenia modulov. Modul s príliš malým správaním nepostačuje na to aby bol užitočné pre ostatné moduly, ktoré ho používajú a preto poskytuje minimálnu hodnotu. Naopak modul, ktorý robí príliš veľa, je ťažké opätovne použiť, pretože poskytuje väčšie správanie, než iné moduly potrebujú. Preto je veľmi dôležité identifikovať správnu úroveň granularity. Pri prenášaní existujúcej aplikácie na novú platformu najskôr vznikne jeden veľký modul. Vo zvyku je deliť aplikáciu na modul grafického užívateľského rozhrania aplikácie, dátový modul a logický modul. V prípade tejto práce je potreba klásť dôraz hlavne na závislosti medzi samotnou implementáciou simulácie algoritmu, podporujúcimi funkciami okna simulátora a editorom grafu. Je nutné vytvoriť rozhranie, ktoré definuje všetko potrebné pre simuláciu daného algoritmu. Rovnako je potrebné vytvoriť samostatné moduly pre implementáciu daného rozhrania. Vďaka nemu bude aplikácia poskytovať užívateľovi viacero implementácií.

4.1.1 Odstránenie závislostí

Ak je aplikácia správne navrhnutá, je jednoduché vytvárať nové moduly a rozširovať funkcionalitu. Okrem zníženia väzieb v návrhu aplikácie, poskytuje aj väčšiu flexibilitu. Metóda alebo trieda definujúca príliš veľa funkcionality spôsobuje nežiadúce závislosti. Odstránenie týchto závislostí je možné oddelením implementácie na dva rôzne moduly. Tento problém sa však netýka len konceptu závislostí, ale aj základným princípom objektovo orientovaného návrhu.

Pre správne logické rozdelenie do OSGi modulov je potrebné v aplikácií identifikovať kľúčové časti. Z analýzy existujúcej aplikácie vyplynulo ako vhodné rozdeliť aplikáciu na viacero nasledovných celkov. Balík tried pre editor grafu bude súčasťou modulu `program` spolu s hlavnou časťou užívateľského rozhrania a triedami pre rôzne akcie v editore a v hlavnom menu programu. Triedy pre generovanie základných objektov, ktoré sa budú vkladať

do grafu bude definovaný v module `simulator.graphs`. Modul `simulator.gui` bude obsahovať triedy pre spustenie simulácie, panel ovládania, jednotlivé panely okna simulácie. Pre zapezpečenie dynamického pridávania modulov, je nutné pridať modul rozhrania služby, pomocou ktorého sa budú pripájať jednotlivé implementácie simulácií. Pre definíciu rozhrania pre panely jednotlivých grafových algoritmov z iných modulov je vytvorený modul *algorithm.service*. Obsahom `simulator.controller` bude rozhranie riadiacej jednotka simulátora, v ktorom sa nachádza implementácia základných inštrukcií. Ďalším modulom je `simulator.controlSler`, ktorý okrem iných obsahuje aj abstraktnú trieda pre poskytovanie metódy na prevod pseudo-kódu algoritmu do podoby vhodnej pre upravený panel pseudo-kódu.

Samostatný modul je vytvorený rovnako aj pre každú implementáciu grafového algoritmu. Pre prehľadávanie do hĺbky je definovaný modul `bfs`, pre topologické usporiadanie *toposort*, algoritmus na hľadanie silne súvislých komponentov je obsahom modulu `scc`, Dijkstrov algoritmus pre nájdenie najkratšej cesty v grafe je obsiahnutý v module `dijkstra`, hľadanie najkratšej cesty v podobe Bellman Fordovho algoritmu je zahrnutý v module `bellmanford` a posledný implementovaný algoritmus prehľadávania do hĺbky je obsahom modulu `dfs`.

4.2 Nové moduly

Okrem vytvorenia modulov pre už implementované algoritmy je nutné naimplementovať triedy, ktoré reprezentujú nové implementácií nových vybraných algoritmov.Vybrané algoritmy sú Primov algoritmus a Kruskalov algoritmus, ktoré patria do algoritmov pre hľadanie najkratšej cesty v grafe.Algoritmus pre nájdenie Eulerovho ťahu a Edmond Karpov algoritmus pre vyhľadávanie maximálneho toku v sieti.

Modul implementácie vybraného algoritmu musí na prvom mieste poskytovať služby OSGi, ktorú si bude môcť rozhranie definované v *algorithm.service* registrovať. Okrem toho musí poskytovať komponenty pre okno simulátora, ktorými sú panel s grafom, panel premnenných zobrazujúcich stav simulácie, panel vizualizujúci pseudo-kód daného algoritmu a panel slúžiaci na ovládanie simulácie.

4.2.1 Algoritmus pre nájdenie Eulerovho ťahu v neorientovanom grafe

Podstata algoritmu spočíva v nájdenie cesty prechodu grafom s navštívením každého vrcholu práve jeden krát. Pre nájdenie Eulerovej cesty v grafe s neorientovanými hranami musia byť splnené určité podmienky. Všetky vrcholy s nenulovým stupňom musia byť prepojené.Všetky vrcholy musia mať párny počet hrán alebo môžu existovať maximálne dva vrcholy s nepárnym počtom hrán. V prípade výskytu vrcholu s nepárnym stupňom existuje v grafe Eulerova cesta len z tohto vrcholu.

4.2.2 Primov algoritmus

Algoritmus hľadá minimálnu kostru grafu. Pracuje s neorientovaným grafom. pseudo-kód algoritmu je prevzatý z učebných materiálov predmetu GAL. Algoritmus začína počiatočným vrcholom a postupne pridáva ďalšie susedné uzly, čím zväčšuje veľkosť stromu dovtedy, kým neobsahuje strom nepokrýva všetky vrcholy.

4.2.3 Kruskalov algoritmus

Rovnako ako Primov algoritmus, aj Kruskalov hľadá minimálnu kostru grafu. Tiež pracuje s neorientovaným grafom. Cestu hľadá postupným vyberaním hrán s najmenšou váhou. Hrana je vybraná len v prípade že sa označením hrany nevytvorí v grafe kružnica. Algoritmus sa končí v prípade, že sú navštívené všetky uzly. pseudo-kód algoritmu je rovnako ako Primov prebratý z učebných materiálov predmetu GAL.

4.2.4 Edmond Karpov algoritmus

Algoritmus implementujúci Ford Fulkersonovu metódu hľadá maximálny tok grafom. Špecifickou vlastnosťou tejto metódy je, že je definované poradie výberu cesty. V prípade existencie väčšieho počtu ciest od počiatočného ku koncovému uzlu, je vybraná najkratšia možná. To je zabezpečené algoritmom prehľadávania do šírky (BFS), ktorý sa volá na začiatku každého cyklu. V prípade, že BFS nenájde cestu od počiatočného uzlu ku koncovému, je algoritmus ukončený.

4.3 Viacjazyčnosť aplikácie

Java aplikácie sú typicky prekladané prostredníctvom *property* súborov. Takýto súbor obsahuje kľúče a hodnoty. Zdrojové balíky sú podľa špecifikácie Javy kódované v LATIN-1 (ISSO 8859-1) [12]. Hodnota na ľavej strane predstavuje kľúč, ktorý sa používa v programe. Preto by sa nemal meniť. Mení sa len jeho hodnota na pravej strane. Nasledujúca časť demonštruje príklad obsahu daného súboru.

```
# Bundle.properties file
mainTitle=Simul\u00e1tor Graf\u016f
file=Soubor
help=N\u00e1pov\u011bda
edit=Editace
print=Tisk
pageSetup=N\u00e1hled Tisku
exit=Ukon\u010dit
```

Je potrebné si najprv vytvoriť východzí súbor vlastností, ktorý bude použitý v prípade, že nebude programom stanovené inak.

Prekladové súbory začínajú základným názvom *ResourceBundle* [8] a končia príponou *.properies* a sú definované pre rôzne lokalizácie.

```
messages.properties: standardny subor
messages_cz.properties: preklad do češtiny
messages_en_US.properties: standardny pre angličtinu
```

Pre kompletnosnosť viacjazyčnosť sú vytvorené objekty *Locale* pre všetky podporované jazyky. Objekty by mali zodpovedať prekladovým súborom. Napríklad objekt lokalizácie *Locale.ENGLISH* zodpovedá súboru *Bundle_en.properties*. S pridávaním podpory pre nový jazyk je potreba vytvoriť nový súbor vlastností spolu s novým objektom lokalizácie.

```
Locale[] locales = {
    Locale.EN,
```

```
    Locale.CS,
};
```

Objekt ResourceBundle pomáha s manipuláciou s viacerými lokalizáciami naraz. Ak program potrebuje špecifický zdroj, môže ho načítať z balíka zdrojov,

```
ResourceBundle resources =
     ResourceBundle.getBundle("bundle", currentLocale);
```

Metóda getBundle vyhľadá súbor, ktorý zodpovedá základnému názvu "bundle". Hodnotu zo súboru získame pomocou funkcie getString

```
resources.getString("file");
```

Keďže rôzne preklady musia byť k dispozícií viacerým modulom, je kvôli závislostiam vytvorený nový modul, zabezpečujúci preklad aplikácie.

Štandardné umiestnenie, kde Eclipse hľadá možné preklady je OSGI-INF/l10n/bundle. Pomocou atribútu *Bundle-Localization* v manifest súbore možno špecifikovať alternatívnu destináciu prekladových súborov takto:

```
Manifest-Version: 1.0
Bundle-SymbolicName: gal.demo.priklad; singleton=true
Bundle-ManifestVersion: 2
Bundle-Version: 6.0
Bundle-Name: I18n
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Require-Bundle: gal.demo.program,
 org.eclipse.core.runtime
Bundle-Localization: OSGI-INF/anotherlocation/bundle
```

Kapitola 5

Implementácia

V tejto kapitole je znázornený vývoj jednotlivých modulov využívaním OSGi technológiu. K tvorbe modulov aplikácie som využívala Eclipse Java EE IDE veziu Oxygen.8.Release (verzia 4.7.2). Vďaka zabudovanému kontajneru Equinox ponúka Eclipe IDE vynikajúcu podporu pre vývoj nových modulov. Kontajner je rovnako možno použiť k spúšťaniu a ladeniu modulov.

Implementácia celej aplikácie je rozložená do viacerých častí. Prvým bodom je tvorba jednotlivých komponentov aplikácie a ohľadom na minimalizáciu závislostí medzi nimi. Ďalší bod tejto kapitoly sa venuje prekladu aplikácie do anglického jazyka, ktorý možno prepínať dynamicky počas behu programu. Predposledný bod opisuje vytvorenie rozhrania pre služby jednotlivých algoritmov poskytujúich API novému rozhraniu. Záverom kapitoly sú novo implementované moduly v podobe simulácie nových grafových algoritmov

5.1 Rozdelenie na menšie moduly

Po zoznámení sa s architektúrou existujúcej aplikácie a naštudovaní si technológie sa zoznámime s architektúrou novej aplikácie. Postupne budú predstavené všetky moduly a vysvetlená ich funkcionalita v aplikácií.

5.1.1 Hlavný modul programu

Modul `program` obsahuje balíky tried pre ovládanie editoru a samotného programu. Do tohoto modulu sú doimplementované metódy na podporu viacjazyčnoti aplikácie.

Aktivátor Do modulu vytvárajúceho základ celej aplikácie je nutné implementovať triedu *Activator* z balíku tried $org.osgi.framework.BundleActivator$. Trieda obsahuje metódu *start*, ktorú volá kontajner pri aktivácií modulu za účelom príležitosti vytvorenia nového okna editoru. Okrem toho obsahuje aj metódu *stop*, ktorú volá kontajner pri zastavovaní modulu. Metóda zatvára hlavné okno programu. Aby mohli byť tieto metódy volané kontajnerom, musia byť definované v súbore MANIFEST.mf nasledovne:

```
Bundle-Activator: program.Activator
```

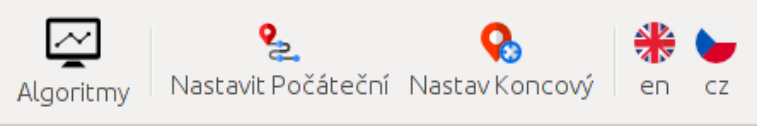
Súčasťou tohto modulu sú základné akcie nad oknom grafového editoru implementované v triede $BasicActions.java$. V nemodulárnej implementácií aplikácie existuje jedna samostatná akcia na spustenie každej simulácie algoritmu. Napríklad pre spustenie algoritmu Tolopogického usporiadania je definovaná metóda $TopologicalSortSimulatorAction$. V novej

implementácií je potrebné pre udržanie modularity a dynamickosti definovať jednu triedu pre volanie všetkých simulácií. Preto vznikla nová metóda *AlgorithmSimulatorAction*. Rozšíriteľnosť novými modulmi som dosiahla využitím OSGi služieb, ktoré sú kontrastne objektami Javy. Služby nie sú automaticky registované. Ak chceme službu registrovať, musí modul najprv vytvoriť objekt a následne volať rozhranie OSGi API.

Existuje viacero spôsobov ako vyhľadať modul a používať službu iného modulu. OSGi služba je implicitne dynamická. Služby môžu kedykoľvek prichádzať a odchádzať, čo môže predstavovať problém, ktorý možno vyriešiť opakovaným vyhľadávaním služby vždy, keď k nej potrebujeme pristúpiť. Tento problém je vyriešený využitím balíka *org.osgi.util.tracker.ServiceTracker* [3]. Jedná sa o triedu nástrojov, ktorá zjednodušuje používanie služieb z registru služieb rámca.

Pri registrácií služieb a prístupu k službám interaguje OSGi cez inštanciu rozhraní *BundleContext* [3]. Toto rozhranie je jediným spôsobom, akým môžu moduly počas behu programu spolupracovať.

Pri vytváraní hlavného modulu sú pridané nové funkcionality. Panel nástrojov bol upravený tak, aby vyhovoval tvorbe grafov pre všetky algoritmy. Bola pridaná akcia na označenie koncového uzlu grafu. V grafickom užívateľskom rozhraní reprezentuje túto akciu tlačidlo *Vyber koncový uzel* alebo anglicky *Set Sink*. Panel nástrojov obsahuje aj tlačidlá nastavujúce jazyk aplikácie. Každé z nich je reprezentované ikonou daného jazyka.



Obr. 5.1: Pridanie možností do panela nástrojov

V hlavnom panely editoru sa nachádza tlačidlo *Algorithmy*. Stlačením sa načítajú dostupné aktívne služby implementovaných algoritmov. Aby si mohol užívateľ vybrať, ktorú implementáciu služby chce simulovať, ich názvy sa zobrazia vo vyskakovaciom menu. Po vybratí kliknutím na názov sa spustí simulácia vybraného algoritmu.

```
Object[] readers = null;
tracker = new ServiceTracker(FrameworkUtil.getBundle(this.getClass())
                             .getBundleContext(),
                             AlgorithmService.class.getName(),
                             null);
serviceTracker.open();
readers = serviceTracker.getServices();
for (int i = 0; i < readers.length; i++) {
       IAlgorithmService lrs = (IAlgorithmService) readers[i];
       algorithmNames.add(lrs.getName());
}
serviceTracker.close();
```

5.1.2 Modul pre zmenu jazyka

Pre zmenu jazyka počas behu programu je vytvorený samostatný modul *program.locales*, ktorý obsahuje prekladové súbory a metódy na ich prehľadávanie. Ak užívateľ zmení lokalizáciu, modul začína automaticky vyhľadávať hodnoty ku kľúčom v súbore danej lokalizácie.

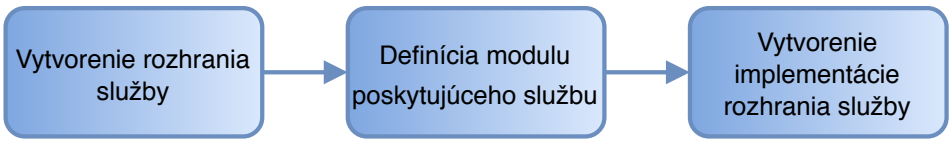
Na vytvorenie grafického užívateľského rozhrania sa v aplikácií používa knižnica Swing, ktorá žiaľ neobsahuje vstavané funkcie pre podporu prepínania jazyka v dobe nehu programu. GUI aplikácie sa skladá z mnohých komponentov UI, ako sú štítky, tlačidlá, ponuky a podobne. Každá z týchto komponentov musí zobrazovať istý text, aby bol komponent užitočný. Pri zmene aktuálnej lokalizácie sa texty Swing komponentov nemenia. Preto je potreba prispôsobiť implementáciu a pri zmene automaticky volať zvlášť funkcie na prepisovanie textov všetkých komponentov v editore a to nasledujúcim spôsobom.

```
redoButton.setText(CurrentLocale.GetResource("redo"));
```

Metóda *setText* teda nastaví text komponentov podľa aktuálnej lokalizácie, a v prípade zmeny je volaná znova. Tieto funkcie, slúžiace k prepisu komponentom grafického užívateľského rozhrania sú implementované v hlavnom module. Prepisujú sa všetky komponenty hlavného okna programu, čo znamená, že pri spustení simulácie s istým nastavením lokalizácie bude načítané okno simulácie s lokalizáciou aktívnou počas spustenia. Pre spustenie simulácie v inom jazyku je potrebné zmeniť lokalizáciu v editore a následne znovu spustiť simuláciu, tentoraz v inom jazyku. Modul je vytvorený za účelom spravovania lokalizácií, a poskytovania prekladových súborov pre ostatné moduly. Trieda obsahujúca tieto metódy je exportovaná a viditeľná pre celú aplikáciu. V priečinku *resources* sa nachádzajú súbory *bundle.properies*, *bundle_cs.properties* a súbor *bundle_en_US.properties* v ktorých sa hľadajú hodnoty pre kľúče definované v aplikácií. Balík tried obsahujúci metódy hľadajúce hodnôt v súboroch je exportovaný a importovaný všetkými modulmi, ktoré definujú text v aplikácií.

5.1.3 Modul rozhrania služby

Prvým krokom ako definovať OSGi služby je definovať rozhranie, ktoré má služba poskytovať 5.2. Nazývame ho rozhraním služby (angl. service interface).



Obr. 5.2: Kroky pri vývoji OSGi služieb

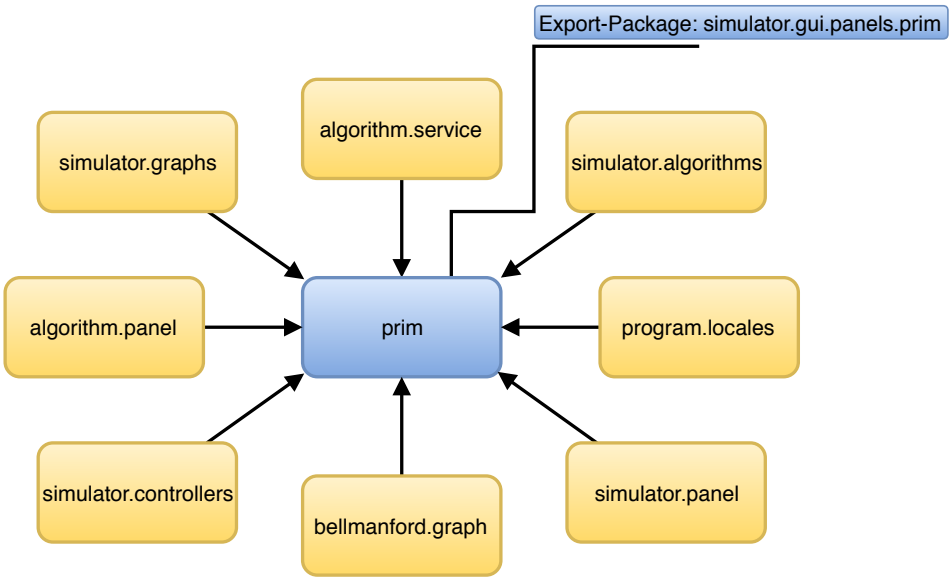
Rozhranie IAlgorithmService definuje dve metódy. Metóda *getName* je volaná pri stlačení tlačidla simulácie v editore. Pomocou nej zisťuje užívateľ aktuálny stav aktívnych služieb. Vďaka tejto metódy má možnosť vybrať konkrétnu inštanciu implementácie simulácie algoritmu. Program si najprv vyžiada názov jednotlivých implementácií služieb a po vybratí užívateľom volá metódu *createPanel*. Jediným parameterom je graf zobrazený v editore. Vracia panel simulátora konkretnéj implementácie algoritmu.

Druhým krokom vývoja OSGi služieb je definícia konzumenta služieb. Ten je je definovaný v hlavnom module a registruje dostupné služby pri volaní akcie simulácie.

```
public interface IAlgorithmService {
    public JFrame createPanel(AlgorithmGraph gaph);
public String getName();
}
```

5.2 Moduly poskytujúce implementáciu rozhriania služby

Posledným krokom pri vývoji OSGi služieb je vytvorenie implementácie rozhrania služby.



Obr. 5.3: Závislosti Primovho algoritmu na ostaných moduloch

Vybrané implementácie sa nachádzajú v oddelených moduloch. A to tak, že spolu s potrebnými triedami k definícií novej simulácie vytvárajú samostatný modul. Ako už bolo spomenuté, služby sa registrujú dôsledkom užívateľskej akcie volania simulátora. Z toho vyplýva, že ak počas behu programu nainštaluje nový modul, ktorý ponúka službu rozširujúcu rozhranie *AlgorithmService*, modul sa automaticky zobrazí vo vyskakovacom menu. Pridávanie a odoberanie modulov funguje dynamicky a počas behu programu.

5.2.1 Pomocné moduly

Obsahom modulu *simulator.controllers* sú pomocné triedy pre riadiacu jednotku simulátora. V module s názvom *simulator.panel* sa nachádza abstraktná trieda pre rozširovanie panelu. V panely sa nachádzajú riadiace tlačidlá slúžiace pre beh, krokovanie alebo zastavenie simulácie. *Simulator.algorithms* zahŕňa abstraktnú triedu pre algoritmus, poskytuje metódy pre prevod do podoby pre upravený panel pseudo-kódu. Dochádza k prevodu do podoby dokumentu a prepočítania čísel riadku.

Využívanie knižníc V rámci OSGi je možné používanie knižníc typickým Java spôsobom. V module *simulator.graphs* je v zdrojovom priečinku vytvorený priečinok *lib* s potrebnými knižnicami *jgraphx.jar*, *log4j-api-2.0-beta4.jar* a *log4j-core-2.0-beta4.jar*. S pridaním

knižníc musí byť aktualizovaný aj súbor MANIFEST.mf a definovať v $Bundle-ClassPath$ cestu k jednotlivým JAR súborom. Keďže je potrebné aby knižnice boli využívané aj ostatnými modulmi, sú potrebné balíčky využitím metadát exportované. V triede *AlgorithmGraph* sú pridané metódy poskytujúce kontrolu vhodnosti grafu pred spustením konkrétnej simulácie. Funkciou *isNegativeEdgeInGraph* program detekuje výskyt negatívne ohodnotených hrán v grafe. Trieda *BasicStyleSheet* bola rozšírená a štýly potrebné k novým algoritmom.

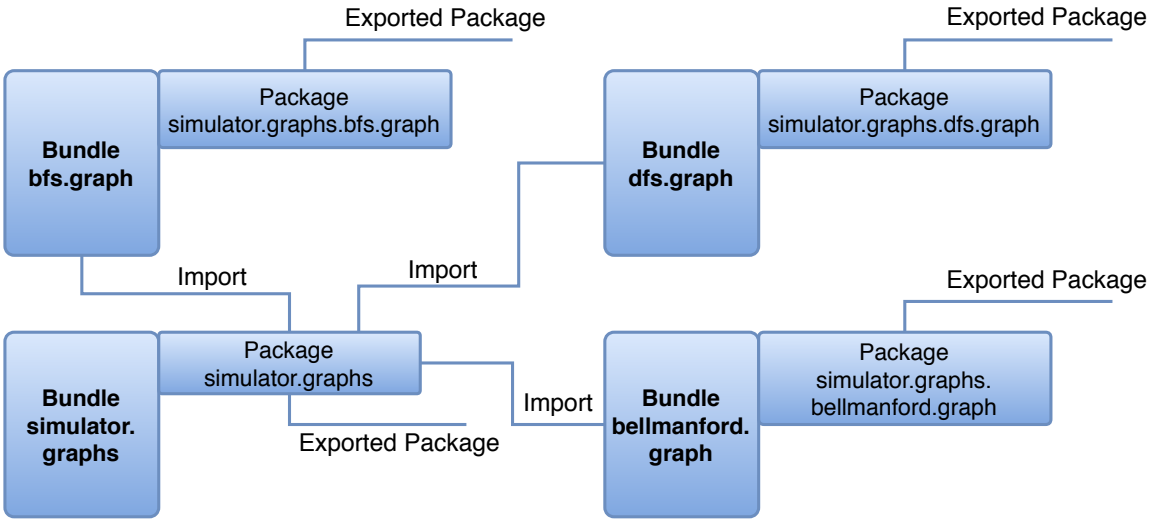
Modul grafického užívateľského rozhrania vychádza z balíka tried *simulator.gui* a obsahuje triedy pre GUI komponenty simulátora. Definuje tiež abstraktnú triedu *AbstractAlgorithmPanel* obsahujúcu hlavnú funkciu pre zastavenie simulátora.

5.2.2 Podporné moduly pre grafovú reprezentáciu algoritmov

Štruktúru objektov pre reprezentáciu základných hrán a uzlov 5.3 je možno rozšíriť podľa potrieb pridávaného algoritmu. Balíky trieb poskytujúce podporu pre algoritmy sú oddelené v samostatných moduloch.

- *bfs.graph*
- *dfs.graphs*
- *bellmanford.graph*

Triedy vznikli za účelom dodania funkcionality prvkom grafu v simulácií prehľadávania do šírky, prehľadávania do šírky a pre Bellman Fordov algoritmus. Oddelenie týchto objektov do samostatných modulov poskytuje možnosť ich využitia inými modulmi.

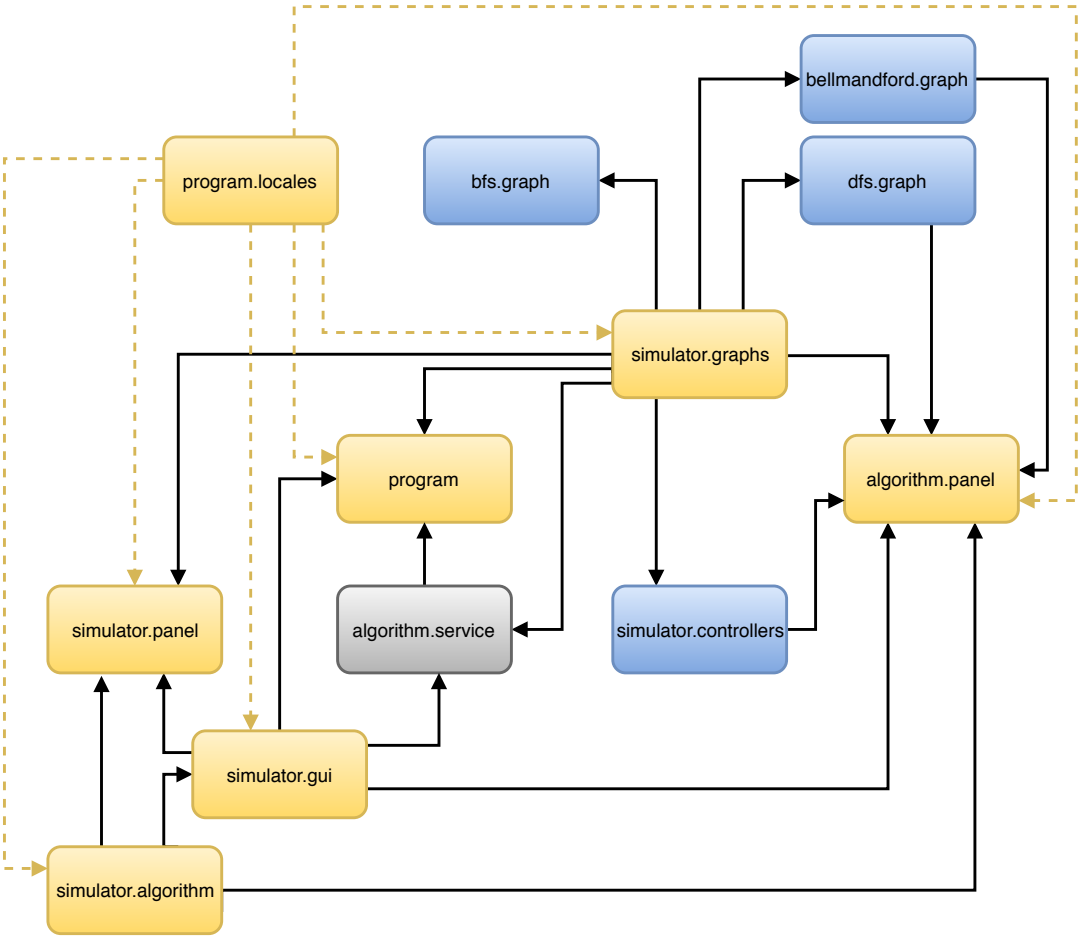


Obr. 5.4: Závislosti podporných modulov grafovej reprezentácie

5.3 Prepojenie modulov a spúšťanie aplikácie

Každý modul obsahuje vo svojom JAR archíve vlastnú konfiguráciu závisloti presne definovaných vo jeho manifest súbore. Na obrázku 5.5 je možné vidieť závislosti modulov aplikácie.

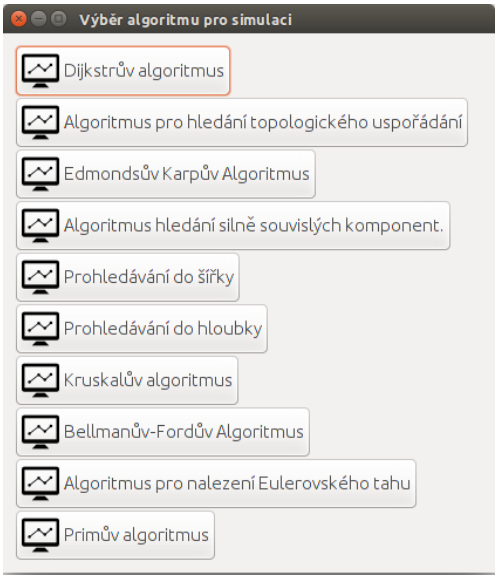
Pri spúšťaní modulov je potreba sa nad nimi ešte raz zamyslieť. Ak modul importuje balík tried z iného modulu, korešpondujúci balík musí byť v čase inštalácie modulu exportovaný jedným z nainštalovaných modulov. Nemôže nastať situácia, kedy jeden modul importuje balík druhého modulu a zároveň druhý modul importuje balík tried prvého modulu. Jedná sa o kruhovú referenciu, pri ktorej nie je možné aplikáciu korektne spustiť. Závislosti medzi modulmi aplikácie sú znázornené na obrázku 5.5.



Obr. 5.5: Závislosti modulov aplikácie

Ako bolo spomenuté vyššie, pri inštalácií modulov v OSGi kontajneri, záleží na poradí. Poradie je definované v konfiguračnom súbore *config.ini*. Ako prvé sa inštaluje modul *program.locales*, ktorý poskytuje preklad celej aplikácie. Je importovaný modulmi označenými žltou farbou na obrázku 5.5. Ako druhý sa nainštaluje modul *simulator.algorithm*, pretože neobsahuje žiadne importované balíky. Inštaláciou nasledujúceho modulu *simulator.graph* sa vyriešia závislosti pre *bfs.graph*, *dfs.graph* a *bellmanford.graph*. Ďalej na-

sledujú moduly *simulator.controllers*, *simulator.gui*, *algorithm.panel*, *simulator.panel*, *algorithm.service* a *program*. V tomto momente sú naištalované všetky potrebné moduly na beh aplikácie. Je možné načítanie, vytvorenie a úprava grafu. Nainštalovaním modulu implementujúceho konkrétnu simuláciu určitého algoritmu sa pri volaní simulácie objaví služba vo vyskakovacom okne viz. obrázok 5.6



Obr. 5.6: Vyskakovacie menu pre výber implementácie simulácie

5.4 Rozšírenie aplikácie novými modulmi

V tejto sekcií sú popísané algoritmy, ktoré boli doimplementovaná do aplikácie v podobe samostatných modulov. Primov a Kruskalov modul ponúka možnosť simulácie v interaktívnom režime. Interaktívny režim vyžaduje akcie užívateľa v behu simulácie. Zapnutie režimu je možné nájsť v kontextovom menu panela s pseudo-kódom algoritmu. Aktivácia tohto módu dodáva simulácií edukačné využitie aplikácie.

5.4.1 Aktivátor modulov

Parametrami oboch funkcií je práve objekt *BundleContext*, vďaka ktorému môžu moduly interagovať s rámcom OSGi.

5.4.2 Modul Primovho algoritmu

pseudo-kód Primovho algoritmu je prevzatý priamo z učebných materiálov predmetu GAL. Pred spustením simulátora prebehne kontrola výskytu záporne ohodnotených hrán v grafe. Pri ich výskyte vypíše program chybovú hlášku a simulátor sa nespustí.

Algoritmus pracuje s neorientovaným grafom, začína od užívateľom zvoleného počiatočného uzlu. Uloží si všetky susedné hrany do zoznamu. Nájde hranu s najnižším ohodnotením zo zoznamu a použije ju k prechodu do ďalšieho uzlu. Uloží susedné hrany ďalšieho vrcholu. Pokračuje pokiaľ nie sú navštívené všetky vrcholy.

Pre reprezentáciu vrcholov je využitá trieda *BellmanFordGraphVertex*, ktorá poskytuje funkcie inicializácie vrcholov a definíciu ich vzdialeností. Hrany sú reprezentované triedou *BellmanFordGraphEdge*, ktorá poskytuje metódy na definíciu ich číselného ohodnotenia.

Kontrolér simulátora rozširuje *AbstractShortestPathController*, ktorý poskytuje rozširujúce metódy pre algoritmy vyhľadávajúce najkratšiu cestu.

Algorithm 1 Primov algoritmus na výpočet najkratšej cesty v grafe

```
1:  procedure PRIMOV ALGORITMUS(G, w, r)
2:      InitializeSingleSource(G)
3:      Q ← V
4:      while Q ≠ ∅ do
5:          v ← Adj[u]
6:          for each v in VISITED do
7:              if v in Q && w(u, v) < key[v]  then
8:                  π[v] ← u
9:                  key[v] ← w(u, v)
10:             end if
11:         end for
12:     end while
13: end procedure
14: procedure INITIALIZESINGLESOURCE(G, r)
15:     for each u in V do
16:         key[u] ← ∞
17:         π[u] ← NIL
18:     end for
19:     key[r] ← 0
20: end procedure
```

V simulácií Primovho algoritmu je možnosť zapnúť interaktívny mód, ktorý ponúka beh simulácie spojený s interakciou užívateľa. Na začiatku simulácie je užívateľ vyzvaný k určenia počiatočného uzlu. V každom cykle algoritmu označuje ďalší spracovávaný vrchol. Označenie prebieha kliknutím ľavého tlačidla na myši na daný vrchol v grafe. Následne musí užívateľ posunúť beh simulácie o krok dopredu, alebo nechá simuláciu bežať. Pri ďalšej možnosti interakcie sa beh zastaví a užívateľ je znova vyzvaný k vykonaniu akcie.

5.4.3 Modul implementácie hľadania Eulerovej cesty

K implemetácií algoritmu pre nachádzanie Eulerovej cesty som využila ako základ upravený `Hierholzerov algoritmus`. Hierholzerov algoritmus hľadá Eulerovu kružnicu v neorientovanom grafe. Podmienkou pôvodného algoritmu je, že všetky vrcholy musia mať párny počet priliehajúcich hrán. Myšlienka spočíva v tom, že pre každý vrchol je potrebný párny počet hrán určený tak, aby jedna mohla byť označená ako vychádzajúca a druhá ako spiatočná. Pre Eulerovskú cestu v grafe platia však iné podmienky. Je povolené aby maximálne dva vrcholy boli ohodnotené nepárne, pričom hodnota vrcholu definuje počet hrán daného uzlu. V prípade že sa nachádzajú v grafe dva nepárne ohodnotené uzly, jeden musí definovať začiatok a druhý koniec Eulerovej cesty.

Algorithm 2 upravený Hierholzerov algoritmus pre Eulerovské grafy

```
1: procedure HIERHOLZEROV ALGORITMUS(G, w, r)
2:     InitializeVerticesDegrees(G)
3:     Q ← G.getAllEdges()
4:     VISITED ← ∅
5:     v ← startVertex
6:     while Q ≠ ∅ do
7:         VertexList = Adj[v]
8:         v ← maxUnvisitedadjEdge
9:         VISITED ∪ {v}
10:        if v == startVertex then
11:            break
12:        end if
13:    end while
14:    for each v in VISITED do
15:        if v.hasUnvisitedEdges() then
16:            x ← v
17:            while v ≠ x do
18:                VertexList = Adj[v]
19:                v ← maxUnvisitedAdjEdge
20:                VISITED ∪ {v}
21:            end while
22:        end if
23:    end for
24: end procedure
25: procedure INITIALIZEVERTICESDEGREES(G)
26:    for each v in V do
27:        value[v] ← edgeCount(v)
28:    end for
29: end procedure
```

Zatiaľ čo v Hierholzerovom algoritme nezáleží na výbere počiatočného uzlu, pri Eulerovej ceste áno. Preto sa v implementovanom algoritme pri spúšťaní simulácie kontrolujú hodnoty uzlov. V prípade nájdenia nepárneho uzlu, prebehne kontrola počiatočného uzlu. Ak bol označený vrchol z ktorého nemožno nájsť Eulerovu cestu, užívateľovi je tento fakt oznámený pomocou vyskakovacieho okna. Užívateľovi je rovnako oznámený názov vhodného vrcholu a automaticky sa nastaví ako počiatočný. Simulácia pokračuje a postupne prechádza susedné hrany, pokiaľ nenavštívi znova počiatočný uzol. Ak vznikla kružnica a neboli navštívené všetky hrany, algoritmus prechádza postupne navštívené vrcholy a kontroluje, či vrchol nemá susednú nenavštívenú hranu. Ak objaví hranu, ktorá nepatrí do nájdenej kružnice, uloží si daný uzol a pokračuje rovnakým spôsobom ako na začiatku. Keď sa vráti k uloženému uzlu, pokračuje v prehliadke navštívených vrcholov a opakuje cyklus, pokiaľ nie sú navštívené všetky hrany.

Simulácia využíva triedu pomocnú triedu *BfsGraphVertex* vytvorenú pôvodne na podporu algoritmu vyhľadávania v grafe do šírky. Táto trieda ponúka ohodnotenie uzlu číselnou hodnotou. Na reprezentáciu hrán je použitá trieda *BellmanFordGraphEdge*, vďaka ktorej sú číslované navštívené hrany podľa poradia definujúceho Eulerovu cestu.

5.4.4 Modul pre Kruskalov algoritmus

Kruskalov algoritmus, podobne ako Primov tiež hľadá najkratšiu cestu v grafe. pseudo-kód je rovnako prevzatý zo študijných materiálov predmetu GAL. V grafe sa nesmú nachádzať záporne ohodnotené hrany. Pri ich vyskýte sa program správa rovnako ako pri predošlom algoritme. Na začiatku sa vytvorí množina pre každý vrchol grafu. Potom sa vytvorí zoznam všetkých hrán v grafe, hrany sa usporiadajú do neklesajúcej postupnosti. Cyklicky sa vyberá hrana s najnižším ohodnotením a porovnáva sa, či patria vrcholy hrán do rovnakých množín. Ak sú rôzne, množiny sa zjednotia do jednej. V prípade že patria do rovnakej množiny, hrana sa vynechá. Algoritmus sa vyhýba kružniciam a pokračuje, pokiaľ nie sú navštívené všetky vrcholy.

Algorithm 3 Kruskalov algoritmus na výpočet najkratšej cesty v grafe

```
1:  procedure KRUSKALOV ALGORITMUS(G, w)
2:      A ≠ ∅
3:      for each node v in V do
4:          MAKESET(v)
5:      end for
6:      usporiadaj hrany E do neklesajucej postupnosti podľa váhy w
7:      for each node (u, v) in E ako neklesajúcu postupnosť do
8:          if FINDSET(u) ≠ FINDSET(v) then
9:              A ← A ∪ (u, v)
10:             UNION(u, v)
11:         end if
12:     end for
13: end procedure
```

Reprezentácia vrcholov a hrán je rovnaká ako pri implementácií Primovho algoritmu. Kontrolér rovnako rozširuje triedu *AbstractShortestPathController*.

V simulácií je možnosť zapnutia interaktívnejho režimu, kde užívateľ po vyzvaní vyberá počiatočný uzol. V každom cykle sa čaká na interakciu užívateľa v podobe výberu hrany, ktorá má byť podľa Kruskalovho algoritmu následne spracovaná.

5.4.5 Modul pre Edmond Karpov algoritmus

Ako bolo už spomenuté, hrany a uzly v grafe sú reprezentované triedou *mxCell* z knižnice JGraphX. Je vytvorená univerzálna trieda *AbstractGraphCell* a rozhraním *GraphCell*.Triedy poskytujúce podporu algoritmom dedia práve od tejto abstraktné triedy. Pri implementácií simulácie Edmond Karpovho algoritmu je potrebné aby hrany vizualizovali dve hodnoty. Hodnotu reprezentujúcu tok a hodnotu reprezentujúcu jej kapacitu. Preto bola hodnota vzdialenosti v týchto triedach implementovaná typom String.

Rovnako bola doimplementovaná možnosť výberu koncového uzlu. V editore v panely nájstrojov sa nachádza tlačidlo "Nastav koncový"v angličtine, ktoré definuje označený vrchol ako koncový. Ak nie je vybraný koncový uzol, vyberie sa automaticky posledný definovaný vrchol v grafe.

Hrany grafu načítaného v editore majú hranu ohodnotenú číslom typu *Double*. Po kontrole negatívnych hrán sa spustí simulácia. Na začiatku sú inicializované hrany na výšku toku 0 pomocou inštrukcie *SetFlowAndCapacityInstruction*, ktorá vezme aktuálnu hodnotu hrany a nastaví ju ako kapacitu.

Algorithm 4 Edmond Karpov algoritmus pre zistenie maximálneho toku v sieti

```
1: procedure EDMOND KARPOV ALGORITMUS(G, s, t)
2:     Q ← ∅
3:     path ← ∅
4:     f ← 0
5:     while true do
6:         path ← BFS(G, s, t)
7:         if path == ∅ then
8:             break
9:         end if
10:        Q ← path
11:        while Q == ∅ do
12:            u ← dequeue(Q)
13:            GetMinFlow(u)
14:        end while
15:        while path == ∅ do
16:            u ← dequeue(Q)
17:            UpdateFlow(u)
18:        end while
19:    end while
20: end procedure
```

Algoritmus sa začína hľadaním najkratšej cesty. BFS zvýrazní cestu v grafe. Po ukončení BFS, je volaná inštrukcia *FindPathByBFSInstruction*, ktorá hľadá zvýraznenú cestu v grafe. Začína koncovým uzlom a posúva sa cez zvýraznenú hranu k nižšie ohodnotenej hrane. Ak je cesta nájdená, pokračuje sa v simulácií Edmondovho Karpovho algoritmu vyhľadaním minimálneho maximálneho toku v danej ceste prechádzajúcej všetkými vybranými hranami. Ak cesta pozostáva z troch hrán s hodnotami 0/8, 5/10 a 4/12 (tok/kapacita), nájdený tok je definovaný číslom 5. Následne sa aktualizuje momentálny tok každej hrany. Algoritmus sa zastaví keď algoritmus BFS nenájde hľadanú cestu.

Pri simulácií algoritmu je miesto jedného grafu v panely, rozdelený panel na dva, prvý graf je graf načítaný z editoru programu. Druhý graf sa vytvára počas simulácia a znázorňuje reziduálnu sieť ku pôvodnému grafu. Je aktualizovaný vždy v momente, keď sú zmenené hodnoty toku hrán simulovaného grafu. Graf klonuje jeho vrcholy a hrany sú dané v závislosti s hodnotou toku a kapacity každej hrany. Ak hrana v pôvodnom grafe počas simulácie nadobudne hodnoty 8/20, v grafe reziduálnej siete sa vytvoria dve hrany, prvá orientovaná pôvodným smerom s hodnotou 12 reprezentujúca kapacitu mínus tok a druhá orientovaná spätným smerom s hodnotou 8 reprezentujúca spätný tok.

Kapitola 6

Pridanie implementácie novej simulácie algoritmu

6.1 Tvorba modulu

Ako je známe z predošlých kapitol, OSGi poskytuje podporu dynamického pridávania modulov do aplikácie. Pre tvorbu nového modulu je vhodným programom samotné prostredie Eclipse. Aby to bolo možné je potreba inštalácia doplnku nazývaného *Plugin development environment* (PDE). Tento doplnok poskytuje nástroje uľahčujúce vývoj rozširujúcich modulov. Obsahuje editor na tvorbu súboru MANIFEST.mf, možnosť spúšťania a ladenia aplikácie a rovnako sprievodca pri vytváraní nového PDE modulu.

K základom vývoja patrí správne nastavenie cielovej platformy (angl. target platform). Nastavením poskytujeme prostrediu informáciu, pre aký systém je modul vyvíjaný. Veľkou výhodou je možnosť generovať modul vo formáte JAR, ktorý je potrebný pri inštalácií do aplikácie.

6.1.1 Príprava do formátu k inštalácií

Keď je vývojár s implementáciou modulu hotový, modul má v záložke s názvom *Dependencies* manifest súboru definované potrebné závislosti, je definovaná cestu k triedam a exportované balíky v záložke *Runtime*. V záložke *Build* musia byť označené všetky zdroje, ktoré chceme exportovať. Pravým tlačidlom klikneme na Project->Export->Plugin-Development->Deployable Plugins and Fragments.

6.2 Implementácia nového modulu

Kapitola obsahuje popis vytvárania nového modulu. Popisuje jednotlivé triedy a ich najdôležitejšie vlastnosti a metódy.

6.2.1 Čo musí poskytovať nový modul

Každý modul nového algoritmu musí byť schopný simulovať určitý algoritmus na grafe definovanom v editore programu. Prvou predispozíciou je teda schopnosť získania grafu a jeho prekreslenie do nového okna simulátora. Nový modul vytvára vlastné samostatné okno typu *JFrame* a poskytuje programu viacero tried. Triedy reprezentujú objekty v okne simulátora.

Okno simulátora obsahuje panel s už spomínaným grafom, panel premnenných zobrazujúcich stav simulácie, panel vizualizujúci pseudo-kód daného algoritmu a panel slúžiaci na ovládanie. Samotná simulácia môže využívať nástroje tried iných modulov. Pre korektnú registráciu implementácie rozhrania služby musí modul implementovať triedu `Activator`, ktorý pri spustení modulu registruje službu rozhrania. Implementácia služby je definová triedou rozširujúcou rozhranie služby.

```
public class Activator implements BundleActivator {

    public void start(BundleContext context) throws Exception {
        IAlgorithmService service = new AlgorithmServiceKruskal(context);
        context.registerService(IAlgorithmService.class.getName(), service,
                null);
    }
    public void stop(BundleContext context) throws Exception {
    }
}
```

6.2.2 Implementácia služby IAlgorithmService

Pri registrácií sa vytvára nová inštancia služby implementujúca rozhranie *IAlgorithmService*.Pre zaistenie dostupnosti modulu v aplikácií, implementuje každý modul metódy rozhrania služby. Ako už bolo spomenuté v kapitole 5 trieda definuje dve metódy. Názov algoritmu vrátený funkciou *getName* je nastavený podľa aktuálne nastavenej lokalizácie. Pri volaní metódy *createPanel* je u každého algoritmu spustená kontrola, či graf vyskytujúci sa v editore je vhodný pre zvolenú simuláciu. Vytváraný modul teda importuje rozhranie služby `algorithm.service` a modul `program.locales` poskytujúci prekladové súbory a metódy pre vyhľadanie správnej hodnoty k požadovanému kľúču.

```
public class AlgorithmServiceKruskal implements IAlgorithmService {

  public JPanel panel;
  public AbstractAlgorithm algorithm;
  public BundleContext context;
  public AlgorithmServiceKruskal(BundleContext context) {
this.context = context;
  }
  public JFrame createPanel(AlgorithmGraph graph) {
if (graph.isNegativeEdgeInGraph()) {
  JOptionPane.showMessageDialog(
panel,
CurrentLocale.GetResource("SimulatorNegativeNonValidGraph"),
              CurrentLocale.GetResource("error"),
              JOptionPane.ERROR_MESSAGE);
  return null;
}
return AlgorithmFrame.getAlgorithmFrame(
new KruskalPanel(graph),
```

```
CurrentLocale.GetResource("KruskalSimulatorFrame"));
}

@Override
public String getName() {
return CurrentLocale.GetResource("KruskalAlgorithm");
}
}
```

6.3 Potrebný obsah modulu

Pre účely simulátora programu sú vytvorené objekty reprezentujúce jednotlivé komponenty. Okno simulátora sa skladá zo štyroch panelov.

6.3.1 Trieda panelu pseudo-kódu

Prvým objektom okna simulátora je textový panel znárorňujúci pseudo-kód algoritmu. V module *simulator.algorithm*, je definovaná abstraktná tieda *AbstractAlgorithm* rozširujúca triedu *DefaultStyleDocument*. Poskytuje základné metódy pre prevod textového dokumentu pseudo-kódu do podoby vhodnej pre simulátor programu. Pri implementácií pseudo-kódu nového algoritmu je možné rozšíriť túto a ďalšie pomocné triedy. Abstraktná trieda využíva objekty definované v module *simulator.algorithm* nasledovným spôsobom.

```
new AlgorithmElement(AlgorithmBase.algorithmLineName0, new
    AlgorithmContent[] {
     new AlgorithmContent(AlgorithmBase.algorithmKeywordsName, "for "),
     new AlgorithmContent(AlgorithmBase.algorithmOthersName, CurrentLocale
         .GetResource("eachNode")),
     new AlgorithmContent(AlgorithmBase.algorithmVariableName, " v"),
     new AlgorithmContent(AlgorithmBase.algorithmOthersName, " in "),
     new AlgorithmContent(AlgorithmBase.algorithmVariableName, "V"),
     new AlgorithmContent(AlgorithmBase.algorithmKeywordsName, " do"),
});
```

Znázornená trieda *AlgorithmElement*, predstavuje jeden element algoritmu, ktorý reprezentuje jeden riadok. Pri jeho vytváraní mu sú poskytnuté dva parametre. Prvým je názov štýlu daného riadku. Štýly sú definované reťazcom *algorithmLineName<číslo>*, kde číslo definuje veľkosť ľavého odsadenia riadku. Definované sú štýly v rozmedzí 0 - 3, ktoré pomáhajú logicky delit častí pseudo-kódu. Druhým parametrom konštruktora určuje trieda obsah textu. Obsah môže byť definovaný viacerými spôsobmi:

- *algorithmName*
- *algorithmKeywordsName*
- *algorithmVariableName*
- *algorithmMethodsName*
- *algorithmOthersName*

Rozlišuje sa štýl zobrazenia názvu algoritmu, názvu metód, mená premenných, kľúčové slová a bežný text.

6.3.2 AbstractController

Kontrolér samotného algoritmu je najdôležitejšiou časťou simulácie. K jeho implementácií využívame triedu *AbstractController*, v ktorej sú definované základné inštrukcie. Parametrami konštruktora je graf editora, názov algoritmu, panel s pseudo-kódom a panel premenných.

```
public NewAlgorithmController(final AlgorithmGraphComponent graphCom,
                        final JLabel graphComponentLabel,
                        final VisualizationPseudocodePanel completeCodePanel,
                        final VariablesPanel variablesPanel);
```

Kontrolér vytvára pásku inštrukcií. Páska je definovaná ako zoznam typu *Instruction*, ktorá je definovaná nasledovne:

Implementovaný algoritmus je upravený tak, aby našiel nie len Eulerovu kružnicu, ale aj Eulerovu cestu. Už pri samotnom volaní simulácie dochádza ku kontrole grafu. Kontroluje sa, či sa v grafe nachádza Eulerova.Ak sa v grafe cesta nachádza a v grafe sa objavujú nepárne ohodnotené vrcholy, je potrebné aby sa algoritmus začínal práve z jedného z nich. Ak je nastavený počiatočný uzol inak, rovnako sa otvorí vyskakovacie okno s dotazom zle zvoleného počiatočného uzlu. Počiatočný uzol sa automaticky správne nastaví a simulácia môže začať.

```
/**
 * Rozhranie pre objekt instrukcie
 */
public interface Instruction {
  public void preform();
  public void stop();
  public int nextTapePos();
  public boolean isInteractive();
}
```

Pre vykonávanie bežných inštrukcií je vytvorená trieda *BasicInstruction*, ktorá implementuje rozhranie inštrukcie a inkrementuje pozíciu v páske. Metóda *resetController* uvedie simuláciu do počiatočného stavu.

```
public abstract class BasicInstruction implements Instruction{
  @Override
  public boolean isInteractive() {
    return false;
  }

  @Override
  public int nextTapePos() {
```

```
      return tapePosition + 1;
    }

    @Override
    public void stop(){}
    }
```

6.3.3 AbstractAlgorithmPanel

Trieda predstavuje okno simulátora a definuje metódu na jeho zastavenie. V panely simulátora je potrebné vytvoriť graf oknu simulátora, nastaviť rozloženie komponentov,štýl vizualizácie jeho komponentov a určiť jeho orientovanosť.

```
public abstract class AbstractAlgorithmPanel extends JPanel{
   public abstract void stop();
}
```

6.3.4 BasicStylesheet

Posledná trieda definuje štýly pre simuláciu grafu. Nový modul si môže určovať štýly hrán a vrcholov. Napríklad definícia neorientovanej nenavštívenej hrany v Module Kruskalovho algoritmu.

```
public class KruskalStyleSheet extends BasicStylesheet {

  private KruskalStyleSheet() {}

  Map<String, Object> styleEdge = new HashMap<>();
    styleEdge.put(mxConstants.STYLE_STROKECOLOR, "08E8DE");
    styleEdge.put(mxConstants.STYLE_STARTARROW, "none");
    styleEdge.put(mxConstants.STYLE_ENDARROW, "none");
    styleEdge.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_CONNECTOR);
    styleEdge.put(mxConstants.STYLE_FONTSIZE, 20);
    styleEdge.put(mxConstants.STYLE_FONTCOLOR, "black");
    styleEdge.put(mxConstants.STYLE_EDGE, mxConstants.ENTITY_SEGMENT);
    newSheet.putCellStyle(EDGE_STYLE, styleEdge);
    newSheet.setDefaultEdgeStyle(styleEdge);
```

Kapitola 7

Testovanie nových implementácií algoritmov

V tejto sekcií sú znázornené grafy v rovnakej vizuálnej podobe v akej sú zobrazené v okne simulátora. Pre účely testovania bola vytvorený sada grafov pre novo implementované algoritmy. Algoritmy boli rovnako otestované na príkladoch grafov z predmetu GAL. V simulácií je možné posúvať beh o krok dozadu, dopredu alebo beh reštartovať. Užívateľ môže v panely s pseudo-kódom nastaviť breakpoint definujúci miesto, v ktorom sa beh simulácie zastaví. Jednotlivé inštrukcie musia mať implementované metódy pre vykonanie a vrátenie do predošlého alebo pôvodného stavu. Preto bol pri testované kladený dôraz na krokovanie dopredu a vykonávanie spätných krokov. Napríklad pri inštrukcií naplnenia fronty, sa musí obsah prvkom fronty rovnať pôvodnému. Konzistencia inštrukcií zabezpečuje určitosť a vyhýba sa nedefinovanému stavu behu simulácie. Pred spustením simulácie sa testuje vhodnosť grafu pre konkrétny algoritmus. To zapezpečuje správne fungovanie simulácie.

Grafové algoritmy boli otestované v oboch jazykových variantách, čím bol skontrolovaný výskyt kľúčov v oboch prekladových súboroch.

7.0.1 Edmondov Karpov algoritmus

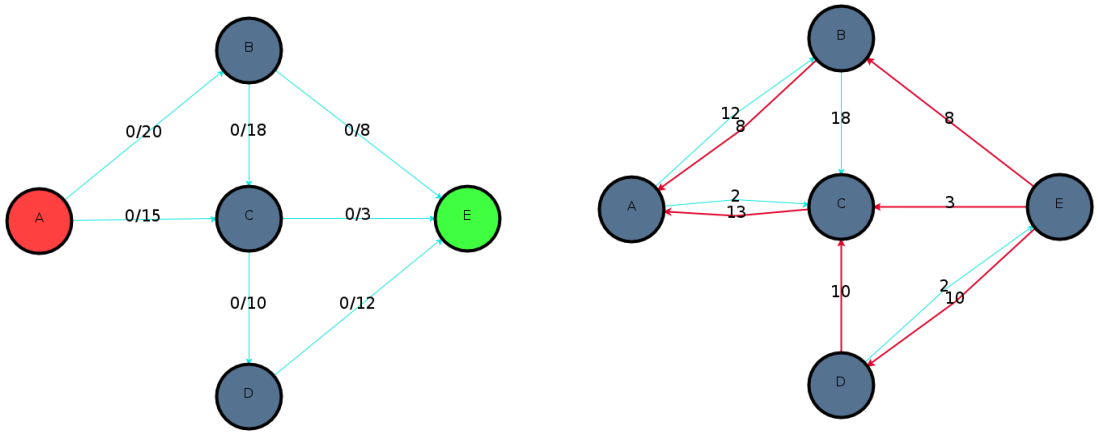
Na obrázkoch 7.1 je možné vidieť graf spracovávaný Edmond Karpovým algoritmom. Ľavá strana vizualizuje stav grafu po inicializácií a nastavení hodnôt tokov a kapacít. Na pravej strane možno vidieť reziduálnu sieť, ktorá je reprezentovaná druhým grafom, ktorý je počas simulácie vytvorený v panely simulátora. Modré hrany značia kapacitu hrany, zatiaľ čo červené znázorňujú spätný tok.

7.0.2 Primov algoritmus

Primov algoritmus funguje správne, ak sa v grafe nenachádzajú záporne ohodnotené hrany, preto je možné spustiť simulácie len pre vhodný graf. PO inicializácií sa zafarbí počiatočný vrchol červenou farbou. Hodnoty vrcholov sa menia počas behu simulácie. Po ukončení simulácie je zvýraznená najkratšia cesta v grafe.

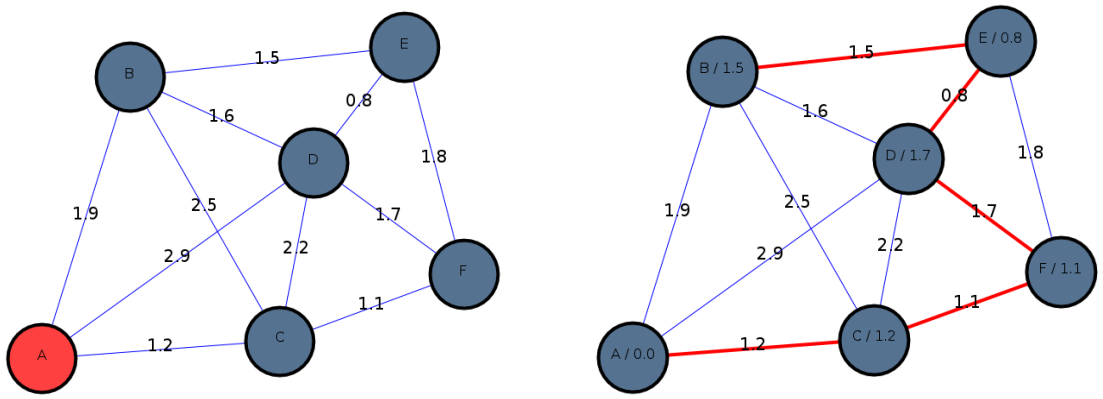
7.0.3 Kruskalov algoritmus

Keďže Kruskalov algoritmus patrí do rovnakej kategórie algoritmov ako Primov algoritmus, platia pri ňom podobné podmienky. Algoritmus sa nespustí pri existencií zápornej hrany v grafe. Hodnoty vrcholov nie sú počas behu menené, hrany sa postupne zvýrazňujú od



(a) Po inicializácií (b) Rezidálna sieť

Obr. 7.1: Simulácia Edmondsovho Karpovho algoritmu



(a) Po inicializácií (b) Po dokončení simulácie

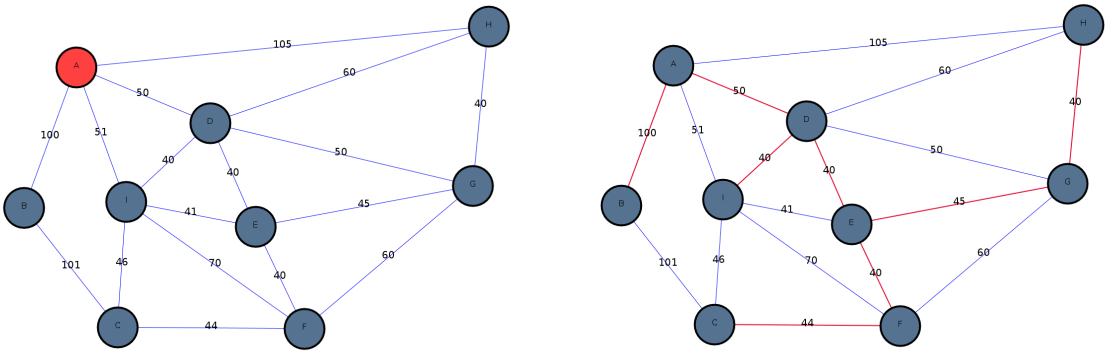
Obr. 7.2: Simulácia Primovho algoritmu

najmenej ohodnotenej vhodnej hrany. Po ukončení simulácie je cesta zvýraznená červenou farbou.

7.0.4 Algoritmus pre nájdenie Eulerovej cesty

Testovanie algoritmu pre nájdenie Eulerovej cesty som rozdelila na dve kategórie. Testované sú grafy obsahujúce Eulerovu kružnicu a grafy obsahujúce Eulerovu cestu. V oboch prípadoch je však za účelom správnej funkcionality potrebné pred simuláciou zabezpečiť splnenie podmienok daného algoritmu.

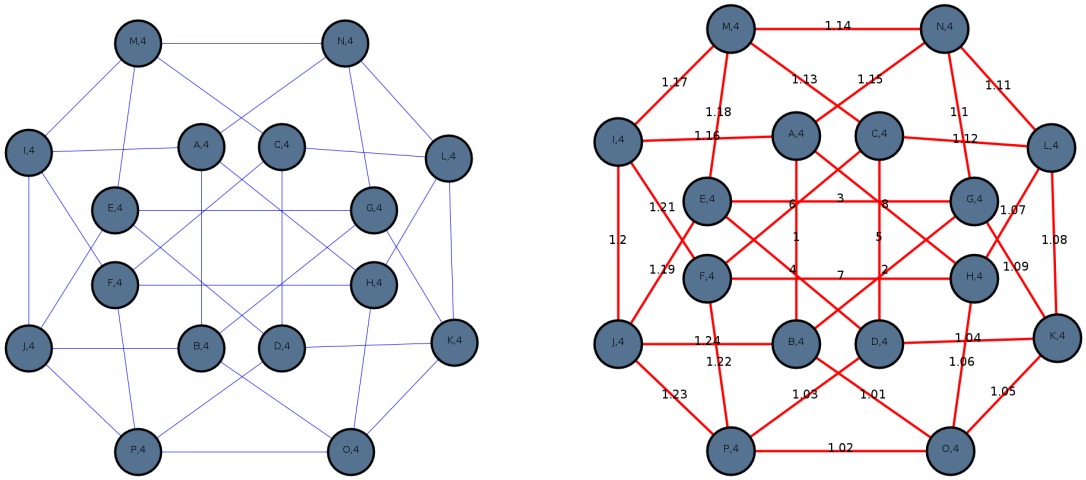
Pri prechádzaní grafu sú hrany postupne indexované. V prípade, že pri spúšťaní simulácie nie sú detekované žiadnenegatívne ohodnotené uzly. Simulácia sa spustí bez ďalšej kontroly počiatočného uzlu. Algoritmus hľadá postupne kružnice v grafe. V prípade dokončenia kružnice, ktorá neprechádza všetkými hranami grafu, sa hľadá prvý uzol v ceste, ktorý má nenavštívenú hranu. Uloží sa index hrany prichádzajúcej do uzlu a pri vytváraní novej



(a) Po inicializácií (b) Po dokončení simulácie

Obr. 7.3: Simulácia Kruskalovho algoritmu

kružnice sa inkrementuje. V konečnom stave algoritmu sú indexované hrany postupne, tak aby vytvárali Eulerovu kružnicu.



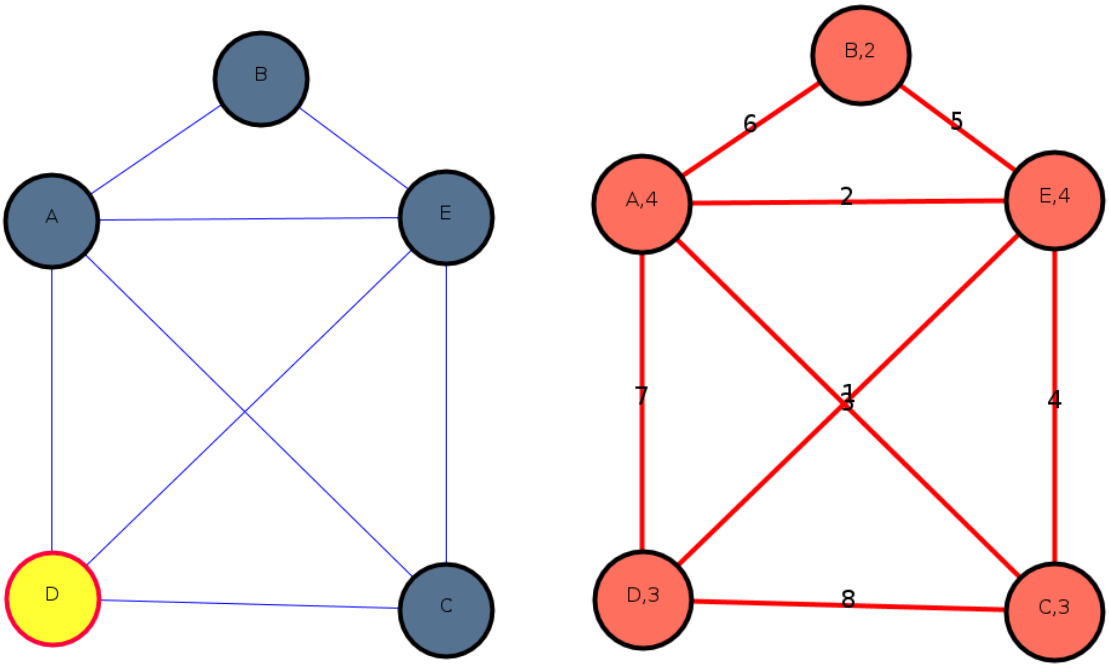
(a) Po inicializácií (b) Po dokončenie simulácie

Obr. 7.4: Simulácia Algoritmu pre nájdenie Eulerovej kružnice

Ak sa nájdu v grafe viac ako dva vrcholy s nepárnym počtom susedných hrán, modul namiesto panelu simulátora vracia null a otvorí vyskakovacie okno s dotazom na graf, neobsahujúci Eulerovu cestu.

7.1 Prínosy práce

Vývojom aplikácií s použitím platformy OSGi programátor môže počas celého vývoja získavať nové skúsenosti. Nasadenie OSGi štandardu do už existujúcej aplikácie prináša aj svoje problémy. Táto technológia má dlhú krivku učenia a je potrebný čas, pokiaľ si vývojár osvojí všetky aspekty technológie. V tejto práci neboli využité všetky prostriedky, ktoré tento štandard ponúka. Výhody sú zjavné hlavne pre projekty väčšieho rozsahu. Mo-



(a) Vstupný graf

(b) Po dokončenie simulácie

Obr. 7.5: Simulácia Algoritmu pre nájdenie Eulerovej cesty

dulárny návrh novej architektúry redukuje zložitosť, dovoľuje lepšiu variabilitu, paralelný vývoj, zlepšenú schopnosť testovania, znovupoužiteľnosť a flexibilitu.

7.1.1 Stabilita

OSGi implementace Equinox je stabilný produkt. Pri rozširovaní aplikácie sa môže stať že programátor omylom vytvorí cyklickú referenciu. Zatiaľ čo pri vývoji v Eclipse IDE sa moduly spustia v konzole sa aplikácia nespustí. Preto je dôležité dbať na definíciu závislosti v súboroch MANIFEST.mf.

Kapitola 8

Záver

Cieľom práce bolo dosiahnutie modularity existujúcej Java aplikácie. Modularita je dosiahnutá rozdelením programu na menšie moduly a tým je zaistená ľahšia rozšíriteľnosť novými grafovými algoritmami. Čitateľovi je predstavená architektúra existujúcej nemodulárnej aplikácie a priblížená technológia OSGi, z ktorej vychádza navrhnuté rozdelenie modulov. Aplikácia je rozšírená novými modulmi Primovho, Kruskalovho, Edmondovho Karpovho a upraveného Hierholzerovho algoritmu pre nájdenie Eulerovej cesty. Každý algoritmus predstavuje samostatný modul, obsahujúci funkcionalitu definujúcu jeho chovanie. Požiadavkou bolo rovnako vytvoriť viacjazyčné užívateľské rozhranie.

Dynamickej zmeny jazyka bolo dosiahnuté využitím Eclipse Plugin internacionalizácie. Boli vytvorené súbory typu *.properties*, v ktorých sú hľadané hodnoty k určitým kľúčom. Užívateľ nastavuje lokalizáciu aplikácie a program nastaví ResourceBundle pre správny prekladový súbor. Keďže používaná knižnica Swing nepodporuje dynamickú zmenu textu, všetky komponenty sú pri zmene lokalizácie prepísané novým reťazcom.

Užívateľovi je umožnené využívať ľubovoľný počet simulácií v jednom čase. Simulácia je spustená v jazyku aktuálne nastavenej lokalizácie. Je teda možné spustiť dve inštancie rovnakej simulácie v dvoch jazykoch súčasne. Aplikáciu je možné rozšíriť novým algoritmom počas samotného behu aplikácie nainštalovaním nového modulu. Moduly podporujúce simulácie algoritmov sú registrované službou rozhrania. Po ich inštalácií sa automaticky zobrazia v ponuke algoritmov k simulácií. Vytvorením rozhrania služby je možné registrovať všetky aktuálne aktívne moduly algoritmov. Moduly je možné dynamicky pridávať aj odoberať z aplikácie počas behu programu.

Spolu bolo vytvorených 22 modulov, z toho 11 modulov je nutných pre beh programu. Ostatné moduly rozširujú program o konkrétne simulácie grafových algoritmov.

Aplikáciu je možné využiť k výuke grafových algoritmov. Užívateľ má možnosť vytvorenia vlastného grafu, výberu algoritmu, možnosť krokovania, zastavenia a ovládanie behu simulácie. Aplikáciu je možné rozšíriť o nové algoritmy. Spôsob rozširovania je obsiahnutý v kapitole 6.

Literatúra

[1] Alder, G.: JGraphX User Manual. *Copyright (c) JGraph Ltd 2006-2017*, ročník JGraphX Version 3.9.3, 2016-2017.

[2] Alliance, O.: OSGi Architecture. `https://www.osgi.org/developer/architecture/`.

[3] Alliance, O.: OSGi^{TM} Service Platform Release 4 Core Specification Version 4.3. `https://osgi.org/javadoc/r4v43/core/overview-summary.html/`, 2000,2012.

[4] Hall, R. S.: OSGi Implementation and Experience Report. `https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html/`, 2004.

[5] Jeff McAffer, S. A., Paul VanderLei: OSGi and Equinox Creating Highly Modular Java System. 2004.

[6] O'Conner, J.: Creating Extensible Applications With the Java Platform. `http://www.oracle.com/technetwork/articles/java/extensible-137159.html`, September 2007.

[7] Oracle: Class ServiceLoader. Březen 2015.
URL
`https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html/`

[8] Oracle: Java Platform Standard Edition 8 Documentation. `https://docs.oracle.com/javase/8/docs/index.html/`, 2015.

[9] Pedro Capelastegui, F. M., Olga Gadyatskaya; AntonPhilippov: Security-by-Contract for the OSGi platform. Leden 2012.

[10] Tavares, A. L.; Valente, M. T.: A Gentle Introduction to OSGi. *SIGSOFT Softw. Eng. Notes*, ročník 33, č. 5, Srpen 2008: s. 8:1–8:5, ISSN 0163-5948.

[11] Varadinek, I. J.: Brno: Fakulta Informačních Technologií, Vysoké Učení Technické, 2012/2013.
URL `https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=118630`

[12] Vogel, L.: *Eclipse RCP and Plugin Internationalization*. 2009.

Príloha A

Obsah CD

- galdemo - priečinok súborov potrebných pre spustenie aplikácie, moduly vo formáte JAR a konfuguráčný súbor config.ini
- source - zložky jednotlivých modulov, ktoré obsahujú všetky zdrojové súbory
- README - súbor so základnými informáciami k obsahu média
- xgalan02.pdf - text práce vo formáte pdf