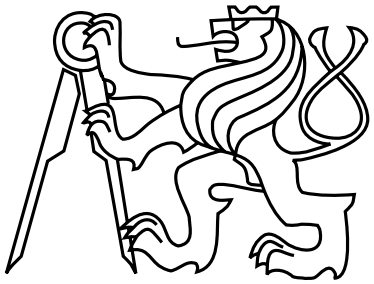
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Aplikace pro generování testovacích situací pro techniku Process Cycle Test

Lukáš Löwinger

Vedoucí práce: Ing. Miroslav Bureš, Ph.D.

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Web a multimédia

21. května 2014

Poděkování

Chtěl bych především poděkovat vedoucímu práce Ing. Miroslavovi Burešovi, Ph.D za pomoc jak s pochopením problematiky zadaného tématu, tak za cenné rady při tvorbě aplikace a celé práce. Dále bych chtěl poděkovat osobám, které mi pomohly při uživatelských testech aplikace.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 15. 5. 2014 .....

Abstract

This bachelor thesis deals with the design and implementation of an application which will generate testing situations based on a graph in accordance with the Process Cycle Test technology contained in TMap Next. The graph can be drawn in an interactive editor or generated based on values from a synchronized table. The algorithm for generating test situations will be the result of an analysis. Furthermore, the application will enable exporting and importing the graph as well as the testing situations. The implementation will be written in the Java programming language and tested with a set of JUnit tests, functional tests and a set of comparative tests in order to verify the quality of the algorithm.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací aplikace, která bude generovat ze zadaného grafu testovací situace dle techniky Process Cycle Test z metodiky TMap Next. Graf bude možné nakreslit v interaktivním editoru, nebo pomocí zadaných hodnot do tabulky, která bude s grafem synchronizována. Algoritmus použitý pro generování testovacích situací bude výsledkem analýzy. Dále bude aplikace umožňovat export a import grafu a testovacích situací. Implementace bude napsána v jazyce Java a otestována sadou JUnit testů, funkčních testů a sadou srovnávacích testů pro ověření kvality algoritmu.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

1.1 Cíle práce

Cílem této práce je navrhnout a implementovat desktopovou aplikaci, která bude uživateli umožňovat nakreslení grafu v interaktivním editoru a následně z tohoto grafu vygenerovat testovací scénáře dle metody *Process cycle test*.

K vytvoření aplikace, zabývající se generováním testovacích situací z activity diagramů, vedlo hned několik faktorů. Jedním z nich je ten, že pro objemnější data se ruční počítání stává složitým procesem a je zde více prostoru k vytvoření chyb, které vedou ke špatně vytvořeným testům. Lze si samozřejmě vypomoci jinými programy (Excel, Enterprise Architect[1], ...), ty ale nejsou vytvořené tak, aby vyhovovaly veškerým požadavkům této problematiky. Dalším faktorem je ten, že software řešící generování testovacích situací pro test design techniku *Process cycle test* existuje pouze v omezené podobě (viz odstavec 1.2). To je jeden z hlavních důvodů pro vytvoření nové aplikace, která by se tímto problémem měla zabývat.

Aplikace by měla splňovat přinejmenším dvě kritéria a to možnost nakreslení grafu v interaktivním editoru (podobně jak je tomu v Enterprise Architect, byť ve značně omezenějším rozsahu) a schopnost z tohoto grafu vygenerovat testovací situace do hloubky N (o technice více v odstavci 1.3). Další funkcionality už pouze rozšiřují program do takové podoby, aby byl uživatelsky přívětivý (struktura projektů, možnost ukládání a načítání těchto projektů) a aby splňoval určitou formu podobných programů. To hlavně proto, aby uživatel hned po první interakci s programem nemusel nahlížet do manuálu k aplikaci.

1.2 Analýza stávajících řešení

Firma Sogeti, sídlící v Nizozemsku, provozuje na svém webu aplikaci nesoucí název Cover [10]. Než popíšeme tuto aplikaci, musíme zmínit, že je tato firma tvůrcem metodiky TMap Next (Test Management Approach) [9], která popisuje metodu *Process Cycle Test*. Je tedy logické, že pro zjednodušení tvorby testů napsala i aplikaci (byť velmi jednoduchou). Cover

[1] Enterprise Architect patří mezi programy, které jsou zaměřené na kreslení různých typů diagramů (activity diagram, sekvenční diagram, ...). Pro kreslení grafů je tedy ideální, ale zde jeho pomoc s generováním testovacích situací končí.

je webová služba, která dostane na vstup data v textové podobě a vytvoří z nich testovací situace pro hloubku 1 a 2. Je tedy značně omezená, protože podporuje jen dvě hloubky pokrytí. Druhým problémem je to, že služba funguje pouze pro interní účely, neboť uživatel musí mít email u firmy Sogeti.

1.3 Technika Process Cycle Test

Process cycle test je název pro metodu, která umožňuje vytvářet testovací scénáře z UML activity diagramů. Tyto diagramy jsou nejčastěji používány pro popis business procesů v podniku. Pokud dojde na testování těchto procesů, je potřeba pokrýt ve scénářích co nejvíce kombinací, aby uniklo co nejméně chyb (pokud možno aby neunikly žádné). Tato metoda se tedy používá pro vytváření testovacích scénářů s tím, že nám zaručí určitou hloubku pokrytí. To znamená, do jaké míry jsou kombinace v diagramu pokryté. Z UML diagramu si vystačíme pouze s jeho strukturou, která se pro pokrytí cest v programu používá zjednodušeně – pouze větvící body (uzly) a přechody mezi nimi (hrany). Tím vznikne orientovaný graf (Obr.3.2), ve kterém se snažíme vhodně nakombinovat akce.

Jak mají být akce zkombinované za sebou nám říká tabulka 1.1 převzata z prezentace [2]. Podle níž je popsán algoritmus (sekce 3.2.1), který vytváří testovací kombinace pro danou hloubku pokrytí.

Hloubka pokryti	**Popis**	**Kdy použít**
1	Každá akce (hrana grafu) se vykoná jednou	Testy málo prioritních akcí, smoke testy
2	Pro každé větvení (uzel) projdeme všechny kombinace možných vstupních akcí do uzlu a výstupních akcí z uzlu	Střední úroveň testování
3	Pro každé větvení (uzel) projdeme všechny kombinace možných vstupních akcí do uzlu a 2 na sebe navazujících výstupních akcí	Vysoká intenzita testů
N	Pro každé větvení (uzel) projdeme všechny kombinace možných vstupních akcí do uzlu a N-1 na sebe navazujících výstupních akcí	

Tabulka 1.1: Definice hloubky pokrytí

Kapitola 2

Analýza a návrh řešení

2.1 Úvod

Tato kapitola se bude zabývat analýzou požadavků a návrhem aplikace. Nejprve bude uveden seznam funkčních a nefunkčních požadavků, z kterých se bude později vycházet. V další podkapitole budou podrobněji popsány čtyři skupiny případů užití, vždy graficky znázorněny spolu s textovým vysvětlením. Třetí podkapitola bude popisovat základní uživatelské scénáře. Jednotlivé kroky scénářů budou mít strukturu číselného seznamu, spolu s výčtovým seznamem prerekvizit, které musejí být zajištěny pro splnění konkrétního scénáře. Předposlední kapitola vypíchne dva základní procesy v aplikaci. Ty budou graficky znázorněné pomocí activity diagramu. Závěr této kapitoly předvede náčrt hlavní obrazovky aplikace.

2.2 Přehled funkčních a nefunkčních požadavků

2.2.1 Funkční požadavky

Vytvoření grafu interaktivním editorem

1. Systém umožní uživateli v interaktivním editoru nakreslit schéma grafu.
2. Nakreslené schéma se promítne do tabulky.

Vizualizace grafu

1. Systém nakreslí schéma grafu podle tabulky.

Konfigurace grafu pomocí tabulky

1. Uživatel bude mít možnost zadat seznam větvicích bodů (uzlů), přechodů (hran) včetně akcí v rámci přechodu (ty se počítají jen jako metadata hrany) a začátku grafu.
2. Větvící body a přechody budou mít ID, název a popis.
3. Uživatel bude mít možnost editovat tabulku, která popisuje graf (uzly x hrany).

4. Pokud se změní tabulka, upraví se příslušně i graf.

Import a export grafu ve formátech XML a CSV

1. Import a export tabulky grafu jako formát CSV se záhlavím.
2. Import a export tabulky grafu jako formát XML.

Generování testovacích situací obecně pro test depth level N

1. Pro určený coverage level systém umožní generovat testovací situace. Testovací situace budou uložené v tabulce. Bude možno uložit více verzí tabulek testovacích situací pro různé coverage levels.
2. Pokud dojde ke změně schématu stavového automatu, systém vhodným způsobem propaguje změny i do tabulek testovacích situací (barevně podbarví testovací situace).

Export testovacích situací v CSV a XML

1. Export testovacích situací jako formát CSV s příslušným záhlavím.
2. Export testovacích situací jako formát XML.

2.2.2 Nefunkční požadavky

Běhové prostředí java

1. Aplikace bude vyvíjena v jazyce Java.

Nezávislé na operačním systému

1. Zvolené vývojové prostředí je multiplatformní v tom smyslu, že operační systém potřebuje pouze JVM pro spuštění aplikace.

Rozšiřitelnost a modifikovatelnost

1. Program bude snadno rozšiřitelný, modifikovatelný, a spravovatelný.

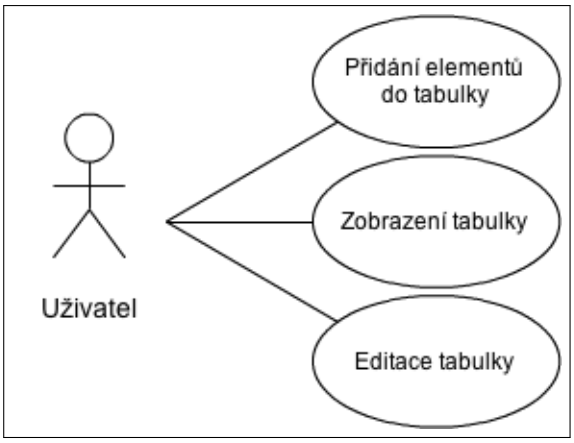
2.3 Případy užití

Tato sekce obsahuje výběr nejdůležitějších případů užití.

2.3.1 Zadávání grafu pomocí tabulky

Prací s tabulkou je myšleno přepnutí z vizuální podoby grafu do jeho textové reprezentace. Ta bude uložena v tabulce, kterou bude možno editovat, pro jednodušší a efektivnější přidávání hran a uzlů. Tato tabulka bude použita pro export do souboru csv.

Na obrázku (Obr.2.1) jsou tři případy užití. Přidání elementů do tabulky pokrývá přidání elementu (v textové reprezentaci) do tabulky. Tabulku bude možné zobrazit a také ji editovat. Při úpravě grafu v interaktivním editoru se promítne změna i do této tabulky a obráceně. Nově přidaný uzel v tabulce se zobrazí v levém horním rohu kreslící plochy.

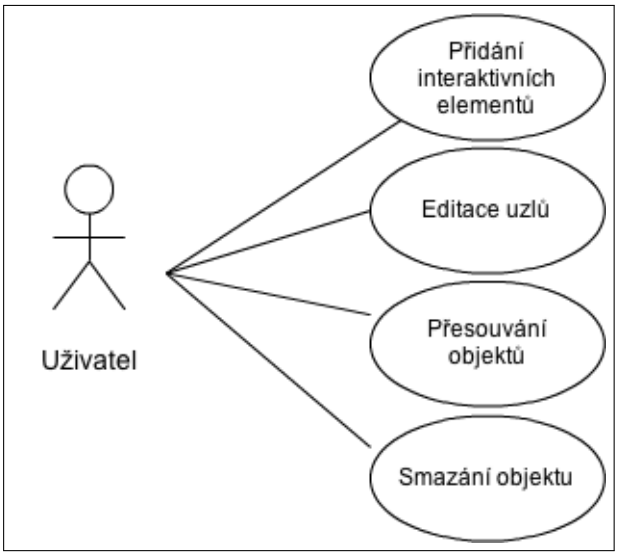


Obrázek 2.1: Práce s tabulkou

2.3.2 Práce s grafem

Pro uživatele, který s výslednou aplikaci bude pracovat poprvé, bude tato vlastnost stěžejní. Interaktivní nakreslení grafu velmi usnadní práci a je velmi intuitivní.

Obrázek 2.2 popisuje čtyři základní případy užití pro práci s grafem. Prvním a nejdůležitějším je přidání interaktivních elementů z panelu prvků. Tyto prvky bude možné přesouvat a mazat. Konkrétně se jedná o uzly a počátek grafu. Hrany se budou tvořit automaticky při spojení dvou uzlů myší. Zároveň bude přidána možnost editace uzlů a jejich vlastností (pojmenování, informace, ...) v pravém panelu.

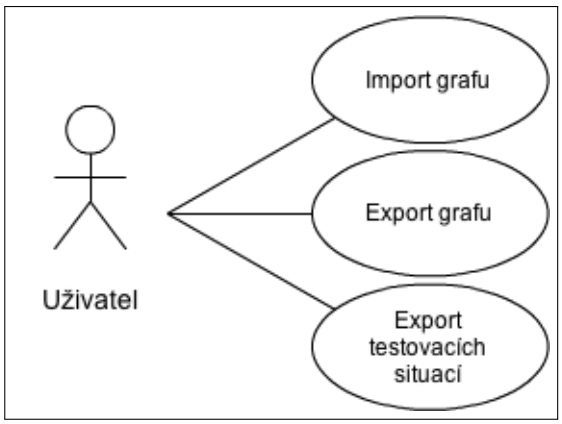


Obrázek 2.2: Práce s grafem

2.3.3 Import/export grafu a testovacích situací

Pro lepší manipulaci s daty, úpravu a jejich zálohování, je přidána funkcionalita importu/exportu. Přinese uživateli možnost vyexportovat data do běžných formátů, která bude moci editovat v programech typu MS Excel, Textový editor, apod. A následně je bude moci importovat zpět do aplikace.

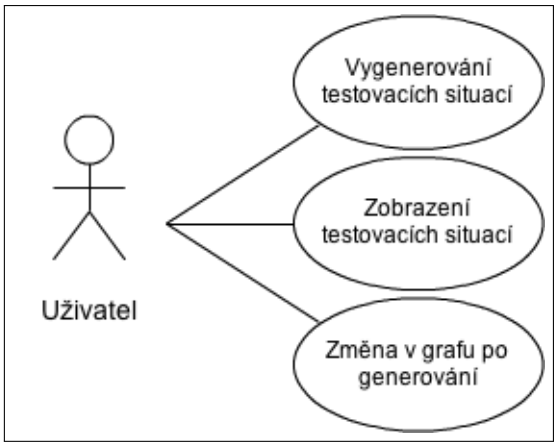
Diagram (Obr.2.3) znázorňuje základní funkce importu a exportu dat z aplikace. Veškeré importy a exporty budou ve formátech CSV a XLS. Soubory se budou lišit pouze strukturou a vlastními daty.



Obrázek 2.3: Import/export

2.3.4 Generování testovacích situací

Hlavním účelem celé aplikace je generování testovacích situací. A právě diagram na Obr. 2.4 popisuje s ním spojené funkce. Samotné generování obstarává první případ užití. Vygenerovaná data bude potřeba zobrazit a k tomu poslouží druhý případ užití. Tabulka se zobrazenými testovacími situacemi bude umožňovat zvýraznění cest v grafu, pro konkrétní testovací situaci. Dále bude uživatel moci měnit strukturu grafu, což způsobí barevné zvýraznění již vygenerovaných testovacích situací (indikace toho, že vygenerovaná data už nejsou aktuální).



Obrázek 2.4: Testovací situace

2.4 Uživatelské scénáře

Níže uvedené uživatelské scénáře popisují interakce uživatele s aplikací. Prerekvizity určují, co musí být splněno před vykonáním samotného scénáře.

2.4.1 Přidání elementů do tabulky

Prerekvizity :

- V aplikaci je alespoň jeden projekt a v něm vybraný graf.

Kroky uživatelského scénáře :

1. Uživatel přidá do tabulky jeden z elementů, který je textově reprezentován.
2. Systém uloží textovou reprezentaci tohoto objektu do paměti (aktualizuje se) a propaguje tuto změnu do vizuální podoby grafu.

2.4.2 Zobrazení a editace tabulky

Kroky uživatelského scénáře :

1. Systém zobrazí tabulku.
2. Uživatel vybere konkrétní buňku tabulky a edituje jí.
3. Systém uloží provedené změny do paměti a upraví graf podle těchto změn.

2.4.3 Přidání interaktivních elementů

Prerekvizity :

- V aplikaci je alespoň jeden projekt a v něm vybraný graf.

Kroky uživatelského scénáře :

1. Uživatel vybere jeden objekt z výběru interaktivních prvků a umístí ho do kreslící plochy.
2. Systém uloží reprezentaci tohoto objektu do paměti a vykreslí požadovaný prvek.

2.4.4 Editace uzlů a hran

Prerekvizity :

- Plocha obsahuje alespoň jeden uzel k editaci.

Kroky uživatelského scénáře :

1. Uživatel vybere daný uzel levým tlačíkem myši.
2. Systém zobrazí v pravém panelu různá pole pro editaci buňky.
3. Uživatel je vyplní a uloží stiskem tlačítka „Uložit“.

2.4.5 Přesouvání objektů

Prerekvizity :

- Plocha obsahuje alespoň jeden objekt.

Kroky uživatelského scénáře :

1. Uživatel vybere stiskem myši nějaký objekt a tahem myši přesune na jiné místo.
2. Systém vykreslí objekt na jiném místě.

2.4.6 Smazání objektu

Prerekvizity :

- Plocha obsahuje alespoň jeden objekt.

Kroky uživatelského scénáře :

1. Uživatel vybere stiskem myši nějaký objekt.
2. Systém zvýrazní vybraný element.
3. Uživatel klikne pravým tlačítkem myši na objekt.
4. Systém smaže objekt a jeho vazby z editoru.

2.4.7 Import grafu

Prerekvizity :

- V aplikaci je alespoň jeden oteřený a vybraný projekt.

Kroky uživatelského scénáře :

1. Systém se dotáže na vybrání souboru k importu.
2. Uživatel vybere soubor k importu.
3. Systém vykreslí naimportovaný graf a aktualizuje tabulku.

2.4.8 Export grafu a Export testovacích situací

Prerekvizity :

- Na ploše je validní graf pro export.

Kroky uživatelského scénáře :

1. Systém zobrazí podokno s výběrem názvu exportovaného souboru.
2. Uživatel zadá požadovaný název a místo k uložení na disku.
3. Systém provede export dat.

2.4.9 Vygenerování a zobrazení testovacích situací

Prerekvizity :

- Na ploše je validní graf pro generování testovacích situací.

Kroky uživatelského scénáře :

1. Uživatel klikne na tlačítko „Vygenerovat“ a vybere úroveň hloubky testování.
2. Systém zobrazí tabulku s vygenerovanými testovacími situacemi.
   (a) Systém zobrazí tabulku s vygenerovanými testovacími situacemi.
   (b) Systém zobrazí chybovou hlášku o nekonzistenci grafu.

2.4.10 Změna struktury grafu po vygenerování testovacích situací

Kroky uživatelského scénáře :

1. Uživatel si zobrazí testovací situace a provede operaci nad grafem.
2. Systém označí tabulku červenou barvou.

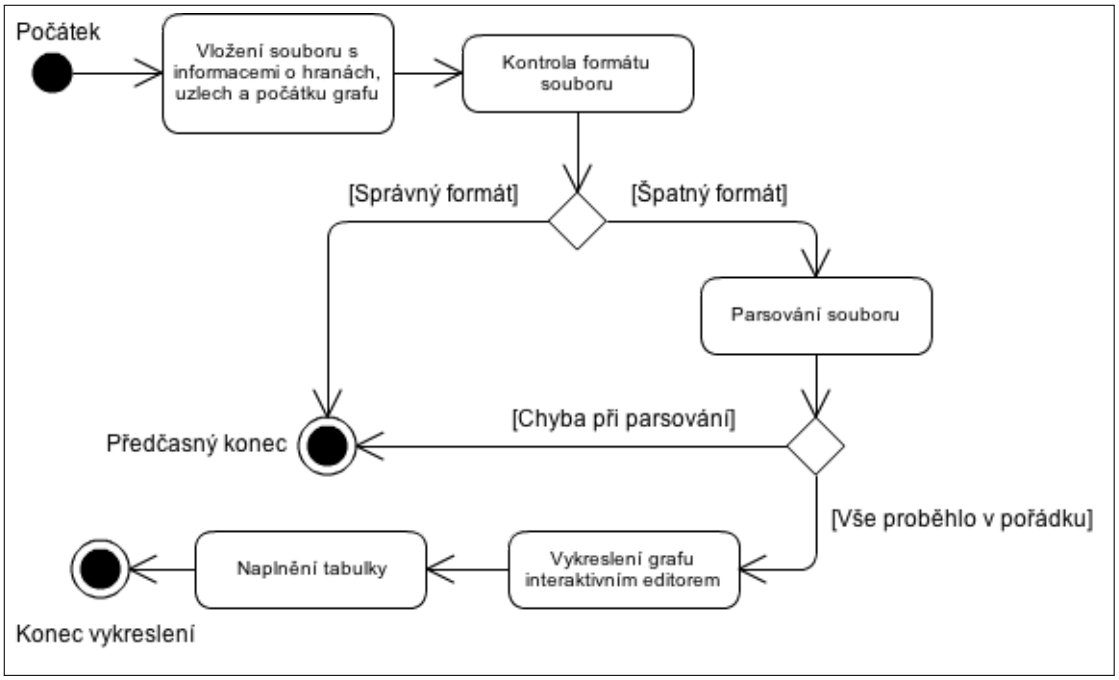
2.5 Základní procesy v aplikaci

V této podkapitole jsou znázorněny a popsány dva nejdůležižtější procesy v aplikaci.

2.5.1 Import grafu

Prvním vybraným procesem je import grafu a jeho vykreslení. Grafické znázornění procesu je na obrázku 2.5.

Prvním krokem je kontrola formátu souboru. Pokud se neshoduje s požadovaným formátem, je proces u konce. Po vložení správného souboru se provede parsování (vytvoření grafu z xml, vytvoření projetku z xml, ...) souboru. Pokud nastane chyba (chybějící atributy u elementů apod.), proces se ukončí. Když proběhne i parsování v pořádku, nic nebrání k vykreslení grafu a naplnění tabulky daty (ta se plní automaticky při „skládání“ grafu, protože jsou vzájemně propojeny).



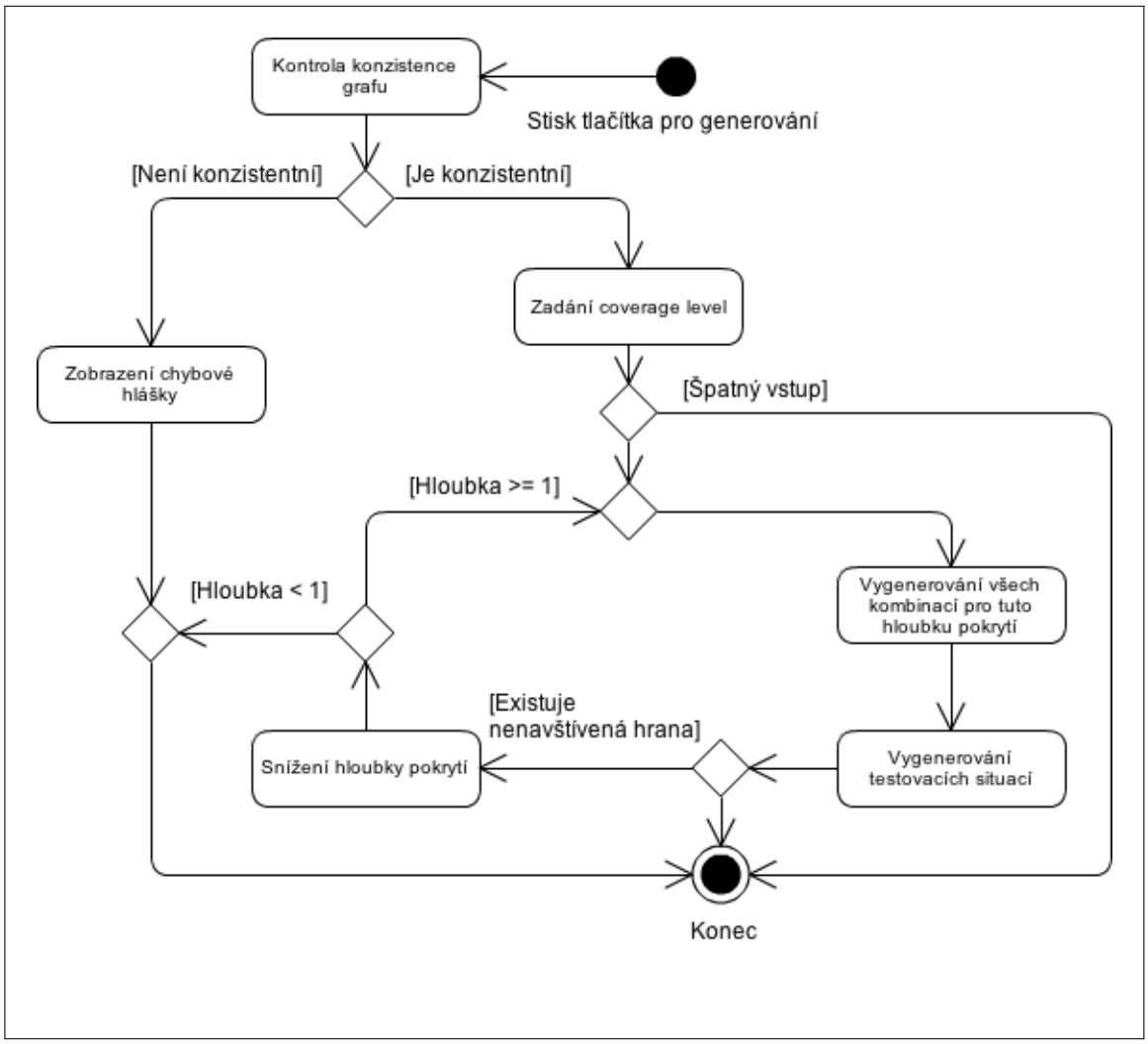
Obrázek 2.5: Proces vykreslení grafu

2.5.2 Generování testovacích situací

Druhým procesem je generování testovacích situací, což je základní funkcionalita aplikace. Jeho proces je graficky znázorněn na obrázku 2.6.

Před samotným začátkem generovacího algoritmu se musí zkontrolovat konzistence grafu (co vše se kontroluje je v sekci 4.5.1). Pokud kontrola selže, zobrazí se chybová hláška a celý proces končí. Jestliže je graf konzistentní, je uživatel vyzván k zadání hloubky pokrytí (coverage level). Zde je provedena kontrola vstupních hodnot, přičemž povolena jsou pouze

čísla větší než nula. Po zadání hloubky pokrytí je spuštěn algoritmus pro generování testovacích situací. Nejprve se tedy vygenerují všechny kombinace a z nich testovací situace. Pokud po vygenerování testovacích situací existuje nějaká hrana grafu, která není obsažena v těchto kombinacích (to může nastat, pokud je zadaná hloubka vyšší, než délka některé cesty v grafu), dochází ke snížení hloubky pokrytí a algoritmus se opakuje dokud není hloubka pokrytí menší než 1.



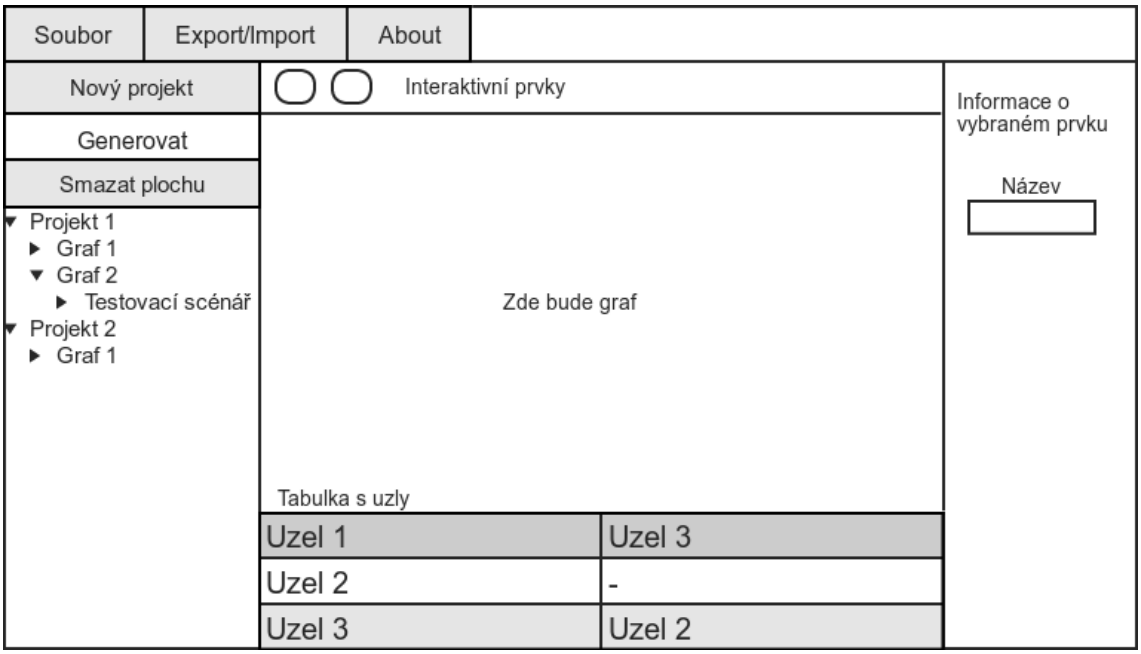
Obrázek 2.6: Proces generování testovacích situací

2.6 Návrh uživatelského rozhraní aplikace

Hlavní obrazovka obsahuje především plochu pro kreslení grafů. Ta je také největším prvkem v celé aplikaci. Nad kreslící plochou budou interaktivní elementy, které bude možné přesunout na kreslící plochu. Pod kreslící plochou bude tabulka obsahující aktuální seznam uzlů a hran, které se náchází na kreslící ploše. Vpravo bude panel pro editaci metadat (název,

informace, ...) a tlačítko pro jejich uložení. Vlevo se bude nacházet panel s adresářovou (projektovou) strukturou. Nad projekty se bude nacházet malý panel s nejpoužívanějšími funkcemi (vytvoření nového projektu, generování, ...). A v horní liště bude menu pro ostatní operace v programu.

Náčrt kreslící plochy a ovladatelných prvků s ní spojených lze vidět na obrázku 2.7. Pro tvorbu tohoto náčrtu byl využit online nástroj mockingbird [18], který lze využít zdarma pro tvorbu jednoho projektu.



Obrázek 2.7: Návrh hlavní obrazovky

Kapitola 3

Algoritmus pro generování testovacích situací

3.1 Úvod

Tato kapitola se zabývá algoritmem pro generování testovacích situací. Nejprve bude popsán algoritmus, který by mohl být aplikovatelný pro ruční počítání. V další podkapitole dojde k výběru vhodného algoritmu, který by umožnil ruční počítací algoritmus převést do pseudokódu, podle kterého bude provedena implementace.

Jelikož budeme v této kapitole pracovat s grafem, je nutné ho nadefinovat z hlediska teorie grafů. Jde tedy o cyklický orientovaný graf, tedy o graf jehož hrany mají určený směr a umožňují vytvoření cyklu.

3.2 Pokrytí cest v aplikaci

Procesy aplikace popsané activity diagramem mouhou mít jeden či více různých průchodů. Jejich počet záleží na složitosti testované aplikace a množství větvení v rozhodovacích bodech.

Jde o to, jak zkombinovat sekvence akcí za sebou, aby byla účinnost testů co nejvyšší a zároveň, aby jich bylo co nejmíň. Toho docílíme tak, že se u jednotlivých průchodů snažíme o minimální počet kroků testu a zároveň o pokrytí všech akcí, které potřebujeme otestovat.

K vytvoření potřebných průchodů nám pomůže metoda test depth level N, neboli hloubka pokrytí N. Jde o metodu dle TMap Next (viz úvod 1.2), která zajišťuje, že jsou pokryty všechny kombinace N na sebe navazujících akcí.

3.2.1 Ruční generování testovacích situací

Vytvoření testovacích situací s hloubkou pokrytí 2 si ukážeme na příkladu, viz obrázek 3.1. Jde o UML activity diagram, který popisuje průběh placení vydražených děl v aukční síni. Jedinou důležitou zmínkou k diagramu je to, že faktura na zaplacená díla se váže vždy

pouze k jednomu období. Tento diagram je vhodné převést do zjednodušené struktury, jak již bylo řečeno v úvodu 1.3. Výsledný orientovaný graf můžeme vidět na obrázku 3.2.

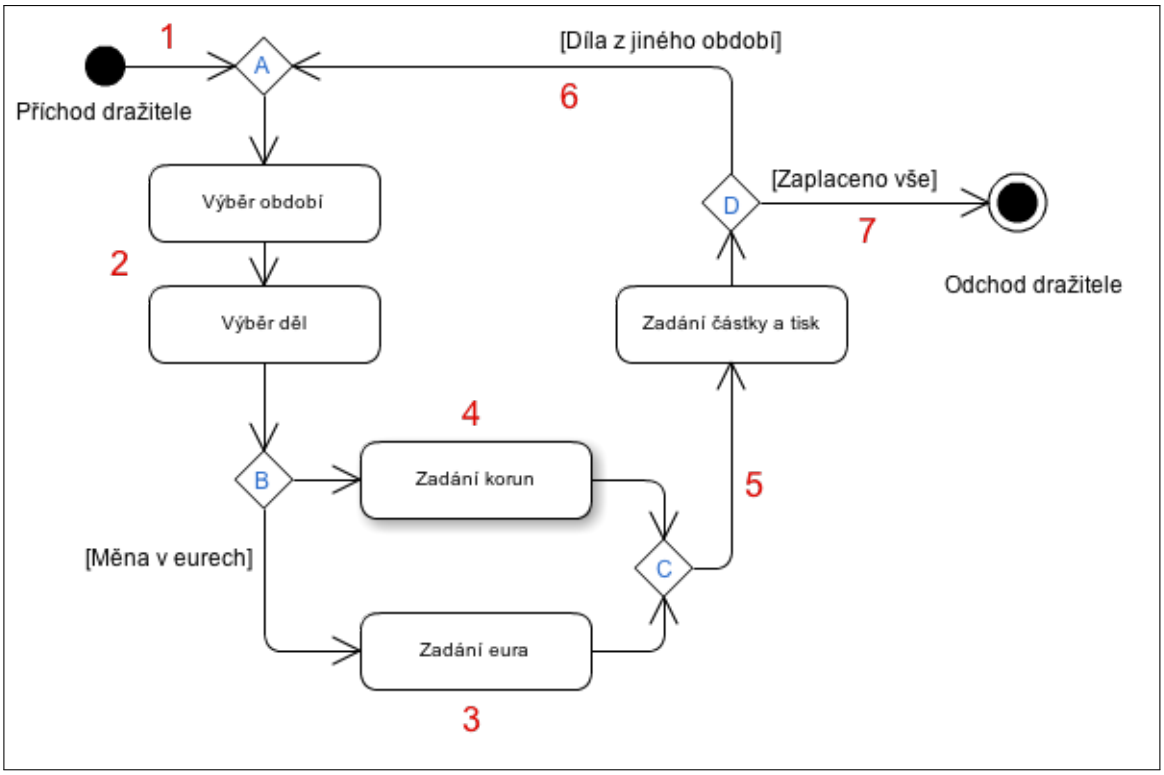
Nyní pro každý uzel stanovíme kombinace, tak jak nám říká tabulka 1.1. Pro jednotlivé uzly tedy :

- A : (1-2), (5-2)
- B : (2-3), (2-4)
- C : (3-6), (4-6)
- D : (6-7), (6-5)

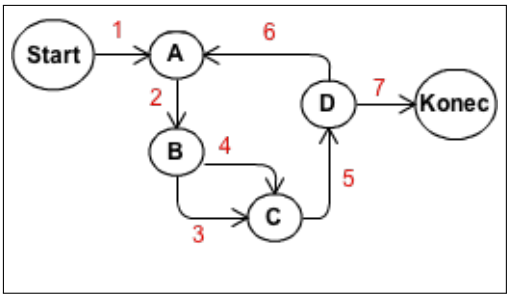
Teď potřebujeme projít všemi nalezenými kombinacemi a zároveň se snažíme o minimální počet kroků. Začneme tak, že zvolíme první kombinaci, tedy (1-2) a určíme ji jako nalezenou (přeškrtneme ji). Nyní máme tedy tyto kombinace ~~(1-2)~~, (5-2), (2-3), (2-4), (3-6), (4-6), (6-7), (6-5). Podíváme se, které kombinace začínají tam, kde naše posledně přešrktlá končí - v hraně 2. Vybereme (2-3) protože jsme ji ještě nenašli (není přeškrtlá). Zbývají ~~(1-2)~~, (5-2), ~~(2-3)~~, (2-4), (3-6), (4-6), (6-7), (6-5). Pokračujeme hranou 3, vybereme podobně kombinaci (3-6) a poté (6-7). Tím jsme došli až do konce a vznikl nám první průchod **(1-2-3-6-7)**.

Pořád nám ještě zbývají volné kombinace ~~(1-2)~~, (5-2), ~~(2-3)~~, (2-4), ~~(3-6)~~, (4-6), ~~(6-7)~~, (6-5). Vezmeme další, např (2-4). Jak vidíme, tato kombinace nezačíná v počátku, proto je potřeba rozdělit hledání do dvou částí - před kombinací a po kombinaci. Z (2-4) se do počátku dostaneme pomocí (1-2). Tím máme jednu část hotovou. Druhou část budeme dělat stejně, jako doposud. Vezmeme kombinaci (4-6), ta pokračuje do kombinace s počátkem v 6, jelikož jsme (6-7) již našli, musíme zvolit jinou (pokud jiná existuje). Zvolíme tedy (6-5). Aktuální stav kombinací je následující ~~(1-2)~~, (5-2), ~~(2-3)~~, ~~(2-4)~~, ~~(3-6)~~, ~~(4-6)~~, ~~(6-7)~~, ~~(6-5)~~. Hranou 5 nám začíná pouze kombinace (5-2), takže ji zvolíme a do konce projedeme jakoukoli cestou. Výsledná testovací posloupnost bude : **(1-2-4-6-5-2-3-6-7)**.

Tímto způsobem jsme dosáhli dvou testovacích situací pro hloubku pokrytí 2. Tato hladina pokrytí je vhodná pro střední úroveň testování. Pro vyšší pokrytí by se použila větší hloubka pokrytí. Ta by byla pro ruční počítání už značně obtížná.



Obrázek 3.1: Activity diagram aukčního systému



Obrázek 3.2: Graf převedený z activity diagramu

3.3 Výběr algoritmu

Aplikace využívá algoritmus založený na prohledávání do hloubky (DFS[1] algoritmus). V každém uzlu se bude procházet do určité hloubky, dle nastavené hloubky pokrytí. Tyto průchody odpovídají testovacím kombinacím pro *Process Cycle Test*. Každá z těchto kombinací musí být obsažena ve výsledné sekvenci testu. Teoreticky založený algoritmus byl popsán výše (sekce 3.2) a níže bude popsán v pseudokódu. Nejprve znázorníme algoritmus pro vytvoření kombinací a poté vytvoření test situations z těchto kombinací.

[1]Depth first search

3.4 Algoritmus pro generování kombinací

3.4.1 Princip

Algoritmus, který je založen na pracích [13] a [11], je převzat ze skript [8]. Algoritmus je upraven v tom smyslu, že nepotřebuje inicializaci uzlů (nemusíme uzel označovat jako nenalezený), protože budeme prohledávat vždy do konkrétní hloubky (depth level). Proto nám nehrozí zacyklení, nebo jiné problémy s tím spojené.

Funkce `GenerateCombinations` (Algoritmus 1) má tři parametry d, N a T. d značí hloubku pokrytí, N pole se všemi uzly a T zásobník. Tato metoda tedy pouze volá metodu `GoToDepth` pro každý uzel a vždy předtím vynuluje zásobník, do kterého se ukládají jednotlivé kombinace. Druhá metoda, `GoToDepth` (Algoritmus 2) bere opět tři parametry, které jsou r, d a T. Jiná je pouze proměnná r, která značí počátek (root). Tato metoda tedy rekurzivně prohledává jednotlivé větve do určité hloubky a k tomu používá právě algoritmus DFS (depth first search).

3.4.2 Algoritmus

Algoritmus 1 GenerateCombinations(d, N, T)

```
1: for n ∈ N do
2:     INIT_STACK(T)                          ▷ nastav prázdný zásobník
3:     GoToDepth(n, d, T)                     ▷ zavolej vlastní generování kombinací
4: end for
```

Algoritmus 2 GoToDepth(r, d, T)

```
1: d = d − 1                                  ▷ sníží hloubku
2: if d < 0 then
3:     SAVE_STACK(T)                          ▷ uloží zásobník (kombinaci)
4:     return   ▷ vrátí se z rekurze, aby mohl pokračovat v prohledávání dalších větví
5: end if
6: O = OUTGOING(r)                            ▷ naplní pole O vystupujícími hranami uzlu r
7: for o ∈ O do
8:     PUSH(T, o)                             ▷ uloží hranu na zásobník
9:     e = END(o)                             ▷ uloží do proměnné e konec hrany o
10:    GoToDepth(e, d, T)                     ▷ zavolá rekurzivně s novým uzlem
11:    POP(T)                                 ▷ odstraní poslední prvek ze zásobníku
12: end for
```

3.5 Algoritmus pro generování testovacích situací

3.5.1 Princip

Algoritmus už byl zmíněn v 3.2.1 a níže je uveden jeho pseudokód. Algoritmus se spustí voláním metody `GenerateTestSituations`, která bere tři parametry. Prvním z nich je s

určující počátek průchodu a e konec průchodu. K značí pole s vygenerovanými kombinacemi.

3.5.2 Algoritmus

Algoritmus 3 GenerateTestSituations(s, e, K)

```
1: INIT_ARRAY(A)                ▷ inicializuje pole, které bude obsahovat testovací situace
2: for k ∈ K do                 ▷ pro všechny vygenerované kombinace
3:     if NOT_VISITED(k) && IS_POSSIBLE(k,e) then  ▷ pokud jsme kombinaci ještě
   nezpracovali a je možné se dostat do konce
4:         SET_VISITED(k)       ▷ nastaví kombinaci jako navštívenou
5:         INIT_ARRAY(T)        ▷ inicializuje pomocné pole
6:         ADD(T,GetPathBefore(k,K))  ▷ uloží do pole cestu před kombinací
7:         ADD(T,k)             ▷ uloží samotnou kombinaci do pole
8:         ADD(T,GetPathAfter(k,K))   ▷ uloží do pole cestu po kombinaci
9:         ADD(A,T)             ▷ přidá výsledek do úložiště všech průchodů
10:    end if
11: end for
```

V algoritmu (Algoritmus 3) se prochází skrz všechny vygenerované kombinace. Pokud je kombinace nenavštívená a je možné se z ní dostat do konce (pro konkrétní konec e), můžeme pokračovat dál. Označíme kombinaci jako navštívenou a vytvoříme lokální proměnnou, kam uložíme mezivýsledky generování. Voláním metod `GetPathBefore` a `GetPathAfter` vytvoříme jeden průchod v aplikaci. Ten složíme tak, že nejprve uložíme do proměnné T cestu nalezenou před aktuální kombinací k, poté samotnou kombinaci k a nazávěr cestu nalezenou po aktualní kombinaci k. A úplně nakonec jen uložíme tento průchod do pole, které drží všechny průchody aplikací.

Algoritmus 4 GetPathBefore(s, c, K)

```
1: INIT_STACK(S)        ▷ inicializace zásobníku, který bude držet průchod do začátku s
2: SET_STACK(h, getWithoutLast(c))  ▷ nastaví zásobník h podcestou, kterou později
   budeme porovnávat
3: for k ∈ K do
4:     a = getWithoutFirst(k)       ▷ inicializace a kombinací bez první hrany
5:     if a = h then                ▷ pokud se a a h rovnají, došlo k navázání cest
6:         if CONTAINS(C,a) then    ▷ pokud je kombinace a již nalezena
7:             o = findOther(h)     ▷ zkusíme najít jinou nenalezenou
8:             if o ≠ NULL then     ▷ pokud najdeme jinou
9:                 a = o            ▷ nastavíme ji jako aktuální nalezenou
10:                ADD(C,a)         ▷ přidáme ji do nalezených
11:            end if
12:        else
13:            ADD(C,a) ▷ kombinaci a jsme ještě nenašli a proto ji uložíme do nalezených
14:        end if
15:        p = PEEK_LAST(a)         ▷ vezmeme poslední hranu z kombinace
```

Algoritmus 4 GetPathBefore(s, c, K) - pokračování

```
16:         PUSH(S, p)                                  ▷ uložíme hranu do zásobníku S
17:         h = getWithoutLast(a)                       ▷ nastavíme aktuální hledanou podcestu
18:         if GET_START(PEEK_LAST(A)) = s then         ▷ počátek se rovná začátku s
19:             return S                                ▷ vrátíme zásobník s cestou do počátku
20:         else
21:             RESTART_CYCLE▷ nedošli jsme do začátku - algoritmus aplikujeme znovu
22:         end if
23:     end if
24: end for
```

Metoda `GetPathBefore` má tři parametry a to c (aktuální kombinace), s značící začátek (uzel do kterého by se měla metoda dostat) a K obsahující všechny vygenerované kombinace. Metoda prochází všemi kombinacemi a hledá takovou, která se při odstranění první hrany (resp. poslední, protože pracujeme se zásobníkem) rovná podkombinaci c, které byla opačně odebrána poslední hrana. V tomto kroku tedy dochází k napojení cesty, která se blíží k začátku (př. Mějme kombinace k = 1-2-3 a c = 2-3-6. Nyní odebereme z k první hranu, čímž zbyde 2-3 a z c odebereme poslední a zbyde opět 2-3.). Po napojení se zkontroluje, zda jsme tuto kombinaci už použili, pokud ne, uložíme a použijeme. Pokud jsme ji už navštívili, zkusíme najít jinou nenalezenou pomocí funkce `findOther` (ta zde není popsána, protože je velmi jednoduchá a není zdaleka tak důležitá). Když jsme úspěšní a najdeme jinou nenalezenou, uložíme jí do proměnné a a uložíme do nalezených. Pak už stačí uložit poslední hranu z kombinace (př. Z předchozího příkladu bychom uložili hranu 1.) a nastavit nově hledanou podkombinaci do proměnné h. V závěru zkontrolujeme, zda už jsme nahodou nedošli na začátek s, pokud ano, vrátíme zásobník S s cestou do začátku. Pokud ne, algoritmus spustíme znovu.

Algoritmus 5 GetPathAfter(e, c, K)

```
1: INIT_STACK(S)          ▷ inicializace zásobníku, který bude držet průchod do konce e
2: SET_STACK(h, getWithoutFirst(c))   ▷ nastaví zásobník h podcestou, kterou později
   budeme porovnávat
3: for k ∈ K do
4:     a = getWithoutLast(k)              ▷ inicializace a kombinací bez poslední hrany
5:     if a = h && IS_POSSIBLE(a, e) then▷ pokud se a a h rovnají a zároveň je možné
   dojít do konce e, došlo k navázání cest
6:         if CONTAINS(C,a) then                   ▷ pokud je kombinace a již nalezena
7:             o = findOther(h)                    ▷ zkusíme najít jinou nenalezenou
8:             if o ≠ NULL then                    ▷ pokud najdeme jinou
9:                 a = o                           ▷ nastavíme ji jako aktuální nalezenou
10:                ADD(C,a)                        ▷ přidáme ji do nalezených
11:            end if
12:        else
13:            ADD(C,a) ▷ kombinaci a jsme ještě nenašli a proto ji uložíme do nalezených
14:        end if
```

Algoritmus 5 GetPathAfter(e, c, K) - pokračování

```
15:        p = PEEK_FIRST(a)                          ▷ vezmeme první hranu z kombinace
16:        PUSH(S, p)                                 ▷ uložíme hranu do zásobníku S
17:        h = getWithoutFirst(a)                     ▷ nastavíme aktuální hledanou podcestu
18:        if GET_END(PEEK_FIRST(a)) = e then         ▷ počátek se rovná začátku s
19:            return S                               ▷ vrátíme zásobník s cestou do počátku
20:        else
21:            RESTART_CYCLE▷ nedošli jsme do začátku - algoritmus aplikujeme znovu
22:        end if
23:    end if
24: end for
```

Druhá metoda `GetPathAfter` dělá v principu stejnou věc, pouze s dvěma rozdíly. Nehledá cestu do počátku, ale do zadaného konce e. To znamená že operace, které se prováděli ve funkci `GetPathBefore`, jsou prováděny vždy zrcadlově obráceny (např. Místo odebrání poslední hrany, se odebírá první). Navíc je zde přidaná kontrola pomocí funkce `IS_POSSIBLE`, která říká, zda se z dané kombinace k lze dostat do konce e. Tato kontrola zde musí oproti metodě `GetPathBefore` být, protože v grafu může být obecně více konců (zatímco začátek je vždy pouze jeden) a i kdyby hrana nebyla nalezená a bylo by ji možné spojit, nemusela by hrana vést do konce e, čímž bychom se dostali do situace, že by tento algoritmus skončil s nekorektním průchodem, který by končil někde uvnitř grafu a ne v konci.

3.6 Složitost algoritmu

Hledání složitosti algoritmu bylo rozloženo na dvě části. První se zabývá algoritmem (Algoritmus 1) a druhá algoritmem (Algoritmus 3).

První algoritmus má nejhorší časovou složitost $\mathcal{O}(N \times O^d)$. Kde N je počet uzlů v grafu, O je maximální počet vycházejících hran z uzlu a d je zadaná hloubka pokrytí. Toto odvození vychází z toho, že algoritmus prochází všechny uzly a v každém uzlu spouští metodu `GoToDepth`. V této metodě se prochází všechny kombinace dané hloubky. To odpovídá přesně mocnině O^d. Stála by zde za zmínku optimalizace v tom smyslu, že po dosažení určité hloubky už není potřeba dále procházet graf. Pokud zvolíme aktuální hloubku zpracovávaného uzlu h a celkovou hloubku grafu f, lze vypustit prohledávání právě tehdy, když platí $f - h \leq d$, kde d je opět zadaná hloubka pokrytí. Protože už neexistuje cesta do konce grafu, která by mohla mít délku alespoň tak velkou jako d.

Druhý algortimus je delší a bohužel také časově náročnější. Jeho zjednodušená nejhorší časová složitost je $\mathcal{O}(K^4)$, kde K je počet vygenerovaných kombinací. Abychom byli přesnější, museli bychom do složitosti zakomponovat počet hran H, které jsou zásadní při volání metody `IS_POSSIBLE`, protože využívají prohledávání do hloubky, které má v tomto případě složitost $\mathcal{O}(H)$ [8]. Aby tato proměnná měla ve výsledné složitosti nějakou váhu, muselo by platit že je počet hran H asymptoticky vyšší, než K^4.

Pro zjištění celkové asymptotické složitosti je pouze potřeba sečíst tyto dvě složitosti. Jelikož je složitost $\mathcal{O}(K^4)$ asymptoticky větší než $\mathcal{O}(N \times O^d)$, můžeme druhou složitost zanedbat a celková složitost je tedy $\mathcal{O}(K^4)$. To je maximální složitost, vycházející z toho, že

je každý cyklus v algortimu vykonán nejdelší možnou dobu. To ale obecně neplatí a proto je vyjádřena odhadovaná složitost jako $\mathcal{O}(K^3)$. Ta vychází z toho, že je tabulka (`HashTable` v Javě) dobře rozložená a volání metody `get` má konstantní složitost $\mathcal{O}(1)$. Po dohodě s vedoucím práce byla složitost, vzhledem k rozsahu dat, který bude program zpracovávat a složitosti úlohy, akceptována.

Kapitola 4

Implementace

4.1 Úvod

V této kapitole bude nejprve popsána architektura aplikace, proč byla takto zvolena a později je uvedena její aplikace v programu. Následně jsou popsány zvolené technologie a knihovny použité v programu. Také je zde vysvětleno, proč byly vybrány právě tyto knihovny a jejich výhody, či nevýhody. Důkladněji se rozebere framework JGraphX, který je pro gui stěžejním konstruktem. Dále se tato kapitola věnuje konkrétním částem aplikace a vyzdvihuje to nejzajímavější, co se v ní odehrává. Ve vybraných částech této kapitoly jsou také vepsány úryvky zdrojových kódu, ať už přímo z aplikace nebo pouze demonstrační příklady.

4.2 Návrh architektury aplikace

4.2.1 Úvod ke vzoru Model View Presenter

Obecná struktura aplikace je rozdělena na GUI, datový model reprezentující uzly grafu a funkce pracující s těmito daty. Takové rozdělení je běžně používané a není moc zajímavé. Důležitější je vzájemné propojení těchto částí. S tím může pomoct jeden z architektonických vzorů, kterých je celá řada (PAC[1], MVC[2], MVP, ...). Pro tuto aplikaci byla zvolena architektura MVP (model view presenter), která je hojně užívaná v desktopových aplikacích. Její struktura je velmi podobná vzoru MVC (model view controller), což je patrné už z jejich názvů, ve kterých se liší pouze v jednom slově. Ale mají mezi sebou samozřejmě i další rozdíly. Jeden z nich byl hlavním důvodem pro zvolení právě MVP. A to ten, že vzor MVC pracuje typicky s více views. Kdežto tato aplikace má pouze jedno hlavní view, které se nemění.

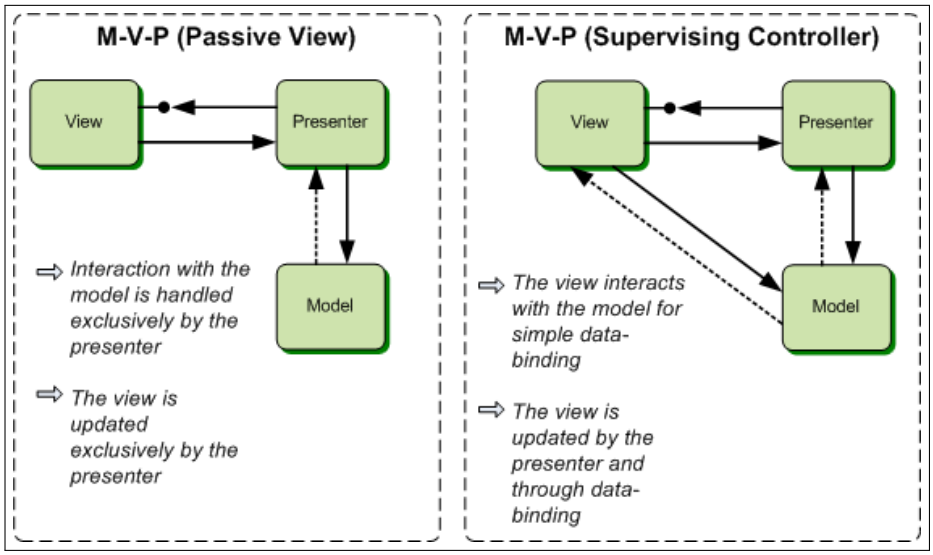
4.2.2 Rozdělení vzoru MVP

MVP se ještě dělí na dva podtypy (obr. 4.1), které se liší v tom, jak moc je view propojeno s presenterem. S rozdělením přišel Martin Fowler a první z nich pojmenoval *Passive View* [4]. Zde se view snaží nestarat téměr o nic a všechny vstupy přeposílá na presenter. Preseneter

[1] Presentation-Abstraction-Control
[2] Model-View-Controller

je jakýsi prostředník mezi view a model. To znamená, že view neví o modelu přímo. Ale presenter poslouchá na změny v modelu a podle toho updatuje view. Tato varianta nepoužívá data binding, ale využívá settery ve view pro předání dat. Druhý typ se jmenuje *Supervising Controller* a hlavním rozdílem je data binding, který umožňuje vzájemné propojení view s modelem. Data binding funguje tak, že pokud se změní model, změní se view a opačně. Toto řešení je sice elegantnější, ale složitější na napsání, má zvýšený coupling a neumožňuje jednoduché testování presenteru. K otestování presenteru je nyní totiž potřeba nejen znalost presenteru, ale i znalost konkrétního view.

Rozdíly těchto dvou vzorů pěkně popisuje obrázek (Obr. 4.1), převzatý ze stránky [12]. Vlevo můžeme vidět typ *Passive View*, na kterém je vidět oddělení view od modelu, takže se interakce mezi nimi děje pouze skrze presenter. Zatímco vedle, u verze *Supervising Controller* vidíme, že mezi view a modelem vazba exituje. A toto projení je realizováno pomocí data bindingu, které bylo zmíněné výše.



Obrázek 4.1: Rozdělení MVP

4.2.3 Princip MVP Passive View

Protože komunikaci mezi komponentami většinou odstartuje nějaký uživatelský vstup, začneme nejprve s view. View přijímá uživatelské požadavky a deleguje je na konkrétní presenter (nebo více presenterů). View je tedy pevně spojeno s presenterem (má na něj odkaz). Presenter zpracuje vstup (získá data z modelu, zpracuje nějaké dalších procesy, ...) a updatuje view prostřednictvím setterů. Presenter je spojen s view skrze interface, což se dělá hlavně kvůli testování. Presenter má dále vazbu na model, aby s ním mohl pracovat.

4.3 Zvolené technologie

4.3.1 Java

Pro tvorbu aplikace byl zvolen jazyk Java, jehož historie sahá až do roku 1991, ale první implementace byla zveřejněna až v roce 1995. Zvolen byl především kvůli jeho nezávislosti na použité platformě (WORA[3]). Pro jeho spuštění stačí zkompilovat kód a spustit ho v JVM[4], který je odpovědný za interpretaci zkompilovaného zdrojového kódu (byte kódu) do strojového kódu. Jde o objektově založený jazyk, který je hojně používaný také díky své jednoduchosti (např. oproti C++ kde se musí o dereferenci proměnných starat programátor[5]) a množství frameworků rozšiřující tuto základní edici [16]. K vytvoření grafického rozhraní je využit Swing.

4.3.2 Swing

Swing je knihovna obsažená od verze 1.2. přímo v Java SE [17]. Slouží k vytváření oken, dialogů, tlačítek a řady dalších zobrazitelných prvků. Někdo by mohl namítnout, proč nebyl vybrán framework JavaFX, který je obsažený v Java SE 8.[6] Protože je novější, podporuje nové trendy a považuje se za nástupce Swingu. Zde jsou dva důvody, proč právě Swing a ne JavaFX. Prvním z nich je ten, že je Swing pořád nejvíce rozšířený a také k němu existuje mnoho příkladů a tutoriálů. Druhým, možná důležitějším důvodem v tomto případě je ten, že framework pro tvorbu grafů (viz sekce 4.4.1) používá také Swing knihovnu.

4.4 Využití znovupoužitelných knihoven

4.4.1 JGraphX

4.4.1.1 Úvod

JGraphX[7] je swingově založený framework umožňující vizuálně reprezentovat UML diagramy. Dokáže pracovat s velkým množstvím různých typů diagramů. Také má podporu pro různé operace s uzly (zvětšování, přesouvání, spojování s hranami, ..) [5]. A je proto vhodnou volbou pro tvorbu aktivity diagramu.

[3] Write Once, Run Anywhere, bylo motto vyřčené už roku 1995 [15].

[4] Java Virtual Machine

[5] V Javě existuje garbage collector, který se stará o „úklid“ po naší práci s objekty a automaticky uvolňuje paměť u objektů, které jsou už nedosažitelné.

[6] JavaFX je v Java SE od verze 7 update 6 a nese označení JavaFX 2.2. Tato verze byla určená pouze pro Windows uživatele a byla pouze přechodným obdobím, dokud nepřišla Java SE 8 v níž je už obsažena JavaFX 8 (začalo se číslovat stejně jako značení Java SE).

[7] Framework do verze 1-5 původně nesl název JGraph, ale v další verzi došlo ke komplenímu přepsání kódu a proto se autoři rozhodili tuto větší změnu zvýraznit přímo v názvu. JGraph Ltd vyvíjí ještě jeden framework (mxGraph), který je psaný v javascriptu a slouží tedy pro webové aplikace. Jedná se ale o placenou variantu.

4.4.1.2 Struktura frameworku

Různé operace s grafem se provádějí skrze třídu `mxGraphModel`. Tato třída popisuje strukturu grafu a zachycuje operace s ním prováděné. Operace, které chceme s grafem provádět (např. přidání dvou uzlů se spojující hranou), se vykonávají v transakcích, ale nevolají se přímo na modelu. Volají se přímo ve třídě `mxGraph`, protože je více intuitivnější pracovat přímo s grafem, než s modelem. Další důležitou třídou je `mxGraphComponent`, která je potomkem `JComponent` a slouží tedy k vizualizaci konkrétního potomka třídy `mxGraph`. Framework je velmi založen na vzoru Observer, takže pokud chceme vyvolat nějakou akci, právě tehdy, když nastane nějaká jiná akce na modelu, stačí zaregistrovat posluchače a ten se vyvolá při výskytu události.

Ukázka toho, jak jednoduše lze vytvořit hranu mezi dvěma uzly, je zobrazena v listingu 4.1. V transakci, mezi metodami `beginUpdate` a `endUpdate` volané na modelu se provedou příslušné operace (vytvoření dvou uzlů a mezi nimi hrana) v samotném grafu.

Listing 4.1: Ukázka JGraphX

```
final mxGraph graph = new mxGraph();
Object parent = graph.getDefaultParent();

graph.getModel().beginUpdate();
try {
    Object v1 = graph.insertVertex(parent, null,
            "First", 30, 30, 60, 40);
    Object v2 = graph.insertVertex(parent, null,
            "Second", 30, 120, 60, 40);
    graph.insertEdge(parent, null, "Edge", v1, v2);
} finally {
    graph.getModel().endUpdate();
}
```

4.4.1.3 Možnosti přizpůsobení

Framework umožňuje celou řadu možností, jak výsledný graf (nebo jeho část) přizpůsobit k obrazu svému a jak si vynutit potřebné chování. Vizuální změny grafu (nebo jeho částí) se řeší pomocí stylů, které si můžeme sami nadefinovat. Automatické rozložení jednotlivých uzlů lze provést jednou z mnoha layoutovacích tříd [7]. Taková funkce je potřebná například při importu grafu v csv, ve kterém nejsou uloženy souřadnice uzlů.

V následujícím kódu (4.2) můžeme vidět metodu, která bere graf jako parametr a vytvoří z jeho uzlů kruhový graf. Metoda nejprve vytvoří layoutovací objekt (`mxCircleLayout`) a poté mezi `beginUpdate` a `endUpdate` spustí samotnou layoutovací funkci execute s defaultně nastaveným rodičem (všechny grafy v aplikaci používají defaultního rodiče).

Listing 4.2: Circle layout

```
/**
 * It circularly lays out nodes of the given graph.
 *
 * @param graph
 */
public static void makeCircular(mxGraph graph) {
    mxGraphLayout layout = new mxCircleLayout(graph);
    graph.getModel().beginUpdate();
    try {
        layout.execute(graph.getDefaultParent());
    } finally {
        graph.getModel().endUpdate();
    }
}
```

4.5 Popis vybraných míst z implementace

4.5.1 Validace grafu

Před samotným generováním testovacích situací je potřeba, aby graf prošel validací. Validačních kritérií je v aplikaci více a aby bylo možné jednoduše přidávat další pravidla, stačí pouze implementovat interface `GraphValidator` (Listing 4.3). Ten má dvě metody `getDescription` a `validate`, přičemž metoda `validate` s parametrem mxGraph vrací boolean a říká, zda je konkrétní kritérium splněno a `getDescription` vrací String s poznámkou, která popisuje proč nelze vygenerovat testovací situace a co je potřeba změnit v grafu. Aplikace obsahuje validátory zajišťující tyto kontroly :

- Graf nesmí být prázdný (musí obsahovat alespoň jeden uzel a start propojený hranou)
- V grafu musí být start
- Start nesmí mít vstupní hrany
- Z každého uzlu se dá dostat do alespoň jednoho konce (uzel z kterého nevychází žádná hrana)
- V grafu musí být alespoň jeden konec
- Uzly musí mít vstupní hranu, kromě startu - ten ji mít nesmí
- Každý uzel musí být dosažitelný ze startu

Každé z těchto kritérií je naimplementováno zvlášť a když chceme provést celkovou validaci, zavoláme na modelu metodu `validate`, která tak jako v `GraphValidator` vrací boolean a bere parametr `mxGraph`. Tato metoda projde přes všechny zaregistrované validátory a pokud nějaký z nich selže (vrátí false), uloží metoda popis chyby do seznamu zpráv. Nakonec metoda vrátí false, pokud alespoň jeden ze zaregistrovaných validátorů selhal, jinak true.

Listing 4.3: Rozhraní graph validator

```
/**
 * Interface which describes the validator for graph.
 *
 * @author Lukas Lowinger <lukasinko@gmail.com>
 */
public interface GraphValidator {

    /**
     * Validates a concrete criterion of the given graph.
     * @param graph
     * @return true if the graph accomplishes
     * the rules, false otherwise
     */
    public boolean validate(final mxGraph graph);

    /**
     * @return description about the rule.
     */
    public String getDescription();
}
```

4.5.2 Použití SwingWorker

4.5.2.1 Vlastnosti

SwingWorker je abstraktní třída z balíčku javax.swing [24], která se používá především ve swingově založených programech (aplikace obsahující gui). Vznikla z toho důvodu, že swingové třídy by měly být spouštěny a upravovány pouze v jeho EDT[8] vlákně. S tím, že malé úpravy se mohou měnit přímo v EDT vlákně. Pokud ale chceme zpracovat složitější výpočet, sáhnout do databáze nebo vykonat jinou časově náročnou operaci, je lepší dělat to skrze třídu SwingWorker. A to z toho důvodu, že nechceme, aby uživatel přišel o kontrolu nad programem. Pokud by totiž byly časově náročné oparece zpracovávány v EDT vlákně, musel by uživatel počkat na dokončení tohoto procesu a do té doby by byla aplikace neovladatelná. Použitím SwingWorkeru se tomu dá vyhnout.

`SwingWorker<T,V>` má dva typové parametry a jeho nejzákladnější metody jsou `publish`, `done`, `doInBackground` a `process`. T je typ který vrací metoda `doInBackground`, oproti tomu V je typ který zpracovávají metody `publish` a `process`. Z toho pouze `doInBackground` je abstraktní a je tedy nutné ji implementovat. Měly by v ní být všechny složité operace, protože SwingWorker zajistí, že se budou zpracovávat ve vlastním vlákně a ne v EDT. Metoda `done` je volána po vykonání `doInBackground` a u ní je zase zaručeno to, že bude zpracována ve vlákně EDT (stejně tak je tomu u metody process, která slouží k aktualizaci stavu, v jakém se operace nachází).

[8] Event Dispatch Thread je vlákno, ve kterém by se měly volat metody na swingových objektech. Swing totiž není „thread safe“, což by mohlo způsobit problémy s konzistencí dat při použití vícevláknovosti [21].

4.5.2.2 Implementace

Typická struktura vypadá tak, že třída dědí od `SwingWorker<String, Integer>` (viz ukázka 4.4). `String` parametr je použit pro získání textu (pomocí metody `get`, který se má zobrazit po dokončení procesu. `Integer` parametr je zvolen pro jednoduché aktualizování progress baru (nebo jiného prvku, který dokáže zobrazovat aktuální stav).
V metodě `doInBackground` se pomocí metody `publish` nastavuje aktuální percentuální stav, podle toho, kolik práce už je hotovo. Metoda `process` zachytí tyto aktualizace a updatuje (ve vlákně EDT, jak bylo řečeno výše) progress bar s posledně nastavenou hodnotou (vynecháno kvůli zkrácení kódu) [1]. Ukázka vytvoření a spuštění SwingWorkeru je v listingu 4.5.

Listing 4.4: Ukázka implementace swing worker

```
class MySwingWorker extends SwingWorker<String, Integer> {

        private JProgressBar bar;
        private JLabel label;

        public MySwingWorker(JProgressBar bar, JLabel label) {
            this.bar = bar;
            this.label = label;
        }

        @Override
        protected String doInBackground() {
            publish(20);
            //do some long process
            publish(99);
            return "DONE";
        }

        @Override
        protected void process(List<Integer> progressList) {
            //updating bar with last published value
        }

        @Override
        protected void done() {
            //try - catch for method get() is omitted
            label.setText(get());
    }
```

Listing 4.5: Spuštění SwingWorker

```
SwingWorker<String,Integer> worker = null;
worker = new SwingWorker<String,Integer>();
worker.execute();
```

4.5.2.3 Použití

V aplikaci je SwingWorker použit ve dvou případech. Prvním z nich je načítání projektu, které by mohlo trvat delší dobu, pokud projekt obsahuje velké množství grafů s vygenerovanými testovacícmi situacemi. Druhým časově náročným procesem může být generování testovacích situací a proto je přidán SwingWorker i sem. Výhodou také je, že uživatel může tyto dlouho trvající procesy přerušit v jejich průběhu.

4.5.3 Import/Export

Import a export ve formátu xml umožňuje přímo JGraphX framework. Výsledná struktura xml neposkytuje možnost ukládat vlastní metadata a zároveň je zbytečně složitá. Proto je import i export naprogramován ručně.

Formát csv je sice jednodušší pro export z aplikace, ale složitější pro import zpět do aplikace. Problém je v tom, jak rozpoznat, zda je uzel „normální“ nebo zda se jedná o počáteční uzel (start). Tyto dva typy uzlů je potřeba pro správnou funkci rozlišit. A protože v csv nejsou ukládána žádna metadata (narozdíl od xml), musí se rozlišit formou přesně zadaného názvu uzlu. Pro start bylo vybráno klíčové slovo „start“, u kterého nezáleží na velikosti písma. Takto označený uzel může být v aplikaci (tedy i v csv) pouze jednou a pokud se toto pravidlo nedodrží, mohou nastat problémy v chování aplikace.

4.5.4 Implementace Model View Presenter

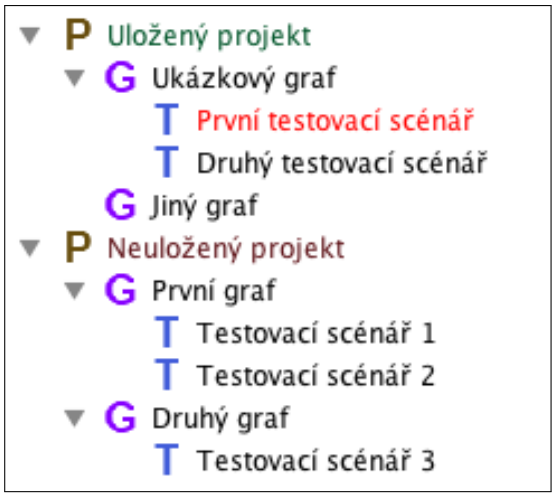
Zvoleným vzorem je tedy Model-View-Presenter ve formě Passive View (viz sekce 4.2.3). Aplikace obsahuje jedno hlavní view (třída `MainGui`), které má v sobě vazbu na preseneter (třída `MainGuiPresenter`) a ten obsluhuje uživatelské vstupy. Takovýchto struktur je v aplikaci více, ale už v menším provedení. Každé větší view má svůj presenter (pokud je potřeba). Za zmínku stojí ještě delegace akcí z `MainGuiPresenteru` na `ProjectController`, který zprostředkovává funkce pracující s projektovou strukturou v levém panelu.

4.6 Přidaná rozšíření aplikace

4.6.1 Struktura projektů

Aplikace pracuje na bázi projektů. Každý projekt může obsahovat více grafů a každý graf může obsahovat více testovacích scénářů (to lze vidět na obrázku 4.2). Aplikace umí tyto projekty ukládat ve formátu xml s příponou „prj“. Ta je zvolena proto, aby se snadněji odlišily od klasických souborů s koncovkou „xml“, které jsou použity u souborů s grafy. Aplikace dále umí tyto soubory načítat z disku. Po načtení projektu z disku je projekt označen jako uložený (podbarvený zelenou barvou). Pokud se v něm provede jakákoli změna, označí se hnědě - tak jsou obarveny i nově založené projekty, protože ještě nejsou uloženy na disk.

Každý projekt, graf a testovací scénář obsahuje nějaké meta informace. Konkrétně projekt nese název, popis, vazbu na dokumentaci a verzi projektu. Graf má název, popis a také vazbu na dokumentaci a testovací scénář je rozšířen o názvev a poznámky.



Obrázek 4.2: Struktura projektů

4.6.2 Funkce pro redukci cyklů

Po zkoumání algoritmu pro generování testovacích situací, jsme spolu se školitelem narazili na situaci, kdy se na první pohled algoritmus nezdá být optimální. Taková situace nastává, když je v grafu obsažena kružnice a je zadaná taková hloubka pokrytí, že je tato kružnice ve výsledné testovacím průchodu obsažena dvakrát (nebo vícekrát) bezprostředně po sobě.

Proto byla přidána do aplikace funkce, kde si uživatel může před generováním testovacích průchodů vybrat, zda chce použít algoritmus pro redukci cyklů. Zároveň je upozorněn, že redukcí dojde ke ztrátě pokrytí, které definuje test depth level N.

Kapitola 5

Testování

5.1 Úvod

Předposlední kapitola popisuje metody použité k otestování chodu aplikace a použitého algoritmu. Prvními, nejzákladnějšími testy, jsou jednotkové testy. Ty by se měly používat pokud možno ve všech aplikacích a měly by být implementovány od začátku vývoje aplikace. Čím dříve se totiž na chyby v malých částech aplikace přijde, tim snazší je jejich oprava. Dalším typem testu je funkční testování, které vychází z uživatelských scénářů a k jejich otestování jsou potřeba lidi, kteří tyto scénáře projdou a vyzkouší. Posledním typem testu k aplikaci je otestování funkčnosti alogritmu. To bylo provedeno ručně a jeho princip je popsán v poslední podkapitole.

5.2 Jednotkové testy

5.2.1 Úvod

Pro testování jednotek (nejmenších prvků aplikace) byl použit framework JUnit 4.10. Vhodným pomocníkem pro generování testovacích tříd ve spojení s JUnit je Netbeans IDE (Integrated Development Environment), které umožňuje generovat testovací třídy se všemi nesoukromými metodami obsažené v konkrétní testované třídě. Tím odpadá značná práce pro přepisování všech metod. JUnit projde všechny metody označené anotací @Test a vyhodnotí v nich volání přislušné assert[1] metody.

Aplikace je pokrytá 25 JUnit testy, které testují funkčnost jednotlivých metod (byly vybrány ty nejdůležitější). Konkrétní otestované třídy jsou zmíněny níže.

5.2.2 Pokrytí aplikace jednotkovými testy

Testy pokrývají ty nejzásadnější metody ve vybraných třídách (třídy, které jsou důležité pro splnění případů užití nebo ty, které obsahují složitější metody). Byly vybrány tyto třídy:

[1] V javě je assert příkaz, který vyhodí vyjímku, pokud je parametr false [20]. Pokud je člověk chce spustit, musí to explicitně nastavit v JVM. JUnit pracuje s metodami označené klíčovým slovem „assert“, které dokáží vyhozenou chybu blíž popsat.

`MainGuiPresenter` pro pokrytí komunikace presenteru s gui, `MainGui` pro otestování komunikace gui s jejími gui komponentami, `ProjectTree` pro otestování přidávání a odebírání prvků z panelu s projekty, skupina potomků třídy `AbstractRightPanel` pro otestování předávání a aktualizace metadat, v hiearchii `TreeNode` nejvýše postavná třída `ProjectTreeNode` a pro otestování funkčnosti tabulky v grafu byla otestována třída `EditTableModel`.

5.3 Funkční testy

5.3.1 Princip testování

Funkční testování testuje vybrané scénáře, které mohou v aplikaci nastat. Tyto zvolené scénáře jsou předány testerům (uživatelům, kteří mají zájem o testování beta verze aplikace), kteří ve dvou kolech zkoušejí procházet aplikaci dle scénářů. Dvě kola jsou zvolena proto, aby byly odhlany jak majoritní problémy, tak minoritní problémy.

5.3.2 Průběh testování

Oslovení uživatelé (dva kamarádi + dva rodinní příslušníci) dostali 15 uživatelských scénářů (průchodů aplikací), které měli v prvním kole splnit. Po tomto prvním kole, byly odhaleny 4 chyby, jmenovitě :

- Změna názvu v uzlu se nepromítla do tabulky
- Při změně grafu po vygenerování testovacích situací se její podbarvení neprovedlo okamžitě, ale až když ručně uživatel provedl jiné změny v panelu s projekty
- Po vyexportování grafu do csv a následném importu nebyl graf správně zobrazen (uzly se překrývaly)
- Po smazání uzlu či hrany zůstal pravý panel pořád viditelný

Po odstranění nalezených problémů byli uživatelé požádáni znovu, aby prošli stejné scénáře. Po druhém kole už žádné chyby nenalezli.

5.4 Testy správnosti generování testovacích případů

5.4.1 Princip testování

Metoda testování správnosti algoritmu spočívá v porovnání ručně vytvořených průchodů z grafů s vygenerovanými situacemi z aplikace.

Nejprve bylo vytvořeno 12 grafů, které jsou od sebe odlišné. Každý graf měl jiné parametry, jakými jsou například množství uzlů, počet cyklů a počet koncových uzlů. Všechny grafy splňovaly minimální vlastnosti určené tabulkou 5.1.

Z těchto grafů byly poté s pomocí školitele vytvořeny ručně průchody (podle metody popsané v sekci 3.2.1). Průchody byly vytvořeny pro hloubky pokrytí 1,2 a 3. Větší hloubka pokrytí se už ručně těžce počítá a dle instrukcí školitele je hloubka 3 maximální test depth

Vlastnost	Minimální počet
Uzly	4
Hrany	6
Koncové uzly	1
Cykly	0
Různé cesty	2

Tabulka 5.1: Minimální vlastnosti testovaného grafu

level N, která se při generování testovacích situací v praxi používá. Tím tedy vznikla jedna část pro srovnání. Pro druhou část se tyto grafy překreslily do aplikace, ve které se provedlo generování opět pro hloubky 1-3.

Následovalo ruční srovnání teoreticky odvozených průchodů s vygenerovanými průchody z aplikace. Tato kontrola vyžadovala důkladné zkoumání, zda jsou průchody opravdu „stejné“. Protože obecně existuje více variant průchodů, nemuselo se ruční počítání přesně shodovat s programovým počítáním.

5.4.2 Důsledek testování

Touto metodou testování se v průběhu implementace přišlo na několik chyb a neoptimalit v algoritmu. Například u jednoho grafu, který obsahoval dva cykly hned za sebou, byla nalezena chyba v zacyklení. Tyto chyby byly následně odstraněny a aplikace přetestována.

Kapitola 6

Závěr

6.1 Dosažení cílů

Zadáním práce bylo vytvořit desktopovou aplikaci, která bude generovat testovací situace dle metody *Process Cycle Test*. Toho se podařilo dosáhnout a dokonce bylo toto zadání rozšířeno o pár dalších vlastností (projektová struktura, algoritmus pro redukci cyklů, ...). Aplikace byla ve výsledku napsána tak, aby byla snadno rozšiřitelná (např. pomocí interface[1]) a modifikovatelná, čehož se dosáhlo především důkladným refactoringem[2]. Zároveň byl kladen důraz na využití nějaké znovupoužitelné knihovny, protože není správným přístupem psát znovu to, co už bylo někým jiným napsáno. Na druhou stranu se najdou i případy, které nebyly vyřešeny naprosto dokonale a šlo by je určitě vylepšit.

Největším přínosem pro mě byla práce s frameworkem. Konkrétně s frameworkem JGraphX a obecně jeho využití a schopnost nastudovat jeho funkcionality. Pokud se v budoucnu setkám s jiným frameworkem, už budu vědět, kde a jak hledat informace, což by mi mohlo ušetřit čas. Přineslo mi to i zkušenost ve čtení a porozumění cizího javadocu[3]. Dále jsem díky tomu vyzkoušel, jak vytvářet pomocí již implementovaných metod nové funkcionality.

Nakonec jsem si uvědomil, že je dobré občas někomu i poradit. A proto jsem se zaregistroval na webu stackoverflow [19], jehož komunita se snaží zodpovídat na položené dotazy (důraz je kladen na otázky týkající se programování). A na tomto webu jsem se díky znalostem, kterých jsem při psaní práce nabyl, připojil do komunity.

6.2 Možnosti dalšího rozšíření

Aplikace vzniklá díky této bakalářské práci by mohla být v budoucnu rozšířena i o další metody generování testovacích scénářů z procesů v aplikaci. Dále by aplikace mohla být obohacena o více uživatelsky přívětivé funkcionality, jakými jsou například automatické ukládání projektů a tisk testovacích scénářů do pdf.

Další možností, která by mohla program zlepšit (ne rozšířit), by mohlo být předpočítávání časově náročných operací. Například ukládat k uzlům informaci o tom, zda z nich

[1] Rozhraní v javě.
[2] Úprava a optimalizace zdrojového kódu.
[3] Strukturované komentáře ve zdrojovém kódu.

existuje cesta do konce. Takové procesy by mohly běžet na pozadí za chodu aplikace, aby výsledné generování s využitím předpočítaných hodnot bylo časově méně náročné. Nicméně k očekávanému rozsahu zpracování dat, není implementace této funkčnosti pravděpodobně nutná.

Literatura

[1] ATTARD, A. *Swing worker exapmle* [online]. 2013. [cit. 6. 4. 2014]. Dostupné z: <`http://www.javacreed.com/swing-worker-example/`>.

[2] BUREŠ, M. Techniky pro návrh manuálních testů : průchody programem. Přednáška předmětu TS1 na ČVUT, 2013.

[3] ECKEL, B. *Myslíme v jazyku Java: knihovna zkušeného programátora.* Grada, 2001. ISBN 80-247-0027-1.

[4] FOWLER, M. *GUI Architectures* [online]. 2006. [cit. 4. 3. 2014]. Dostupné z: <`http://martinfowler.com/eaaDev/uiArchs.html`>.

[5] JGraph. *JGraphX (JGraph 6) User Manual* [online]. 2014. [cit. 6. 2. 2014]. Dostupné z: <`http://jgraph.github.io/mxgraph/docs/manual_javavis.html`>.

[6] JGraph. *jgraph - jgraphx* [online]. 2014. [cit. 15. 2. 2014]. Dostupné z: <`https://github.com/jgraph/jgraphx/tree/master/lib`>.

[7] JGraph. *mxCircleLayout* [online]. 2014. [cit. 6. 4. 2014]. Dostupné z: <`http://jgraph.github.io/mxgraph/docs/js-api/files/layout/mxCircleLayout-js.html`>.

[8] KOLÁŘ, J. *Teoretická informatikas.* Praha : Česká informatická společnost, 2004. ISBN 80-900853-8-5.

[9] KOOMEN, T. a kol. *TMap Next* [online]. 2010. [cit. 6. 4. 2014]. Dostupné z: <`http://www.tmap.net/en/tmap-next`>.

[10] KUIJT, R. *Cover : SOGETI International Backpack* [online]. [cit. 6. 4. 2014]. Dostupné z: <`http://tstr.nl/cover/index.php`>.

[11] LEE, C. Y. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers.* 1961, s. 346–365.

[12] Microsoft. *Model View Presenter* [online]. 2014. [cit. 15. 2. 2014]. Dostupné z: <`http://i.msdn.microsoft.com/dynimg/IC281772.png`>.

[13] MOORE, E. F. The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching.* 1959, s. 285–292.

[14] PECINOVSKÝ, R. *Myslíme objektově v jazyku Java.* Grada, 2009. ISBN 978-80-247-2653-3.

[15] Přispěvatelé Wikipedie. *Java (programming language)* [online]. 2014. [cit. 6. 4. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Java_(programming_language)>.

[16] Přispěvatelé Wikipedie. *Java Platform, Standard Edition* [online]. 2014. [cit. 6. 4. 2014]. Dostupné z: <http://en.wikipedia.org/wiki/Java_Platform{\,}_Standard_Edition>.

[17] Přispěvatelé Wikipedie. *Swing (Java)* [online]. 2014. [cit. 6. 4. 2014]. Dostupné z: <http://cs.wikipedia.org/wiki/Swing_(Java)>.

[18] Some Character LLC. *mockingbird* [online]. 2014. [cit. 6. 4. 2014]. Dostupné z: <https://gomockingbird.com/mockingbird/>.

[19] Stackoverflow. *Stackoverflow* [online]. 2014. [cit. 6. 4. 2014]. Dostupné z: <http://stackoverflow.com>.

[20] Sun Microsystems. *Programming With Assertions* [online]. [cit. 7. 5. 2014]. Dostupné z: <http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html>.

[21] Sun Microsystems. *The Event Dispatch Thread* [online]. [cit. 6. 4. 2014]. Dostupné z: <http://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>.

[22] Sun Microsystems. *Java SE Downloads* [online]. [cit. 6. 4. 2014]. Dostupné z: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

[23] Sun Microsystems. *Java SE 7 Downloads* [online]. [cit. 6. 4. 2014]. Dostupné z: <http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html>.

[24] Sun Microsystems. *Class SwingWorker<T,V>* [online]. [cit. 6. 4. 2014]. Dostupné z: <http://docs.oracle.com/javase/6/docs/api/javax/swing/SwingWorker.html>.

Příloha A

Obsah přiloženého CD

Níže je uveden seznam složek a souborů, které jsou umístěny na přiloženém CD. Složky jsou označeny tučným písmem, soubory pak kurzívou.

- **text** - Text práce ve formátu pdf a zdrojové latex soubory.
- **app** - Spustitelná verze aplikace spolu s knihovnami ve složce lib a licenčním souborem JGraphX.
- **lib** - JGraphX a ostatní knihovny.
- **src** - Zdrojové soubory aplikace.
- **test** - Zdrojové soubory JUnit testů.
- **javadoc** - Dokumentace zdrojového kódu v html.
- *install.txt* - Instalační pokyny.
- *readme.txt* - Informace o souborech a složkách na cd.

Příloha B

Instrukce pro instalaci

B.1 Softwarové požadavky

Požadavky na spuštění jsou pouze na verzi Java. Konkrétně je potřeba mít nainstalovanou Java SE ve verzi vyšší než 7. Pro nižší verze aplikace nebude fungovat, protože kód obsahuje příkazy (konkrétně prázdné diamond operátory), které nejsou v nižších verzích podporovány. Pro spuštění stačí mít nainstalovené Java Runtime Enviroment (JRE), což je varianta ochuzená o vývojářské nástroje, které jsou obsaženy v Java Development Kit (JDK).

Pro stažení verze 7.25, ve které byla aplikace vyvíjena, následujte odkaz na [23] a pokud chcete nainstalovat nejnovější verzi, stahovat můžete zde [22].

JGraphX knihovna, potřebná pro funkčnost aplikace, je obsažena ve složce `lib`. Aplikace byla vyvíjena ve verzi 2.5.0.2. Pro stažení aktuální verze následujte odkaz na [6].

B.2 Hardwarové požadavky

Aplikace je spustitelná na jakémkoliv počítači, který má v sobě JRE a je vybaven jedním ze současných operačních systémů.

B.3 Instalace a spuštění aplikace

Aplikace je připravena ve spustitelném souboru s příponou „jar“ (java archive), tudíž jí není nutné nijak instalovat. Je pouze potřeba zkopírovat složku `lib` spolu se spustitelným souborem „jar“ do vámi zvoleného adresáře. A pro spuštění aplikace stačí zapnout konzoli, vyhledat vybraný adresář a zadat `java -jar ProcessCycleTestEditor.jar`.