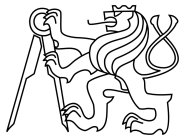
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Systém pro generování umělé aplikace pro testovací účely
Student:	Bc. Lukáš Löwinger
Vedoucí:	Ing. Miroslav Bureš, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Vytvořte systém, který bude na základě parametrů zadaných uživatelem generovat jednoduchou umělou webovou aplikaci s definovanými stavy, datovými entitami, funkcemi, procesy a CRUD operacemi nad datovými entitami, které vykonávají funkce v rámci procesů. Systém bude umožňovat v aplikaci definovat umělé chyby, které mohou být odhaleny konkrétní operací nebo kombinací operací. Výstupem generování bude jednoduché uživatelské rozhraní umělé aplikace a model vygenerované aplikace skládající se z diagramů aktivit popisujících procesy a CRUD matic popisujících operace nad datovými entitami.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 20. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Systém pro generování umělé aplikace pro testovací účely

Bc. Lukáš Löwinger

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

9. května 2016

Poděkování

Poděkování patří vedoucímu práce Ing. Miroslavovi Burešovi, Ph.D za cenné rady při vytváření aplikace. Dále patří poděkování všem osobám, které mi pomohly s uživatelskými testy. A na závěr bych rád poděkoval mým rodičům za podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2016 .....

Odkaz na tuto práci

Löwinger, Lukáš. *Systém pro generování umělé aplikace pro testovací účely.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací aplikace, která bude vytvářet umělou webovou aplikaci, jež bude sloužit k porovnávání účinnosti různých testovacích technik. Vyhodnocení účinnosti testovací techniky pak bude založeno na schopnosti odhalení co nejvíce chyb, které budou v umělé aplikaci definované uživatelem. Model umělé aplikace vznikne z vygenerovaných (se specifikovanými vlastnostmi zadanými uživatelem) nebo ručně nakreslených procesů. Procesy jsou modelovány pomocí jednotlivých stavů umělé aplikace a funkcemi nad datovými entitami. K vytvořeným procesům v rámci jednoho projektu bude možné vygenerovat CRUD matici popisující, jaké operace jsou vykonávány jakou akcí nad datovými entitami. Do umělé aplikace bude možné dogenerovat nebo ručně zadat umělé chyby, které se tester pomocí zvolené testovací techniky bude snažit odhalit. Dále bude aplikace umožňovat CSV export grafu, CRUD matice a definovaných chyb. Implementace bude provedena v jazyce Java a otestována sadou JUnit testů a funkčních testů.

Klíčová slova umělá aplikace, CRUD matice, orientovaný graf, test, testovací technika

Abstract

This master thesis deals with the design and implementation of an application which will create an artificial web application used for comparsion of various test design techniques efficiency. The efficiency evaluation of test design technique will be based on the ability to detect the biggest number of errors defined by users. The model of the artificial application will be created from generated (with specific properties given by user) or manually designed processes. Processes are designed by particular states of artificial application and by functions with data entities. Together with the created processes within a given project it will be possible to generate a CRUD matrix describing which operations are executed by which action with data entities. It will be possible to generate or manually create artificial errors connected with states of the artificial application, which the user will attempt to detect. Further functions of the application will be CSV export of graph, CRUD matrix and defined artificial errors. The implementation will be written in Java programming language and tested with set of JUnit tests and functional tests.

Keywords artificial application, CRUD matrix, directed graph, test, test design technique

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Práce se zabývá návrhem a implementací webové aplikace, která bude umožňovat vytvoření modelu procesů (ať už ručně nebo automatickým vygenerováním), nad kterými umožní vygenerování CRUD matice, která bude popisovat jaké operace vykonávají jednotlivé funkce (akce) v procesech nad jednotlivými datovými entitami. Nad takto vytvořeným a popsaným modelem procesů bude umožňovat vytvoření umělé webové aplikace. Tato umělá aplikace bude simulovat chování předem vygenerovaného procesu (nebo více procesů) a ten se bude skládat z různých akcí v systému. Systém umožní při generování umělé aplikace k těmto akcím vygenerovat a přiřadit umělé chyby.

V textu nejprve uvedu případy užití, které byly motivací pro vytvoření aplikace. V další kapitole zabývající se analýzou a návrhem budou popsány detailně jednotlivé části systému z pohledu požadavků, případů užití či vybraných uživatelských scénářů. Budou v ní dále uvedeny důležité definice vztahující se k teorii grafů, přičemž tato teorie bude využita k modelování procesů. Nadefinované budou pouze grafové struktury, které se uplatní při návrhu algoritmů. Zároveň bude obsahovat definice a popis použitých termínů týkajících se CRUD matice a chyb v umělém systému. Nejobsáhlejší kapitola se bude věnovat návrhu algoritmů pro generování umělé aplikace. Jejich návrh je založen na existujících grafových algoritmech, které mohou být lehce modifikované, aby splňovaly požadované chování. Předposlední kapitola se bude zabývat implementační části práce a bude proto obsahovat návrh architektury aplikace, zvolené technologie pro klientskou a serverovou část a popis vybraných částí. Zároveň obsahuje popis nejdůležitějších balíčků projektu, diagram nasazení a databázové schéma. Na závěr bude popsána infrastruktura, na které bylo ozkoušeno reálné nasazení aplikace. V poslední části této práce budou zmíněné druhy testů, které byly využity pro testování v průběhu vývoje a pro otestování výsledné aplikace.

Kapitola 1

Motivace

V této kapitole uvedu případy užití, které byly motivací pro vytvoření výsledné aplikace. První a nejdůležitější motivací bylo vytvoření systému, nad kterým bude možné zkoumat (testovat) účinnosti testovacích technik. Druhou motivací bylo využití výsledného systému pro školící a výukové účely. A posledním příkladem, jak je možné aplikaci použít, je vytvoření soutěže v testovaní umělé aplikace.

1.1 Návrh testovacích scénářů

První motivací, jak už jsem zmínil v úvodu, je testování efektivity testovacích technik a scénářů. Zde je největším přínosem to, že se vůbec nemusí využívat reálný systém. Stačí namodelovat umělou aplikaci, nadefinovat operace s datovými entitami a vytvořit umělé chyby. Takto vytvořená aplikace se následně může podrobit různým testům (různé testovací techniky např. z knihy TMap NEXT [4]) a na základě úspěšnosti nalezených chyb lze ladit a optimalizovat konkrétní techniku, kterou se vytváří testovací scénáře pro danou aplikaci.

Ladit testovací techniku lze např. pouze na základě počtu nalezených chyb testovacího scénáře, což dokazuje jeden z testovacích principů zmíněných v [5], který říká, že pokud je záměrem testování odhalování defektů[1], pak je testovací scénář dobrý tehdy, když má vysokou pravděpodobnost odhalení nových defektů. Z toho plyne, že pokud nehledíme na žádné další parametry a prioritizace, je smysluplné vybrat testovací scénář, který nalezne více chyb.

Výsledná umělá aplikace tímto slouží pro nalezení lepších testovacích scénářů, vyladění techniky, která je vytváří a pro testování jejich účinnosti.

[1] Definice dle ISO/IEC/IEEE 24765:2010: Problém, který pokud se nevyřeší, může způsobit pád aplikace nebo způsobí, že aplikace začne produkovat špatné výsledky. [6]

1.2 Výukové a školící účely

Další uplatnění aplikace najdou školitelé testovacích technik. Průběh školení by mohl vypadat tak, že školitel získá či vytvoří dle libosti umělý procesní model aplikace. Následně do něj zanese umělé chyby a nechá studenty, aby pomocí svých znalostí různých testovacích technik (např. PCT[2]) nalezli co nejvíce chyb. Po dokončení testu aplikace školitel určí (např. podle největšího počtu nalezených chyb - viz. předchozí sekce) nejlepší testovací scénář a ohodnotí studenty. Zároveň pozná, jak si vedly ostatní scénáře, protože bude přesně vědět, kde jsou jaké chyby a proč je daný scénář nerozpoznal. Tím si student prakticky vyzkouší návrh testovacího scénáře a podívá se na problém z jiného úhlu s přesnou zpětnou vazbou. Zároveň může školitel procesní modely upravit a tím vylepšit testovací model pro další testování.

1.3 Soutěže

Jiným využitím, zajímavějším pro komunitu testerů, by mohlo být vytvoření soutěže v testování, jejíž cílem by bylo zlepšit testerské dovednosti. Nápad vychází z existující soutěže *Testing Cup* [7] konající se v Polsku, ve které se soutěží v nalezení co největšího počtu chyb, v jejich závažnosti a ve kvalitě výsledné zprávy. Pro samotné testery je taková událost zábavnější formou, jak se dozvědět o nových testovacích technikách a obecně zvyšovat povědomí o testování softwaru.

[2]Process Cycle Test (PCT) je technika pro návrh testovacích scénářů, která popisuje jak nakombinovat posloupnosti akcí, aby bylo pokrytí aplikace co největší.

Kapitola 2

Analýza a návrh řešení

Tato kapitola se bude zabývat analýzou požadavků a návrhem aplikace (z pohledu funkčních a nefunkčních požadavků, případů užití, uživatelských scénářů a návrhem uživatelského rozhraní). Funkční požadavky vznikly za konzultace vedoucího práce v průběhu modelování výsledné aplikace, nefunkční požadavky pak definují nároky na výsledný systém. Případy užití znázorňují a detailněji popisují, co bude s aplikací možné provádět (rozdělené do rolí administrátor a tester). Uživatelské scénáře pak rozvádí konkrétní procesy a interakce uživatele se systémem. Na závěr budou uvedeny (graficky znázorněny) základní procesy v aplikaci spolu s návrhem uživatelského rozhraní, které nastíní, jak aplikace funguje, co od ní lze očekávat a umožní čtenáři si vytvořit ucelený pohled na fungování aplikace.

2.1 Funkcionální a nefunkcionální požadavky

2.1.1 Funkcionální požadavky

Vygenerování procesu

1. Systém umožní uživateli vygenerovat graf (modelující testovaný proces) dle zadaných parametrů, či aplikací definovaných globálních parametrů.

2. Nakreslený proces se zobrazí v interaktivním editoru.

Vytvoření procesu v interaktivním editoru

1. Systém umožní nakreslení a následné uložení procesu (grafu) v interaktivním editoru.

Vygenerování CRUD matice

1. Systém umožní uživateli vygenerovat CRUD matici dle zadaných parametrů.

2. CRUD matice se uloží a zobrazí pod grafem v tabulce.

Vytvoření umělých chyb

1. Uživatel bude mít možnost vybrat si ze dvou druhů chyb (chyby vzniklé kombinací chyb a datové chyby), které propojí s již vytvořeným modelem procesů.
2. Systém bude umožňovat chyby vzniklé kombinací akcí vygenerovat dle zadaných parametrů.
3. Systém zobrazí všechny (jak už definované ručně, či generované automaticky) chyby pod CRUD maticí.

Správa umělé aplikace

1. Systém umožní vytvořit umělou webovou aplikaci na základě definovaných procesů, CRUD matice a zanesených chyb.
2. Uživatel se bude moci v umělé aplikaci procházet.

Export CSV

1. Systém bude umožňovat export grafu, CRUD matice a přidaných chyb v CSV.

2.1.2 Nefunkcionální požadavky

Platforma pro vývoj

1. Aplikace bude vyvíjena v jazyce Java.

Server

1. Aplikace bude nasazená a odladěná na serveru Apache Tomcat.

Nezávislost na operačním systému

1. Aplikace bude nezávislá na operačním systému (čehož bude dosaženo využitím běhového prostředí Java, které je multiplatformní a zvolený server Apache Tomcat běží bez problémů jak na Linux, OSX tak Windows).

Rozšiřitelnost a modifikovatelnost

1. Systém bude snadno rozšiřitelný (využitím vhodných návrhových vzorů, implementace proti rozhraní[3]) a modifikovatelný (dobře zdokumentovaný).

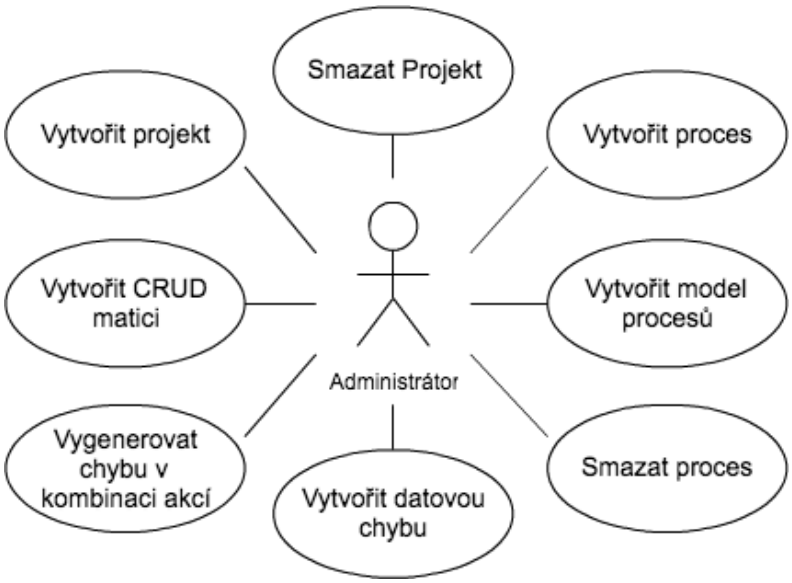
[3]Klíčové slovo Javy (interface), které definuje šablonu konkrétní třídy.

2.2 Případy užití

Případy užití využívají akce v systému a propojují je s jeho aktérem (kterým může být administrátor, uživatel, systém, ...). Případy užití jsou rozděleny do dvou skupin, první z nich je spojena s přípravou procesů, definování chyb a vytvoření CRUD matice. Tyto úkony musejí předcházet před tvorbou samotné umělé aplikace. V druhé skupině jsou popsány případy užití s umělou aplikací.

2.2.1 Práce s projekty a vytváření modelu umělé aplikace

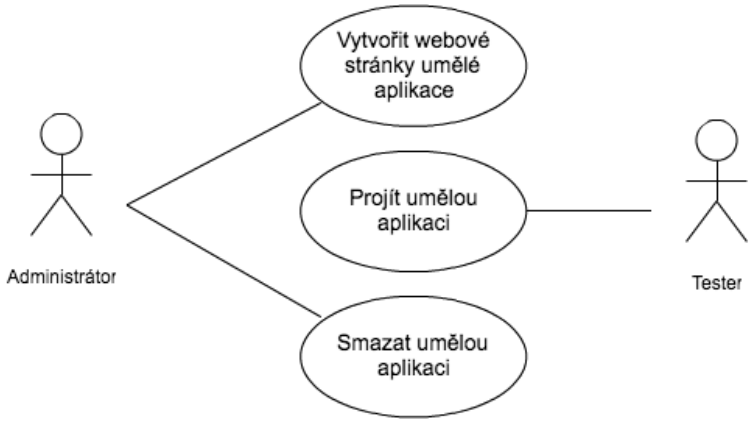
Následující obrázek 2.1 zobrazuje hlavní případy užití administrátora aplikace. Obsahuje akce spojené s projekty a procesy (tvorba a mazání), dále pak vygenerování chyb a další akce (ty budou popsány detailněji v uživatelských scénářích v podsekcích 2.3.1 a 2.3.2) sloužící k tvorbě CRUD matice a modelu procesů.



Obrázek 2.1: Případy užití vztahující se k přípravě umělé aplikace

2.2.2 Vytváření a procházení umělé aplikace

Obrázek 2.2 zobrazuje dva aktéry (Administrátor a Tester), kteří mají na starost různé akce. Administrátor nejprve vytvoří umělou aplikaci (detailněji popsáno v uživatelském scénáři v podsekci 2.3.3), kterou předá testerovi, aby ji mohl projít. Na závěr ji může také smazat (po provedení testu).



Obrázek 2.2: Případy užití vztahující se k manipulaci s umělou aplikací

2.3 Klíčové uživatelské scénáře

Uživatelské scénáře popisují interakci mezi systémem a uživatelem a rozšiřují tak případy užití. Jsou popsány pouze klíčové a složitější uživatelské scénáře a každý z nich je doplněn o nutné prerekvizity.

2.3.1 Vytvoření modelu procesů

Prerekvizity:

- V aplikaci je vytvořený alespoň jeden projekt.

Scénář:

1. Systém zobrazí formulář pro vytvoření procesu.
2. Uživatel vyplní formulářová pole a vytvoří prázdný proces.
3. Systém zobrazí seznam již vytvořených procesů (včetně naposledy vytvořeného).
4. Uživatel vybere nový prázdný proces.
5. Systém zobrazí stránku, kde budou informace o procesu, formulář pro vygenerování grafu, formulář pro vygenerování CRUD matice a formulář pro tvorbu umělých chyb.
6. Uživatel vyplní formulář pro vygenerování grafu.
7. Systém zobrazí v interaktivním editoru graf.

2.3.2 Vytvoření CRUD matice v modelu umělé aplikace

Prerekvizity:

- V aplikaci je vytvořen alespoň jeden projekt a v něm alespoň jeden proces s validním grafem.

Scénář:

1. Systém zobrazí stránku, kde budou informace o procesu, formulář pro vygenerování grafu, formulář pro vygenerování CRUD matice a formulář pro tvorbu umělých chyb.
2. Uživatel vyplní formulář pro vygenerování CRUD matice.
3. Systém zobrazí vytvořenou CRUD matici.

2.3.3 Vytvoření webových stránek umělé aplikace

Prerekvizity:

- V aplikaci je vytvořen alespoň jeden projekt a v něm alespoň jeden proces s validním grafem.
- V aplikaci je vytvořena CRUD matice.
- V systému není ještě žádná umělá aplikace vytvořena.

Scénář:

1. Systém zobrazí detaily o projektu včetně seznamu procesů a tlačítka pro tvorbu umělé aplikace.
2. Uživatel vytvoří umělou aplikaci pomocí tlačítka.
3. Systém zobrazí umělou aplikaci s definovanými procesy.
4. Uživatel vybere proces, který chce začít procházet.
5. Systém zobrazí počáteční stav procesu s možností přechodu do dalších stavů.

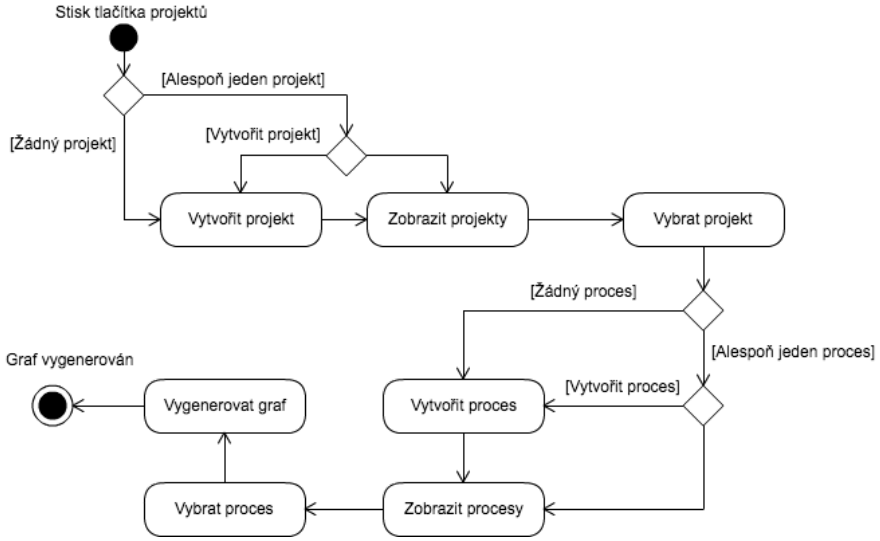
2.4 Základní procesy v aplikaci

V této sekci budou uvedeny (UML[4] diagram aktivit) a popsány dva hlavní procesy v aplikaci. Vybrán byl nejprve proces vytváření grafu, který zároveň slouží jako prerekvizita k dalšímu procesu, který vytváří CRUD matici.

[4]Unified Modeling Language (UML) je jazyk pro vizualizaci modelů a procesů.

2.4.1 Vytvoření procesu v modelu umělé aplikace

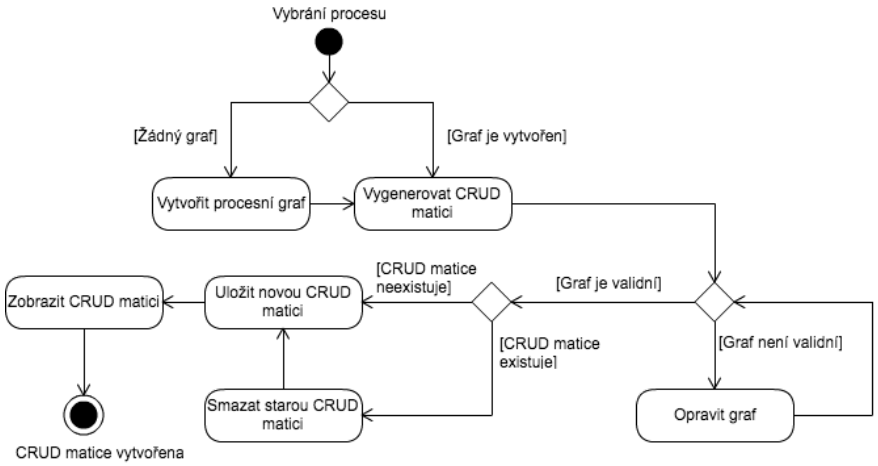
Proces (obrázek 2.3) začíná stiskem tlačítka zobrazující projekty, které buď zobrazí projekty, pokud nějaké existují, nebo požádá uživatele, aby nejprve nějaký vytvořil. Projekt, ať už nově či delší dobu vytvořený, uživatel vybere a zobrazí se mu seznam procesů pro tento projekt. Pokud žádný proces neobsahuje, vyzve uživatele k jeho vytvoření a následně zobrazí všechny procesy včetně aktuálně přidaného. Uživatel nějaký vybere a zvolí v něm generování grafu.



Obrázek 2.3: Proces vytváření procesního grafu

2.4.2 Vytvoření CRUD matice v modelu umělé aplikace

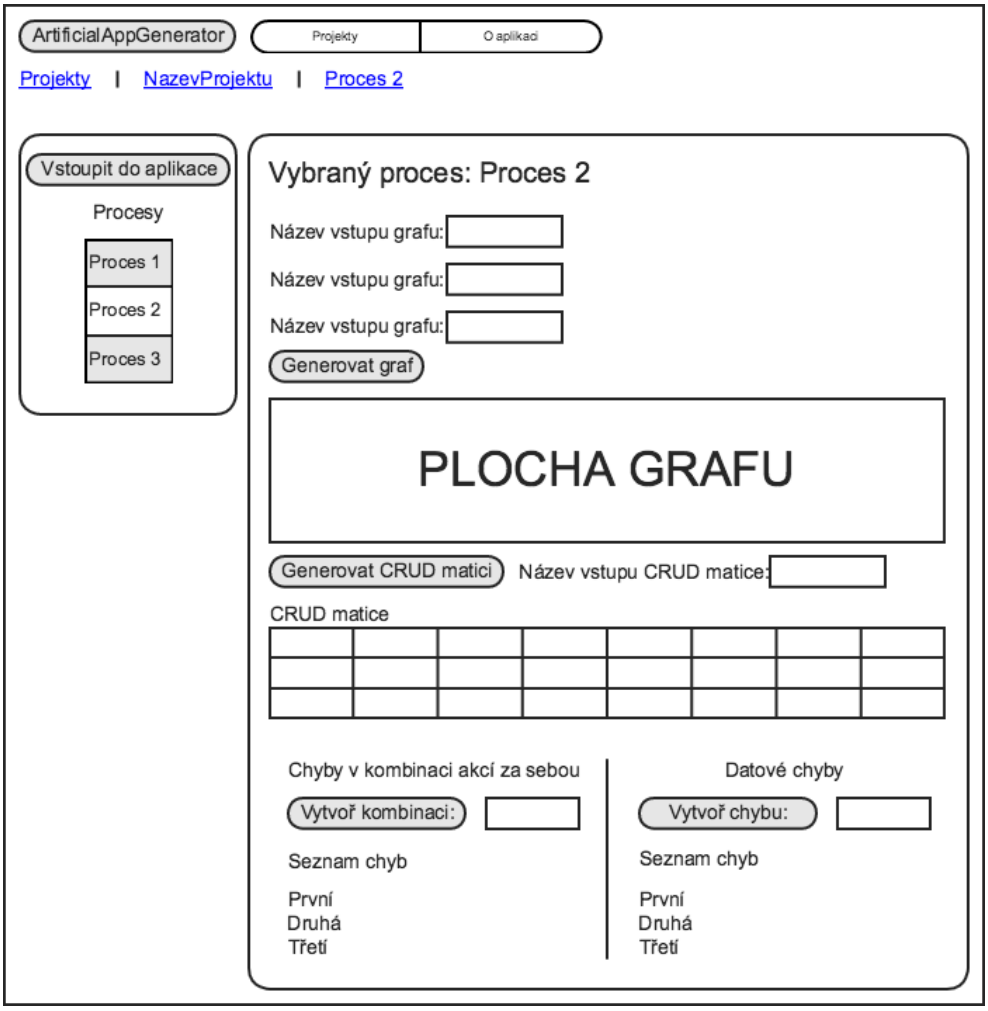
Proces (obrázek 2.4) začíná vybráním procesu, ze kterého se bude vytvářet CRUD matice. Nejprve se zkontroluje, zda je vytvořen nějaký graf a pokud není, je potřeba ho nejprve vytvořit (bez grafu CRUD matici nelze vytvořit). Následuje kontrola validity grafu, která když neprojde, je uživatel upozorněn a musí graf převést do validního stavu. Nakonec se ještě zkontroluje, zda již CRUD matice nebyla vytvořena a pokud byla, tak se smaže, uloží se nová a zobrazí se uživateli.



Obrázek 2.4: Proces vytváření procesního grafu

2.5 Návrh uživatelského rozhraní

Návrh vychází z funkčních požadavků a z případů užití, které nastínily, co by aplikace měla obsahovat. Obrazovka (na obrázku 2.5) tvorby procesu obsahuje téměř veškeré možné konfigurace směrem od uživatele. Prvky jsou soustředěny tak, aby byla tvorba grafu (procesu) co nejjednodušší a aby bylo na první pohled zřejmé, kam má uživatel kliknout. V celé aplikaci se dodržuje hierarchická struktura ve formě: Projekty - Vybraný projekt - Vybraný proces, kterou znázorňují odkazy (oddělené vertikálními čárami) v horní části obrazovky. Směrem dolů můžeme vidět dvě různé plochy, jednu menší (vlevo) a druhou větší (vpravo). Menší plocha bude sloužit k přehledu již vytvořených procesů v daném projektu, kde aktuálně vybraný proces bude zvýrazněn (viz. Proces 2) a zároveň bude obsahovat vstup z aktuálního procesu do umělé aplikace. Větší plocha bude obsahovat informace o procesu a vstupní formulářová pole, kde si uživatel nadefinuje vlastnosti výsledného grafu, který se po stisknutí tlačítka zobrazí na ploše pod nimi. Ve spodní části bude zobrazena CRUD matice a konfigurace chyb v aplikaci.

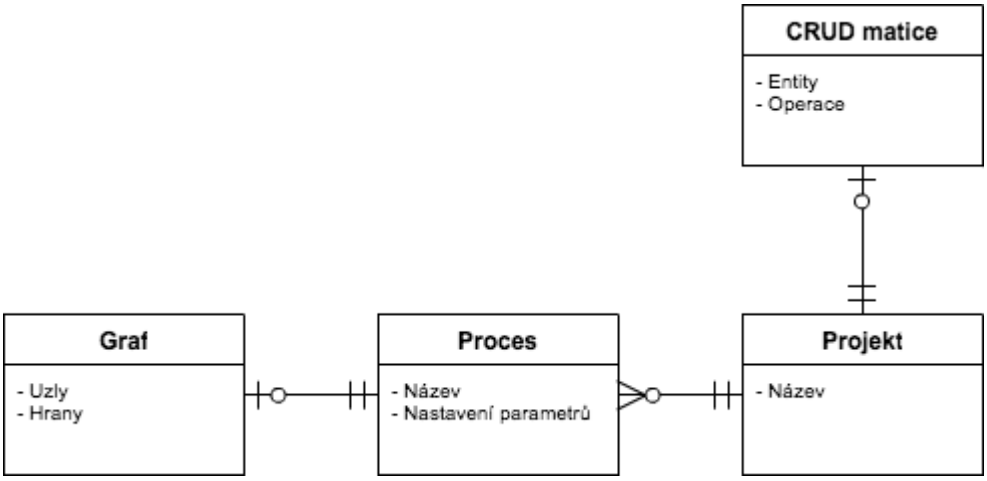


Obrázek 2.5: Hlavní obrazovka generování procesů a CRUD matic

2.6 Základní datové objekty systému pro generování umělé aplikace

Obrázek 2.6 zobrazuje základní objekty aplikace popisující základní nejdůležitější datové entity a jejich vzájemný vztah. Model využívá IE[5] notaci definovanou v [8] pro modelování kardinalit. Pokud bych na model nahlížel z nejvyššího patra hierarchie (tzn. od prvního objektu, z kterého vycházejí všechny ostatní), začal bych s `projektem`, který může obsahovat `procesy` a `CRUD matici`, která je spojená právě s jedním projektem a `proces` taktéž. `Proces` pak může obsahovat `graf`, který náleží právě jednomu `procesu`.

[5]Information Engineering



Obrázek 2.6: Základní objekty aplikace

2.7 Funkcionality implementovány navíc oproti zadání

Některé z uvedených požadavků (v sekci 2.1.1) nebyly původně v zadání práce a byly přidány navíc (pro lepší použitelnost). Prvním z nich je vytvoření datových chyb (definice 14), které se chovají jinak než chyby v sekvenci akcí. Další funkcionalitou je tvorba procesu (grafu) manuálně v interaktivním editoru, což umožňuje namodelovat proces dle libosti. Tento přístup lze kombinovat s generováním procesu a upravit tak pouze některé detaily. Pro možnost propojení s existující aplikací PCTgen pro automatické vytváření testovacích scénářů z modelu testovaného systému byl přidán export do formátu CSV, mající stejnou strukturu jako v aplikaci PCTGen.

2.8 Použitá terminologie: Procesy, procesní grafy a grafy

V textu bude mnohokrát použito slovo *proces*, který normálně znamená nějaký sled akcí v systému. Jelikož aplikace přetěžuje toto slovo ve struktuře `Projekt-Proces` (anglicky Workspace-Workflow), bude nutné slovo *proces* rozlišovat podle kontextu. Pokud bude využito k popisu a struktury `Projekt-Proces`, nebude ho pak možné spojit s žádnými akcemi. Pro vytvoření *procesu* v rámci `Procesu` zavedu spojení procesní graf (či jednoduše graf), který reprezentuje konkrétní sled akcí tak, jak ho definuje normální význam slova *proces*.

2.9 Použité struktury v aplikaci

V této sekci bude formálně popsán orientovaný graf, který bude reprezentovat proces. Následně pak uvedu formální definici CRUD matice a v poslední podsekci popíšu možné chyby, jež lze připojit k procesu.

2.9.1 Orientovaný graf

Popsané algoritmy v kapitole 3 využívají strukturu grafu a proto je nejprve potřeba nadefinovat graf jako takový. Jelikož jsou procesy graficky popsány uzly a orientovanými hranami, využijeme definice orientovaného grafu. Zároveň předpokládáme, že graf může obsahovat cykly a nejsou povoleny souběžné (někdy nazývané rovnoběžné) hrany. Začneme definicí orientovaného grafu.

Definice 1 (Orientovaný graf [9])**.** Nechť V a E jsou libovolné disjunktní množiny a $\sigma : E \mapsto V \times V$ zobrazení. Orientovaným grafem nazveme uspořádanou trojici $G = \langle E, V, \sigma \rangle$, prvky množiny E nazýváme orientovanými hranami grafu G, prvky množiny V uzly grafu G a zobrazení σ incidencí grafu G.

Další specifikací je souvislost grafu, která vychází opět z procesů u kterých předpokládáme, že jsou kompaktní a „v celku“. K nadefinování souvislosti ještě potřebujeme definici sledu.

Definice 2 (Sled grafu [9])**.** Nechť pro danou dvojici uzlů u a v grafu $G = \langle E, V, \varrho \rangle$ existuje posloupnost uzlů a hran

$$S = \langle u_0, h_1, u_1, h_2, \ldots, u_{n-1}, h_n, u_n \rangle$$

kde $h_i \in E,\ \varrho(h_i) = [u_{i-1}, u_i]$ pro $i = 1, 2, \ldots, n, u_i \in V$ pro $i = 0, 1, \ldots, n, u_0 = u, u_n = v$. Pak tuto posloupnost nazýváme sledem grafu G mezi uzly u a v.

S touto znalostí můžeme definovat souvislý neorientovaný graf. Kterou využijeme pro definici souvislosti orientovaného grafu.

Definice 3 (Souvislý neorientovaný graf [9])**.** Souvislým grafem nazýváme takový neorientovaný graf, mezi jehož libovolnými dvěma uzly existuje sled.

Pro orientovaný graf se souvislost dělí na dvě části:

- slabá souvislost
- silná souvislost

Silná souvislost, jak už z názvu vyplývá, má větší nároky na vlastnosti a strukturu grafu, které nejsou u tvorby procesů potřeba. Proto si vystačíme se slabou souvislostí (typicky se slovo „slabá“ vypouští, protože je analogií k souvislosti neorientovaného grafu).

Definice 4 (Symetrizace orientovaného grafu [10])**.** Symetrizací orientovaného grafu se nazývá neorientovaný graf, který dostaneme tak, že „zapomeneme“ na orientaci hran. Symetrizací orientovaného grafu $\langle V, E, \sigma \rangle$ je tedy neorientovaný graf $\langle V, E, \sigma' \rangle$, kde $\sigma'(e) = [x, y]$, jestliže $\sigma(e) = (x, y)$ nebo $\sigma(e) = (y, x)$.

Definice 5 (Slabě souvislý orientovaný graf [11])**.** Orientovaný graf je slabě souvislý, pokud jeho symetrizace je souvislý graf.

Nyní už máme všechny potřebné definice k tomu, abychom popsali grafovou strukturu, nad kterou pracují algoritmy. Přesně jde tedy o **cyklický prostý (slabě) souvislý graf**. Ještě si ale zjednodušíme zápis samotného grafu. Definice orientovaného grafu lze zjednodušit právě díky použitému sousloví *prostý graf*. V prostém grafu totiž neexistují souběžné hrany a proto není potřeba definovat zobrazení σ, které by takové hrany povolovalo (bez zobrazení nelze určit a tedy ani definovat rovnoběžné hrany, tudíž je u neprostého grafu potřebné). A díky tomu si definici grafu zjednodušíme na uspořádanou dvojici $\langle V, E \rangle$, kde množina uzlů V se shoduje s množinou z definice 1 a množina hran je podmnožinou kartézského součinu $E \subseteq V \times V$. Navíc u grafu předpokládáme, že má vždy jeden kořen (uzel bez vstupních hran a s právě jednou vystupující hranou), který reprezentuje vstupní bod (začátek) procesu a alespoň jeden konec (uzel bez výstupních hran a s alespoň jednou vstupní).

V textu dále budeme rozlišovat čtyři typy hran, které navštívíme při průchodu grafu do hloubky (DFS[6]). Těmi jsou (definice z [9]):

- Stromové hrany - hrana (u, v) je stromovou hranou, jestliže k objevení uzlu v došlo při procházení seznamu sousedů uzlu u.
- Zpětné hrany - nestromové hrany (u, v), které vedou od uzlu u k nějakému jeho předkovi.
- Dopředné hrany - nestromové hrany (u, v), které vedou od uzlu u k nějakému jeho potomkovi.
- Příčné hrany - všechny ostatní hrany.

Dalším pojmem je faktor grafu, který budeme používat při převodu cyklického grafu na acyklický. Pro jeho definici musíme nejprve uvést, co je to podgrafu grafu.

Definice 6 (Podgraf grafu [9])**.** Graf $G' = \langle H', U', \sigma' \rangle$ nazýváme podgrafem grafu $G = \langle H, U, \sigma \rangle$, jestliže platí

$$(H' \subseteq H) \ \& \ (U' \subseteq U) \ \& \ h \in H'(\sigma'(h) = \sigma(h)).$$

[6]Depth First Search (DFS) je metoda traverzování grafu do hloubky.

Definice 7 (Faktor grafu [9])**.** Podgraf $G' = \langle H', U, \sigma' \rangle$, jehož množina uzlů je shodná s množinou uzlů grafu G, nazýváme faktorem grafu G.

Posledním pojmem je výška uzlu, kterou budeme potřebovat při generování CRUD matice (popsána v další sekci).

Definice 8 (Výška uzlu [12])**.** Výška uzlu u je počet hran na nejdelší jednoduché cestě (cesta bez opakujících se uzlů) směřující z uzlu u dolů k listu.

2.9.2 CRUD matice

CRUD matice udává, jaké databázové operace (CREATE, READ, UPDATE a DELETE) provádějí funkce (akce) nad datovými entitami. Je to výsledný produkt, který může a nemusí vzniknout během analýzy. Propojuje procesní (proces lze chápat jako sled akcí) a datové modely, kde procesní model vznikne dekompozicí business procesů, jehož úkolem je popsat, co jednotlivý proces dělá, ale už neříká proč, jak nebo kdy je vykonán [13]. Datový model graficky znázorňuje propojení a vztah objektů (entit) v aplikaci a je určen na konceptuální úrovni návrhu (obsahuje jen ty nejdůležitější entity a vztahy). Pro formální definici je potřeba nejprve nadefinovat dvě již zmíněné struktury, kde tou první je datová entita.

Definice 9 (Datová entita [14])**.** Datová entita je objekt, který je složen z dat uložených v databázi testovaného systému.

Pro testovací účely se většinou datové entity určují na konceptuální úrovni návrhu (vybírají se ty nejdůležitější bez dodatečných atributů) a typicky jde o snahu přiblížit či spojit entity s těmi reálnými. Konkrétní fyzické uložení dat je už zbytečný detail, na který lze ale stále aplikovat definici CRUD matice.

Druhou strukturou je proces (v této práci je užit termín `akce`), který je obsažen v procesním modelu.

Definice 10 (Akce [14])**.** Akce je proces v testovaném systému, která provádí nějaké z *Create*, *Update*, *Read* a *Delete* (zkráceně C, R, U, D) operací nad datovou entitou.

Určení akcí by mělo sledovat tyto body:

1. Akce by měly být identifikovány na konceptuální úrovni.
2. Je možné danou akci vyzkoušet v testovaném systému.
3. Akce nemusí být atomické, protože by v takové situaci vedl návrh k vysoké úrovni granularity, která pro samotný test není žádoucí.
4. Akce by měly být transakční.

Akcí budeme v grafu označovat uzel, který má alespoň jednu vstupní hranu a přesně jednu výstupní, zatímco rozhodovací uzel má alespoň jednu vstupní hranu a alespoň dvě výstupní. Dále budeme používat pojem sdílené akce.

Definice 11 (Sdílená akce)**.** Nechť $a \in A$ je akce, která se vyskytuje ve dvou či více procesech. Pak je a sdílenou akcí.

Sdílené akce vycházejí z praxe, kde skutečně ve dvou procesech může být vykonávána tatáž akce. Systém by měl umožňovat generování sdílených akcí z jiných procesů. Následuje definice CRUD matice.

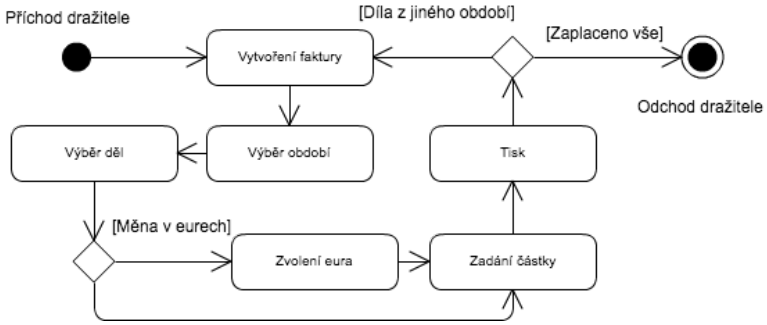
Definice 12 (CRUD matice [14])**.** Nechť $A = \{a_1, \ldots, a_n\}$ je množina všech akcí a $E = \{e_1, \ldots, e_p\}$ je množina všech datových entit testovaného systému. Potom CRUD matici definujeme jako $M = (m_{i,j})_{n,p}$, kde $m_{i,j} = \{o \mid o \in \{C, R, U, D\} \Leftrightarrow$ akce $a_i \in A$ provádí nějaké z CREATE, READ, UPDATE nebo DELETE operací nad datovou entitou $e_j \in E\}$.

Z definice jasně plyne, že akce může provádět mnoho operací nad jednou entitou. My si v této práci zjednodušíme znění definice a budeme požadovat, aby akce měla maximálně jednu operaci nad entitou. U operací navíc budeme požadovat, aby byly vykonávány ve správném pořadí. Konzistentní sekvence akcí s operacemi nad jednou entitou vypadá následovně. Musí začínat právě jednou operací CREATE, kterou následuje alespoň jeden READ nebo UPDATE a na konci právě jeden DELETE. Toto omezení bylo zadáno vedoucím práce.

Pro ukázku CRUD matice jsem zvolil proces vytvoření faktury v galerii s obrazy. Proces na obrázku 2.7 popisuje průběh tvorby faktury. Fakturu je potřeba nejprve vytvořit, následně vybrat období, ve kterém byla díla vydražena. Z děl se vyberou ty, které si dražitel přeje zaplatit, zadá se celková částka (možnost volby eur) a faktura se vytiskne. CRUD matice tohoto procesu by mohla vypadat tak, jak je naznačeno v tabulce 2.9.2, která má tři datové entity (**Faktura**, **Období** a **Dílo**), nad kterými akce provádějí CRUD operace. Navíc si můžeme všimnout, že tato CRUD matice nemá konzistentní sekvenci akcí, protože poslední dva sloupce obsahují pouze READ a v prvním sloupci chybí DELETE.

	Faktura	**Období**	**Dílo**
Vytvoření faktury	C		
Výběr období	U	R	
Výběr děl	U		R
Zadání částky	U		
Tisk	R		

Tabulka 2.1: CRUD matice pro ukázkový proces



Obrázek 2.7: Ukázka procesu

2.9.3 Typy chyb v umělé aplikaci

Uživatel bude mít možnost vytvořit dva druhy chyb. První chyba se váže pouze na akce v procesu, které když se vykonají za sebou, způsobí pád aplikace. Druhý typ popisuje, jak může rozhozená datová entita způsobit pád aplikace. Nejprve uvedu definici (definice dodány vedoucím práce) pro chybu v kombinaci akcí za sebou.

Definice 13 (Chyba v kombinaci akcí za sebou)**.** Uspořádaná n-tice akcí $(a_1, a_2, \ldots, a_n)$, které pokud se zavolají za sebou, způsobí chybu aplikace.

Pro akce v sekvenci platí, že na sebe nemusí přímo navazovat a sekvence musí jít vykonat v nějakém procesu. Speciální případ je *chyba v akci*, což je samotná akce, která způsobí chybu a aplikace na ní spadne. Následuje definice datové chyby.

Definice 14 (Datová chyba)**.** Nechť A je množina všech akcí v CRUD matici a $a \in A$ je akce provádějící operaci CREATE nebo UPDATE nad datovou entitou e, jejíž vykonáním se entita e uvede do nekonzistentního[7] stavu. Pak je $B = \{b | b \in A \ \wedge \ b$ vykonává libovolnou operaci nad entitou $e\}$. Datová chyba je pak definována jako trojice (a, e, B).

Vytvoření datové chyby v aplikace začíná výběrem akce a, která uvede datovou entitu e do nekonzistentního stavu. Následuje výběr akcí z množiny B definující akce, které manipulují s rozhozenou datovou entitou a způsobí tak pád aplikace.

[7](Nekonzistentní datová entita) Podle specifikace mají být v entitě jiná data než ta, která jsou reálně fyzicky uložena příčinou vzniklé chyby způsobené libovolnou operací. Aplikace se pak začne chovat špatně, protože jsou data v rozporu se specifikací.

Kapitola 3

Návrh algoritmů pro generování umělé aplikace

Tato kapitola se zabývá použitými algoritmy v aplikaci. Všechny algoritmy využívají na pozadí strukturu grafu, který popisuje procesy v aplikaci. Vždy uvedu pseudokód konkrétního algoritmu a místo popisování jeho struktury po jednotlivých řádcích se budu věnovat spíše vybraným blokům z větší perspektivy. Ke každé funkci či proceduře vždy přidám i tabulku vstupních parametrů s jejich popisem a na konci uvedu jejich asymptotickou časovou složitost. U průchodu do hloubky (DFS) budeme předpokládat, že jsou sousední uzly uloženy v poli *Adj* (indexováno uzly), které obsahuje $|V|$ odkazů na seznamy sousedů. Časová složitost je pak $\mathcal{O}(|V| + |E|)$, protože navštívíme každý uzel a z každého uzlu se dostaneme pomocí hran, kterých je celkem $|E|$.

3.1 Generování procesů

3.1.1 Úvod

První algoritmus slouží k vytvoření umělého procesu (grafu), který má předem definované parametry (ty jsou vstupem do funkce), jež určuje uživatel dle svých preferencí. Dále ještě obsahuje parametry, které uživatel nedefinuje, ale slouží k volání funkce s již vygenerovanými grafy a tak může potencionálně využít ostatní grafy z projektu k tvorbě nového procesního grafu.

3.1.2 Princip

Na začátku algoritmu 1 se nejprve promažou propojení společných akcí, které byly svázány s již neexistujícím grafem (tato situace může nastat při opětovném volání funkce). Následuje samotná tvorba grafu, která začíná od listů (řádky 7 až 9) grafu a postupuje dál (řádky 10 až 29) až ke kořenu. Při postupu od listů vzhůru se vždy vytvoří uzel a ten se spojí s náhodným počtem již vy-

tvořených uzlů (budoucích potomků). Pravděpodobnost $pOfAction$ rozhodne, zda se tento počet stáhne na jedničku a vznikne tak akce nebo se vytvoří rozhodovací uzel s náhodným počtem (omezen shora parametrem $numDecEdges$) odchozích hran. Výběr potomků je náhodný až na první volbu, kdy je potřeba vybrat naposledy přidaný uzel tak, aby nevznikl uzel bez vstupních hran. Nakonec generování se odstraní uzly (listy), které nemají vstupní hrany, neboť se k nim nepovedlo připojit žádný uzel. Následuje vytvoření cyklů (funkce bude popsána níže), pokud je uživatel požaduje. Nakonec se propojí určitý počet akcí (funkce 6 bude popsána níže), čímž vzniknou sdílené akce spravované v poli C.

Celková složitost této funkce je složením všech bloků a jejich složitostí. V prvním bloku se prochází všechny sdílené akce a pro každou z nich se musí projít všechny uzly ze všech grafů AG. Nechť je tedy množina AGV rovna sjednocení uzlů všech grafů z AG, potom je výsledná složitost tohoto bloku $\mathcal{O}(|C| * |AGV|)$. Následuje dlouhý blok pro vytvoření grafu, pro jehož výpočet složitosti si zavedeme parametr $innerNodes = numOfNodes - numLeafs$ určující počet vnitřních uzlů (pro zjednodušení prohlásíme kořen také za vnitřní uzel). Pak je nejhorší složitost rovna součtu

$$numLeafs + (numLeafs+1) + (numLeafs+2) + \ldots + (numLeafs+innerNodes-1),$$

protože v i-tém kroku iterace přidáme jeden nový uzel a ten napojíme na všechny ostatní již vytvořené uzly, jejichž počet je roven $numLeafs + i - 1$. Součet lze zapsat a vyjádřit v sumě

$$\sum_{i=0}^{innerNodes-1} (numLeafs+i) = \frac{1}{2}\, innerNodes\, (2\, numLeafs + innerNodes - 1).$$

Po dosazení a asymptotickém zjednodušení vychází složitost $\mathcal{O}(innerNodes^2)$. Pokud bude vstupní parametr $numDecEdges$ alespoň $numNodes-1$, budeme tento výsledek zároveň považovat za maximální počet hran s předpokladem, že náhodnost vyhoví každému přidání hrany. Maximální počet uzlů je dán parametrem $numNodes$. Dalším blokem je generování cyklu, který má složitost $\mathcal{O}(numLeafs\ numNodes + innerNodes^2)$ a po něm následuje blok s přiřazením sdílených akcí se složitostí $\mathcal{O}(numActionNodes\ \log(numActionNodes) + numNodes + innerNodes^2)$. Počet akčních uzlů $numActionNodes$ můžeme shora omezit počtem $innerNodes$ a celková složitost je pak tedy $\mathcal{O}(|C| * |AGV| + innerNodes^2)$.

Název parametru	Popis
numNodes	maximální počet uzlů
numLeafs	maximální počet listů (koncových uzlů)
numDecEdges	maximální počet odchozích hran z rozhodovacího uzlu
pOfAction	pravděpodobnost akčního uzlu
numSameActions	počet sdílených akcí (v procentech)
wantCycle	zda chce nebo nechce cyklus v grafu
AG	množina grafů, z které se můžou vybírat společné akce
C	obsahuje seznam již nadefinovaných společných akcí

Tabulka 3.1: Parametry funkce `CreateGraph`

Algorithm 1 Funkce vytvářející graf dle zadaných parametrů

function CreateGraph($AG, C, numLeafs, numNodes, numDecEdges, pOfAction, numSameActions, wantCycle$)
  for $c \in C$ **do**
    if $c.from$ nebo $c.to$ není uzlem v žádném z grafů AG **then**
      smaž společnou akci c
  $V \leftarrow \emptyset$; $E \leftarrow \{(\emptyset, \emptyset)\}$; $i \leftarrow 0$
  $rLeafsCount \leftarrow$ náhodný integer z rozmezí $\langle 1, numLeafs \rangle$
  while $i < rLeafsCount$ **do**
    $node \leftarrow$ vytvoř nový uzel
    $V \leftarrow V \cup \{node\}$; $i \leftarrow i + 1$
  for $k \leftarrow i$ **to** $numNodes - 1$ **do**
    $node \leftarrow$ vytvoř nový uzel
    $random \leftarrow$ náhodné reálné číslo v rozmezí $\langle 0, 1 \rangle$
    if $pOfAction > random$ **then**
      $rEdgesCount \leftarrow 1$
    else
      $min \leftarrow min\{2, \text{SIZE}(V)\}$ ▷ ošetření jednoho koncového uzlu
      $max \leftarrow min\{numDecEdges, \text{SIZE}(V)\}$
      $rEdgesCount \leftarrow$ náhodné celé číslo z rozmezí $\langle min, max \rangle$
    INIT_STACK(T) ▷ inicializuje dočasný zásobník
    for $j \leftarrow 0$ **to** $rEdgesCount - 1$ **do**
      if $k + 1 = nodesCount \wedge j > 0$ **then** ▷ ošetření kořenu
        break
      if $j = 0$ **then** ▷ první napojení
        $end \leftarrow$ odeber z V naposledy přidaný uzel
      else
        $end \leftarrow$ náhodně odstraněný uzel z V
      $E \leftarrow E \cup \{(node, end)\}$

28: vrať zpět do množiny V právě odebrané uzly
29: $V \leftarrow V \cup \{node\}$
30: odstraň všechny uzly z V, které neincidují s žádnou hranou
31: **if** *wantCycle* **then**
32: $numInners \leftarrow numNodes - numLeafs - 1$ ▷ počet vnitřních uzlů
33: GenerateCycle$(V, E, numInners)$
34: $actionNodesToSelect \leftarrow$ všechny akční uzly ze všech grafů G
35: $g \leftarrow (V, E)$
36: $numActionNodes \leftarrow$ počet akčních uzlů grafu g
37: AssignClones$(C, g, actionNodesToSelect, numActionNodes, numSameActions)$
38: **return** g

Algoritmus pro vytvoření cyklu (pseudokód 2) funguje tak, že se nejprve označí (pomocí procedury 3) všechny cesty z listů ke kořenu, čímž zafixujeme hrany, s kterými nesmíme hýbat, aby vždy existovala alespoň jedna cesta do každého z listů. Poté se pokusí vytvořit umělou hranu, která bude spojovat náhodně vybrané uzly tak, aby hrana směřovala směrem od kořenu k listu. Pokud se takovou hranu podaří vytvořit (pomocí funkce 4), přidáme ji k hranám grafu, což nám zajistí, že cyklus určitě vytvoříme. Když se to nepodaří, cyklus nepůjde vytvořit. Nakonec projdeme všechny hrany a pokud jsme nějakou z nich nenavštívili, můžeme otočit její směr, čímž ve výsledku vznikne cyklus a to i kdyby nezbyla žádná nenavštívená hrana (protože máme uměle vytvořenou hranu) a také můžeme odstranit uměle přidanou hranu, pokud se nám podaří otočit směr u nějaké jiné. Složitost je složena z několika bloků, v prvním se musí získat listy grafu (což znamená projít všechny uzly se složitostí DFS), v druhém bloku se pro každý list volá funkce `MarkPathToRoot` (což má složitost $\mathcal{O}(leafs * |V|)$). V předposledním bloku se volá funkce `CreateRandomEdge`, která se složitostí rovná DFS a v posledním bloku se projdou všechny hrany. Celková složitost po zanedbání konstant je tedy $\mathcal{O}(leafs * |V| + |E|)$

Název parametru	Popis
V	množina uzlů
E	množina hran
numInners	počet vnitřních uzlů

Tabulka 3.2: Parametry procedury `GenerateCycle`

Algorithm 2 Procedura, která vytvoří cyklus

1: **procedure** GenerateCycle$(V, E, numInners)$
2: $leafs \leftarrow$ získej všechny koncové uzly z $(V, E, numInners)$
3: $eV \leftarrow$ inicializuj pole s hodnotou $false$ pro každou hranu jako index
4: **for** $leaf \in leafs$ **do**
5: MarkPathToRoot$(leaf, eV, V, E)$

```
6:    randomEdge ← CreateRandomEdge(V, E, numInners)
7:    E ← E ∪ {randomEdge}
8:    for e ∈ E do
9:        if eV[e] = false ∧ hrana e neinciduje s kořenem ani s žádným z koncových uzlů then
10:           obrať směr hrany e
11:           if e ≠ randomEdge then                ▷ pokud hrana nebyla využita
12:               E ← E \ {randomEdge}              ▷ odstraň ji
```

Procedura 3 prochází rekurzivně od listu ke kořenu libovolnou cestou. Při tomto průchodu nastavuje hrany jako navštívené. Složitost je podobná složitosti klasického průchodu do hloubky, nicméně zde vybíráme vždy jen jednu hranu a protože musíme navštívit nejhůře všechny uzly, je složitost $\mathcal{O}(|V|)$.

Název parametru	Popis
n	počáteční uzel cesty ke kořenu
eV	jednorozměrné pole k určení, zda byla hrana navštívená
V	množina uzlů
E	množina hran

Tabulka 3.3: Parametry procedury `MarkPathToRoot`

Algorithm 3 Procedura, která označí cestu od koncového uzlu ke kořenu

```
1: procedure MarkPathToRoot(n, eV, V, E)
2:     I ← vstupující hrany do n
3:     if I není prázdný then
4:         i ← vyber první hranu z I
5:         eV[i] ← true
6:         s ← získej počáteční uzel hrany i
7:         MarkPathToRoot(s, eV, V, E)
```

Poslední algoritmus (funkce 4) týkající se tvorby cyklu se snaží o vytvoření umělé hrany mezi náhodně vybranými uzly. Nejprve spočítá počet uzlů, mezi kterými může propojit hranu. Uzly musejí být logicky alespoň dva, jinak není možné vytvořit hranu. Poté se náhodně zvolí číslo $rFirst$, které určuje, kolikátý uzel v průchodu do hloubky se má vybrat jako první uzel a $rLast$ jako poslední. Následuje průchod do hloubky pomocí zásobníku s postupným ukládáním hloubky uzlu. Pokud narazíme na uzel s hloubkou $rFirst$, uložíme ho do $from$. Ten reprezentuje začátek hrany a později narazíme na uzel s hloubkou $rLast$, který reprezentuje její konec. Složitost je dána pouze průchodem do hloubky, jež byla zmíněna v úvodu.

Název parametru	Popis
V	množina uzlů
E	množina hran
$numInners$	počet vnitřních uzlů

Tabulka 3.4: Parametry funkce `CreateRandomEdge`

Algorithm 4 Funkce, která se pokusí o hrany mezi náhodně vybranými uzly

```
1:  function CreateRandomEdge(V, E, numInners)
2:      if numInners ≤ 1 then
3:          return ∅
4:      rFirst ← náhodné celé číslo z ⟨1, numInners − 1⟩
5:      rLast ← náhodné celé číslo z ⟨rFirst, numInners⟩
6:      INIT_STACK(S)
7:      INIT_ARRAY(L)
8:      r ← kořen grafu (V,E); PUSH(S, r); L[r] ← 0
9:      while S není prázdný do
10:         n ← POP(S)
11:         if L[n] = rFirst then
12:             from ← n
13:         for přes všechny odchozí hrany e uzlu n do
14:             to ← konec hrany e
15:             L[to] ← L[n] + 1
16:             if L[to] = rLast then
17:                 return {(from, to)}              ▷ vrať novou hranu
18:     return ∅
```

Funkce 5 se stará o připravení uzlu při průchodu do hloubky. S každým zavoláním udělá jeden krok DFS, přičemž využívá vstupní parametr S, který reprezentuje zásobník použit při průchodu do hloubky. Funkce předpokládá, že je v zásobníku přidán kořen grafu a pokud již v grafu nejsou další uzly, vrátí hodnotu *null*. Tato funkce je použita i při generování CRUD matice. Její složitost je rovna počtu hran uzlu v na vrcholu zásobníku, tedy $\mathcal{O}(|Adj[v]|)$.

Název parametru	Popis
g	graf
S	zásobník

Tabulka 3.5: Parametry funkce `PrepareNextNode`

Algorithm 5 Funkce, která získá další uzel při průchodu do hloubky

```
1: function PrepareNextNode(g, S)
2:     if S není prázdný then
3:         n ← POP(S)
4:         V ← výstupní hrany uzlu n grafu g seřazené podle koncového uzlu
   (od uzlu s nejvyšší výškou po nejnižší)
5:         for v ∈ V do
6:             t ← konec hrany v
7:             if t není navštívený then
8:                 t nastavit jako navštívený
9:                 PUSH(S, t)                          ▷ ulož do zásobníku
10:        return n
11:    else
12:        return null
```

Poslední pseudokód (algoritmus 6) vytvoří nové propojení mezi uzly z jiných grafů s aktuálním, které simuluje společné akce ve více procesech. Nejprve se seřadí akční uzly ze všech procesů podle jejich výšky (jak to lze provést je ukázáno v další sekci zabývající se návrhem algoritmu pro generování CRUD matice, konkrétně algoritmus 8), aby byla dodržena posloupnost operací. Následuje průchod do hloubky v grafu g a pokud se narazí na akci a ještě nebyl vyčerpán počet možných společných akcí, vytvoří se propojení *clone* a uloží se. Složitost závisí na seřazení vstupní množiny *actionsToSelect*, jejíž velikost můžeme nejprve redukovat na velikost *numActionNodes*, tzn. složitost řazení bude $\mathcal{O}(numActionNodes * \log(numActionNodes))$, pokud použijeme např. *MergeSort*. Následuje DFS, jehož složitost už jsme zmínili v úvodu. Složitost je tedy součet $\mathcal{O}(numActionNodes * \log(numActionNodes) + |V| + |H|)$.

Název parametru	Popis
C	seznam se sdílenými akcemi
g	graf
actionsToSelect	množina uzlů (akcí), z jiných grafů
numActionNodes	počet akčních uzlů v grafu g
numSameActions	počet žádoucích sdílených akcí v grafu g v procentech

Tabulka 3.6: Parametry procedury `AssignClones`

Algorithm 6 Procedura, která spojí akce z grafů s nově vytvořenými akcemi dle parametrů

```
1: procedure AssignClones(C, g, actionsToSelect, numActionNodes,
   numSameActions)
2:     nastav všechny uzly grafu g jako nenavštívené
```

```
3:    r ← kořen grafu g
4:    seřaď uzly actionsToSelect od nejníže položených po nejvýše položené
5:    aToSelect ← (numSameActions/100)∗ numActionNodes
6:    INIT_STACK(S); PUSH(S, r)
7:    do
8:        n ← PrepareNextNode(g, S)                    ▷ získá další uzel
9:        if n je akce ∧ aToSelect > 0 then
10:           aToSelect ← aToSelect − 1
11:           cSource ← vyjmi poslední prvek z actionsToSelect
12:           clone.from ← cSource; clone.to ← n        ▷ nastav propojení
13:           ADD(C, clone)                                ▷ ulož do pole
14:   while n ≠ null
```

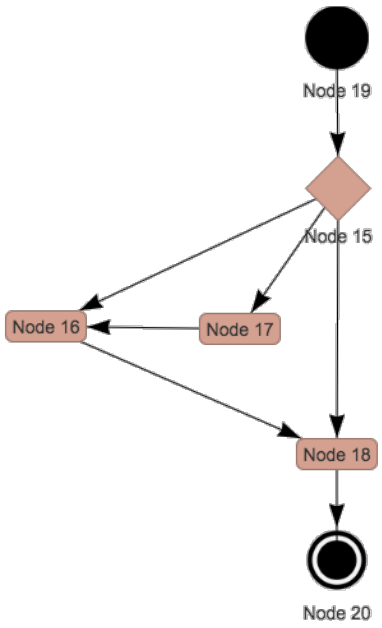
3.2 Generování CRUD matice

3.2.1 Úvod

CRUD matice, tak jak je definována v sekci 2.9.2, je v algoritmu reprezentována jako dvourozměrné pole. Základ algoritmu stojí na tom, že máme k uzlům vypočítanou jejich výšku. Výpočet je složitější, pokud graf obsahuje cykly. Postup je takový, že se nejprve graf převede na acyklický faktor grafu, který má všechny uzly dosažitelné z kořenu, na kterém spustíme algoritmus procházející postupně graf od listů ke kořenu a nastaví tak potřebnou výšku uzlu. Pokud máme určenou výšku uzlu, můžeme při průchodu určit, do kterého uzlu vkročit dříve a zaručit tak, že jedna akce nepředběhne jinou akci. Znázornění lze vidět na obrázku 3.1, na kterém je graf, při jehož průchodu musíme nejprve navštívit uzel 18, který má nejnižší výšku. Kdybychom šli totiž nejprve do uzlu 16 či 17 a přiřadili jim operaci (např. CREATE) nad nějakou entitou e_1, nemohli bychom přiřadit žádnou operaci modifikující e_1 uzlům položeným níže než uzel 18, protože stále existuje kratší cesta z uzlu 15 do 18, kterou bychom se mohli dostat k uzlu dříve a vykonat tak operaci nad entitou, která ještě nebyla vytvořena. Pro lepší popis časových složitostí definujeme n jako počet akcí (řádků) CRUD matice M a p jako počet entit (sloupců).

3.2.2 Popis algoritmu

V prvním bloku algoritmu se vytvoří potřebné datové struktury. První je jednorozměrné pole zásobníků S, jehož indexy jsou grafy z množiny grafů G, které má počáteční velikost $|G|$ s prázdnými zásobníky. Ty budou sloužit k průchodu do hloubky jednotlivými grafy. Druhou je jednorozměrné pole AS booleovských hodnot, které je indexováno uzly grafu a slouží k rozpoznání, zda už má akce přiřazenou nějakou operaci. Třetí je samotná CRUD matice reprezentována dvourozměrným polem, které má definovanou počáteční ve-



Obrázek 3.1: Ukázka grafu s předbíháním akcí

likost $numActionNodes \times numMinEntity$. Čtvrtou je jednorozměrné pole (indexováno grafy) polí (indexováno uzly) H, do kterého se napočítají výšky uzlů. Poslední, pátou strukturou je jednorozměrné pole W, které je indexováno uzly a slouží ke zjištění do jakého grafu uzel patří. Do zásobníků se přidají pro každý graf g kořeny, všechny uzly se nastaví jako bez operace a vypočítají se výšky uzlů pomocí procedury `SetHeight` (algoritmus 8). Následuje cyklus přes všechny grafy v náhodném pořadí, který vždy provede jeden krok průchodu do hloubky grafu, který vybere potencionální akci. Této akci přiřadí (pomocí procedury `AssignOperations`) generické operace (konkrétní přiřazení operace dle vstupního parametru $ratioRU$ nastane až na konci algoritmu voláním procedury `CorrectEntitiesAndOperations`) a zároveň nastaví k její společným akcím, aby se k nim už nepřiřazovaly žádné operace (pomocí procedury `AssignOperationsToClones`).

Celková složitost algoritmu je součtem jednotlivých bloků. Prvním je průchod všech zadaných grafů z množiny G. V něm se najde kořen, nastaví se všem uzlům status a zavolá se procedura `SetHeight`. První a třetí operace mají nejhorší složitost, která je rovna složitosti průchodu do hloubky DFS. Nechť je tedy množina AGV sjednocením uzlů všech grafů z AG a množina AGE sjednocením hran všech grafů AG. Pak je složitost prvního bloku $\mathcal{O}(|AGV| + |AGE|)$. Následuje `do-while` cyklus, ve kterém se prochází opět všechny grafy a každý z nich projde do hloubky. Průchod do hloubky je opět $\mathcal{O}(|AGV|+|AGE|)$, ale navíc se ještě volají dvě procedury, kde

`AssignOperations` má složitost $\mathcal{O}(np)$ a `AssignOperationsToClones` $\mathcal{O}(|C|)$. Každá z procedur je volána v nejhorším případě $|AGV|$-krát. Součet kroků cyklu pro jeden graf je reprezentován následujícím součtem

$$(|Adj[v_1]| + np + |C|) + (|Adj[v_2]| + np + |C|) + \ldots + (|Adj[v_k]| + np + |C|),$$

kde $k = |V|$. Součet můžeme přeskupit, protože platí následující výraz popsaný v [9]

$$\sum_{v \in V} |Adj[v]| = \Theta(|H|).$$

Po vyjádření těchto mezisoučtů a potřebném vytknutí vypadá vzorec takto

$$|H| + |V|(np + |C|).$$

Celková složitost `do-while` cyklu a zároveň celého algoritmu (protože vykonání procedury `CorrectEntitiesAndOperations` je asymptoticky stejně složité jako vykonání procedury `AssignOperations`) je tedy $\mathcal{O}(|AGE|+|AGV|\ (np+|C|))$.

Název parametru	Popis
numMinEntity	minimální počet entit
ratioRU	poměr mezi READ a UPDATE
AG	množina grafů
C	seznam sdílených akcí

Tabulka 3.7: Parametry funkce `CreateCrudMatrix`

Algorithm 7 Algoritmus vytvářející CRUD matici dle zadaných parametrů

```
1:  function CreateCrudMatrix(AG, C, numMinEntity, ratioRU)
2:      INIT_ARRAY_STACK(S, |AG|); INIT_ARRAY(AS)
3:      INIT_ARRAY_ARRAY(H); INIT_ARRAY(W); numActions ← 0
4:      for g ∈ AG do                                   ▷ pro každý graf
5:          r ← kořen grafu g; PUSH(S[g], r)      ▷ přidej kořen do zásobníku
6:          for přes všechny uzly n z grafu g do      ▷ všem uzlům nastav
7:              AS[n] ← false                  ▷ že operace ještě nebyla přiřazena
8:              if n je akce then
9:                  numActions ← numActions + 1
10:         SetHeight(g, H[g])                          ▷ nastav výšky uzlů
11:     INIT_ARRAY_ARRAY(M, numActions, numMinEntity)
12:     do
13:         nodesExists ← false
14:         for g ∈ AG do projdi v náhodném pořadí
15:             n ← PrepareNextNode(g, S[g])          ▷ získá další uzel
16:             if n ≠ null then          ▷ alespoň jeden uzel byl ještě přidán
```

```
17:             nodesExists ← true
18:             if n je akce ∧ AS[n] = false then    ▷ nevykonává operaci
19:                 W[n] ← g                          ▷ nastav na pozici uzlu n graf g
20:                 AssignOperations(n, AS, M, H[g], W)
21:                 AssignOperationsToClones(a, C, AS)
22:     while nodesExists = true▷ opakuj, dokud existuje nenavštívený uzel
23:     CorrectEntitiesAndOperations(M, ratioRU)
24:     return M
```

Algoritmus `setHeight` určí hloubku každému z uzlů grafu g. Princip je takový, že se vytvoří acyklický faktor grafu (funkce `CreateAcyclicFactor` v pseudokódu 9) a zavolá se procedura `GoUpToRootAndSetHeight` (pseudokód 11), která projde graf *acyclicFactor* směrem od listů ke kořenu a v každé úrovni průchodu zvýší výšku uzlu.

Složitost algoritmu je $\mathcal{O}(|V|+|E|)$, protože všechny operace v něm uvedené mají právě tuto nebo nižší složitost.

Název parametru	Popis
g	graf
H	jednorozměrné pole s výškami uzlů

Tabulka 3.8: Parametry procedury `SetHeight`

Algorithm 8 Procedura, která nastaví uzlům jejich výšku

```
1: procedure SetHeight(g, H)
2:     acyclicFactor ← CreateAcyclicFactor(g)
3:     leafs ← získej všechny listy grafu acyclicFactor
4:     nastav všechny uzly grafu acyclicFactor jako nenavštívené
5:     GoUpToRootAndSetHeight(leafs, H)
```

Funkce `CreateAcyclicFactor` vytvoří z grafu acyklický faktor. V prvním kroku vytvoří kopii grafu (aby neměnila strukturu grafu v argumentu funkce). Inicializuje tři množiny W (white - bílý), G (gray - šedý) a B (black - černý), které reprezentují stav, ve kterém se daný uzel v průchodu do hloubky nachází. Uzly v množině W ještě nebyly navštívené, uzly v množině G již byly navštívené, ale ještě nebyly zpracovaní jejich potomci a uzly v množině B už jsou zcela zpracovány. Množiny spolu s množinou R, kam se uloží hrany, které tvoří cyklus, se předají jako parametr do procedury `FindEdgesCreatingCycles`. Nakonec se z grafu odstraní nalezené zpětné hrany tvořící cyklus. Pokud odstraníme všechny zpětné hrany, které najdeme během DFS, už další najít nemůžeme (odstraněním hrany nemůžeme vytvořit nový cyklus). Následující lemma nám říká, že vzniklý graf bude skutečně acyklický.

Lemma 1 ([12])**.** Orientovaný graf G je acyklický právě tehdy, když průchod do hloubky grafem G nevyprodukuje žádnou zpětnou hranu.

Složitost algoritmu je $\mathcal{O}(|V|+|E|)$, protože všechny operace v něm uvedené mají právě tuto nebo nižší složitost.

Název parametru	Popis
g	graf

Tabulka 3.9: Parametry funkce `CreateAcyclicFactor`

Algorithm 9 Funkce, která vytvoří z daného grafu acyklický faktor

```
1: function CreateAcyclicFactor(g)
2:     factor ← vytvoř kopii grafu g
3:     r ← kořen grafu factor
4:     W ← ∅; G ← ∅; B ← ∅; R ← ∅
5:     for přes všechny uzly v grafu factor do
6:         W ← W ∪ {v}
7:     FindEdgesCreatingCycles(r, R, W, G, B)
8:     E ← hrany grafu factor; E ← E \ R
9:     return factor
```

Procedura `FindEdgesCreatingCycles` popisuje algoritmus, který je založen na DFS algoritmu popsaném v [12] obohacen třemi množinami (v knize *Introduction to Algorithms* jsou místo množin využity barevné značky). Na konci rekurzivního běhu musí platit, že $B = V$, kde V je množina uzlů grafu. To je stav, kdy jsou všechny uzly navštívené a zpracované. Cyklus vznikne právě tehdy, když vstoupíme na zpětnou hranu, neboli když navštívíme uzel, který jsme zatím zcela nezpracovali a zároveň jsme ho už ale navštívili (je obsažen v množině G, která reprezentuje rekurzivní zásobník, tedy obsahuje uzly, kterými jsme prošli od kořenu k aktuálnímu uzlu). V takovém případě přidáme hranu do R a dále v rekurzi nepokračujeme.

Složitost odpovídá složitosti procedury `FindEdgesCreatingCycles`, tedy $\mathcal{O}(|V| + |E|)$.

Název parametru	Popis
v	kořen grafu
R	množina hran způsobující cykly
W	množina ještě nenavštívených uzlů
G	množina již navštívených, ale nezpracovaných uzlů
B	množina již zpracovaných uzlů

Tabulka 3.10: Parametry funkce `CreateAcyclicFactor`

Algorithm 10 Procedura, která najde hrany, jejichž odebráním vznikne acyklický graf

```
1:  procedure FindEdgesCreatingCycles(v, R, W, G, B)
2:      W ← W \ {v}; G ← G ∪ {v}                    ▷ přesuň uzel z W do G
3:      for přes všechny odchozí hrany e uzlu v do
4:          to ← konec hrany e
5:          if B ∩ {to} = ∅ then                   ▷ pokud jsme uzel ještě nezpracovali
6:              if G ∩ {to} ≠ ∅ then               ▷ a pokud ho máme na zásobníku rekurze
7:                  R ← R ∪ {to}                   ▷ tvoří cyklus a proto přidáme hranu do R
8:              else                               ▷ jinak pokračuj v rekurzivním DFS průchodu
9:                  FindEdgesCreatingCycles(v, R, W, G, B)
10:     G ← G \ {v}; B ← B ∪ {v}                    ▷ přesuň uzel z G do B
```

Procedura `GoUpToRootAndSetHeight` je založena na podobném principu, jako je tvorba *topologického uspořádání*[8] grafu popsaného v [12]. Funguje tak, že se vezmou listy grafu a ty se z grafu spolu s jejich vstupními hranami odstraní. Tím se zároveň sníží výstupní stupeň uzlů, které byly začátkem těchto hran. Jelikož je graf acyklický, musí vzniknout alespoň jeden nový list. Pokud nevznikne, došli jsme až ke kořenu. V každém kroku odstranění se zvýší výška nově vzniklého listu. Implementace je provedena pomocí fronty, protože potřebujeme procházet graf do šířky.

Složitost odpovídá složitosti průchodu do šířky (BFS[9]), tedy $\mathcal{O}(|V|+|E|)$.

Název parametru	Popis
leafs	listy grafu
H	jednorozměrné pole s výškami uzlů

Tabulka 3.11: Parametry procedury `GoUpToRootAndSetHeight`

Algorithm 11 Procedura, která nastaví uzlům jejich výšku průchodem směrem od listů ke kořenu

```
1:  procedure GoUpToRootAndSetHeight(leafs, H)
2:      INIT_QUEUE(Q)
3:      for leaf ∈ leafs do
4:          H[leaf] ← 0
5:          ENQUEUE(Q, leaf)
6:      while Q není prázdná do
7:          v ← DEQUEUE(Q)
8:          for přes všechny vstupní hrany e uzlu v do
```

[8] Takové seřazení uzlů orientovaného acyklického grafu, které když vyskládáme horizontálně vedle sebe, tak všechny hrany směřují zleva doprava.

[9] Breadth First Search (BFS) je metoda traverzování grafu do šířky.

```
9:          from ← počáteční uzel hrany e
10:         snížíme výstupní stupeň uzlu from o jedničku
11:         if from nebyl navštíven ∧ výstupní stupeň from ≤ 0 then
12:             nastav from jako navštívený
13:             H[from] ← H[v] + 1
14:             ENQUEUE(Q, from)
```

Pseudokód popsán v proceduře `AssingOperations` se pokusí přidat k nějaké již existující entitě (sloupce matice M) operaci, kterou provede akce a. Myšlenka algoritmu je taková, že se vždy projdou již přiřazené operace k dané entitě seřazené od naposledy přidané. Pokud se podaří najít akci (s již přiřazenou generickou operací), která je ze stejného grafu jako a, musíme porovnat jejich výšku v grafu. Povolíme přiřazení operace pouze a jen tehdy, pokud je výška uzlu a nižší než hloubka nalezené akce. Jestliže ale žádnou takovou akci nenajdeme (ze stejného grafu), můžeme přidat akci bez jakékoli další kontroly (to nastane pokud přidáváme první operaci k dané entitě obecně, nebo pokud přidáváme první operaci, kterou provádí akce z jiného procesu).

Složitost určíme opět z jednotlivých bloků kódu. První řádek získává index řádku v matici M pro akci a, což by šlo zrealizovat v konstantním čase, pokud by se udržovalo další pole, které by mělo jako indexy uzly a jeho hodnoty by byly indexy do pole M. Následuje cyklus přes všechny entity e a pro každou se projdou akce, které nad ní vykonávají již nějaké operace. Tyto akce musí být seřazené, což by šlo opět provést v konstantním čase, pokud by se v matici udržovala inkrementální číselná hodnota jasně definující (např. největší) poslední operaci. Poté se projdou všechny takové akce. Složitost se rovná v nejhorším případě průchodu celé matice M, tedy $\mathcal{O}(np)$.

Název parametru	Popis
a	uzel (akce)
AS	jednorozměrné pole do kterého se nastaví, že byla akci přiřazena operace
M	dvourozměrné pole reprezentující CRUD matici
H	jednorozměrné pole s výškami uzlů
W	jednorozměrné pole indikující z jakého grafu uzel pochází

Tabulka 3.12: Parametry procedury `AssignOperations`

Algorithm 12 Procedura, která nastaví uzlu status, že může vykonávat nějakou operaci nad entitou

```
1: procedure AssignOperations(a, AS, M, H, W)
2:     i ← index řádku v matici M pro akci a
3:     for přes všechny sloupce j CRUD matice M do
```

```
4:        l ← null
5:        A ←akce sestupně od naposledny přidané operace nad entitou s
   indexem j
6:        for action ∈ A do
7:            if W[action] = W[a] then        ▷ pokud jsou ze stejného grafu
8:                l ← action                                          ▷ ulož
9:                break
10:       if l = null ∨ H[a] < H[l] then      ▷ pokud operace v tomto grafu
   ještě nebyla přiřazena nebo pokud byla, ale uzel který ji vykonává je výš
   než aktuální uzel
11:           AS[a] ← true
12:           M[i][j] ← GENERIC
13:   if AS[a] = false then      ▷ pokud se nepodařilo uzlu přiřadit operaci
14:       AS[a] ← true
15:       e ← index posledního sloupce + 1
16:       přidej do M nový sloupec na pozici e
17:       M[i][e] ← GENERIC
```

Procedura `AssignOperationsToClones` nastaví všem akcím, které jsou společné s akcí a status, že už mají přiřazenou operaci, abychom zachovali stejnou společnou množinu operací pro všechny stejné akce z více procesů.

Složitost se rovná projití seznamu C, tedy $\mathcal{O}(|C|)$.

Název parametru	Popis
a	uzel (akce)
C	seznam sdílených akcí
AS	jednorozměrné pole do kterého se nastaví, že byla akci přiřazena operace

Tabulka 3.13: Parametry procedury `AssignOperationsToClones`

Algorithm 13 Procedura, která nastaví sdíleným akcím status přiřazení operace

```
1: procedure AssignOperationsToClones(a, C, AS)
2:     for c ∈ C do
3:         if c.from = a ∨ c.to = a then
4:             AS[c.from] ← true; AS[c.to] ← true;
```

Poslední pseudokód popisuje algoritmus, který řeší samotné rozmístění CRUD operací ve volných místech. V prvním kroku provede kontrolu, zda existují alespoň tři operace nad danou entitou se stavem $GENERIC$, pokud ne, nelze vyhovět podmínce určující minimální sled akcí (více v sekci 2.9.2).

Dále se provede náhodné rozmístění tak, že se určí náhodný index první (CREATE) a poslední (DELETE) operace. Poté projde v náhodném pořadí sloupec (reprezentující entitu) CRUD matice M a přiřadí se mu dle podmínek příslušná operace.

Složitost je rovna průchodu všech prvků dvourozměrného pole (matice) M, tedy $\mathcal{O}(np)$.

Název parametru	Popis
M	dvourozměrné pole reprezentující CRUD matici
$ratioRU$	poměr READ ku UPDATE

Tabulka 3.14: Parametry procedury `CorrectEntitiesAndOperations`

Algorithm 14 Procedura, která nastaví výsledné CRUD operace nad entitou

1: **procedure** CorrectEntitiesAndOperations($M, ratioRU$)
2: **for** přes všechny sloupce j CRUD matice M **do**
3: $opSize \leftarrow$ počet operací = GENERIC ve sloupci j matice M
4: **if** $opSize < 3$ **then**
5: odstraň entitu na pozici j
6: **else**
7: $nAsOps \leftarrow$ náhodné celé číslo z $\langle 3, opSize \rangle$ ▷ počet operací
8: $cRU \leftarrow nAsOps - 2$ ▷ operace bez CREATE a DELETE
9: $nR \leftarrow \lfloor cRU * ratioReadUpdate \rfloor$ ▷ počet operací READ
10: $nU \leftarrow cRU - nR$ ▷ počet operací UPDATE
11: $offset \leftarrow opSize - nAsOps$ ▷ počet volných míst
12: $rlOffset \leftarrow$ náhodné celé číslo z $\langle 0, offset \rangle$
13: $rrOffset \leftarrow$ náhodné celé číslo z $\langle 0, offset - rlOffset \rangle$
14: $first \leftarrow rlOffset$ ▷ index první operace (C)
15: $last \leftarrow opSize - 1 - rrOffset$ ▷ index poslední operace (D)
16: **for** náhodně projdi sloupec j s iterační proměnnou i **do**
17: **if** $i = first$ **then**
18: $M[i][j] \leftarrow CREATE$
19: **else if** $i = last$ **then**
20: $M[i][j] \leftarrow DELETE$
21: **else if** $i > first \wedge i < last \wedge cRU > 0 \wedge nU > 0$ **then**
22: $M[i][j] \leftarrow UPDATE$
23: $nU \leftarrow nU - 1$
24: **else if** $i > first \wedge i < last \wedge cRU > 0$ **then**
25: $M[i][j] \leftarrow READ$

3.3 Generování umělých chyb

3.3.1 Úvod

Aplikace umožňuje generování umělých chyb. Algoritmus 15 má za úkol vytvořit posloupnost akčních uzlů dle zadaných parametrů, které když se v umělé aplikaci vykonají, způsobí její pád.

3.3.2 Popis algoritmu

Algoritmus pro hledání posloupností akcí funguje tak, že se nejprve najdou některé posloupnosti akcí z kořenu k listům a z těchto cest se pak jedna náhodně vybere. Funkce `RandomizePath` pak určí posloupnosti správnou délku a náhodně v ní rozloží uzly. Funkce `FindPaths` je založena na velmi podobném principu, jako funguje procedura `CreateAcyclicFactor` (algoritmus 9), jež byla popsána v předešlé sekci. Opět se prochází do hloubky, ale jakmile se dostaneme k cíli (k listu grafu), uložíme obsah (vybereme jen akce) rekurzivního zásobníku (nyní G představuje seznam, jelikož potřebujeme udržet správnou posloupnost uzlů) do nového seznamu, který uložíme do APL. APL je seznam seznamů, který bude obsahovat nalezené posloupnosti uzlů. Z APL se náhodně vybere jedna posloupnost a ta se vloží do volání funkce `RandomizePath`. Složitost celého algoritmu je složením složitostí funkce `FindPaths` a funkce `RandomizePath`. Celková časová asymptotická složitost je $\mathcal{O}(|E| + s|V| + minLength)$, nicméně $minLength$ nemůže být větší než celkový počet uzlů a proto jej zanedbáme a složitost bude $\mathcal{O}(|E| + s|V|)$, kde s je počet všech jednoduchých cest (cesty, ve kterých se neopakují žádné uzly) mezi kořenem a listy.

Název parametru	Popis
g	graf
$minLength$	minimální velikost posloupnosti uzlů

Tabulka 3.15: Parametry funkce `GenerateActionErrorCombination`

Algorithm 15 Funkce, která vytvoří procesní chyby

```
1: function GenerateActionErrorCombination(g, minLength)
2:     APL ← FindPaths(g, minLength)
3:     if SIZE(APL) = 0 then        ▷ pokud neexistuje žádná nalezená cesta
4:         return vrať prázdný seznam
5:     P ← náhodně vybraná cesta z APL
6:     return RandomizePath(P, minLength)
```

Algorithm 16 Funkce, která najde posloupnosti uzlů dané délky

Název parametru	Popis
g	graf
minLength	minimální velikost posloupnosti uzlů

Tabulka 3.16: Parametry funkce `FindPaths`

1: **function** FindPaths($g, minLength$)
2: INIT_LIST_LIST(APL); INIT_LIST(G); $B \leftarrow \emptyset$
3: $W \leftarrow$ uzly grafu g; $r \leftarrow$ kořen grafu g
4: FindPathsToLeafs($r, APL, W, G, B, minLength$)
5: **return** APL

Složitost procedury `FindPathsToLeafs` je opět založena na průchodu do hloubky, ale navíc v něm dochází ke kopírování prvků množiny G (jejíž počet je shora omezen počtem všech uzlů), když dojdeme do listu grafu. Nechť $k = |V|$ a $s_1 + s_2 + \ldots + s_k = s$, kde s je roven součtu všech jednoduchých cest (vyprodukované průchodem do hloubky) mezi kořenem grafu a jeho listy. Pak můžeme jednotlivé kroky algoritmu popsat součtem

$$(|Adj[v_1]| + (s_1\ |V|)) + (|Adj[v_2]| + (s_2\ |V|)) + \ldots + (|Adj[v_k]| + (s_k\ |V|)),$$

který lze zjednodušit

$$|E| + |V|(s_1 + s_2 + \ldots + s_k) = |E| + |V|s.$$

Složitost je tedy $\mathcal{O}(|E| + s|V|)$.

Název parametru	Popis
v	počáteční uzel z kterého se bude procházet do hloubky
APL	seznam, který bude naplněn posloupnostmi uzlů (akcí)
W	množina ještě nenavštívených uzlů
G	seznam již navštívených, ale nezpracovaných uzlů
B	množina již zpracovaných uzlů
minLength	minimální délka posloupnosti uzlů

Tabulka 3.17: Parametry funkce `FindPaths`

Algorithm 17 Pomocná procedura, která projde graf do hloubky a uloží posloupnosti uzlů z kořenu do listu

1: **procedure** FindPathsToLeafs($v, APL, W, G, B, minLength$)
2: $W \leftarrow W \setminus \{v\}$; $G \leftarrow G \cup \{v\}$ ▷ přesuň uzel z W do G

```
3:    for přes všechny odchozí hrany e uzlu v do
4:        to ← konec hrany e
5:        if uzel to je list then          ▷ pokud jsme došli až na konec grafu
6:            INIT_LIST(P)
7:            for přes všechny akce a z G do          ▷ přidáme všechny akce
8:                ADD(P, a)
9:            if SIZE(P) ≥ minLength then          ▷ podmínka splněna
10:               ADD(APL, P)                          ▷ uložíme posloupnost
11:       if B ∩ {to} = ∅ ∧ G neobsahuje to then
12:           FindPathsToLeafs(v, APL, W, G, B)
13:   G ← G \ {v}; B ← B ∪ {v}                          ▷ přesuň uzel z G do B
```

Poslední funkce `RandomizePath` náhodně rozloží uzly v nové posloupnosti s přesně danou velikostí. Pokud je v nové posloupnosti R dostatek volných míst, nechá se přiřazení uzlů na náhodě, pokud už je počet zbývajících uzlů k přidaní roven počtu volných míst, nic se nerandomizuje, ale rovnou se ty to uzly přidají, aby byla zajištěna definovaná délka posloupnosti.

Složitost je dána vstupním parametrem *length*, tedy $\mathcal{O}(length)$.

Název parametru	Popis
P	seznam uzlů, který se bude randomizovat
length	požadovaná délka posloupnosti

Tabulka 3.18: Parametry funkce `RandomizePath`

Algorithm 18 Funkce, která z cest vytvoří náhodnou kombinaci

```
1: function RandomizePath(P, length)
2:     INIT_LIST(R); pathSize ← SIZE(P); i ← 0
3:     while i < pathSize ∧ SIZE(R) < length do
4:         toAddLeft ← length−SIZE(R)
5:         freeSlots ← pathSize − i
6:         random ← náhodné reálné číslo z rozmezí ⟨0, 1⟩
7:         if toAddLeft ≥ freeSlots ∨ random ≥ 0.5 then
8:             ADD(R,GET(P, i))
9:         i ← i + 1
10:    return R
```

Kapitola 4

Implementace

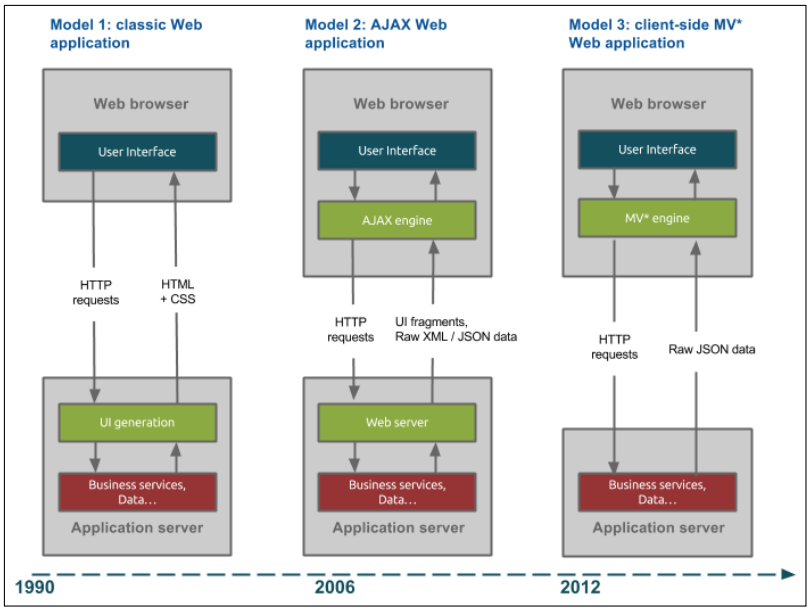
V této kapitole bude nejprve popsána architektura aplikace a proč byla takto zvolena. Budou zmíněné použité frameworky a technologie (jak už pro backend[10], či frontend[11]) a jaký byl důvod jejich zvolení. Následovat bude výběr a popis nejdůležitějších balíčků a tříd aplikace. Dále pak bude uveden diagram nasazení spolu s databázovým návrhem. Na závěr bude popsána infrastruktura, která byla využita při vývoji aplikace.

4.1 Architektura aplikace

Prvním krokem před samotnou implementací je zvolení vhodné architektury a rozvržení jednotlivých vrstev (pokud jich je více), které se starají o dílčí úlohy. V zadání práce je jednoznačně řečeno, že má být výsledkem webová aplikace. Z tohoto důvodu jsem se zaměřil na tři modely (na obrázku 4.1) webových aplikací, které se obecně používají.

Postupně tyto tři modely projdu směrem, který znázorňuje i časová osa, od nejstarší technologie po nejnovější trend. Prvním je klasická webová aplikace patřící do skupiny `klient-server`, která ze směru od klienta komunikuje čistě skrz HTTP[12] protokol s webovým serverem, který na dotaz požadavku odpoví spolu s výslednou HTML[13] stránkou. Druhým modelem je opět `klient-server` architektura založená na AJAX[14] metodách, které se chovají jako prostředník mezi klientem a serverem obstarávající komunikaci a výměnu dat, aniž by to klient v prohlížeči pocítil (nedochází k znovu načtení stránky). Poslední model patří do skupiny klientských aplikací a už jeho název napovídá, že veškerá logika běží na klientovi a server se využívá např. pro databázi.

[10] Jde typicky o serverovou část v architektuře klient-server.
[11] V architektuře klient-server jde o tu část, která se stará o vizuální zobrazení klientovi.
[12] Hypertext Transfer Protocol (HTTP)
[13] HyperText Markup Language (HTML)
[14] Asynchronous JavaScript and XML (AJAX)

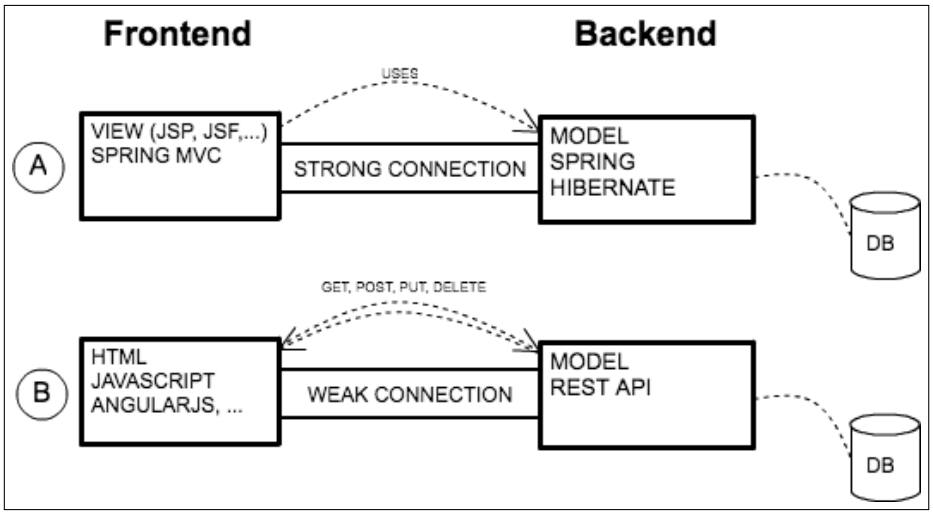


Obrázek 4.1: Porovnání modelů webových aplikací [1]

4.1.1 Volba komponent

Pro výběr vhodných komponent aplikace byly navrženy dvě varianty typu `klient-server`, jež jsou kombinací prvních dvou modelů z obrázku 4.1. Obě dvě předpokládaly znalosti Javy (EE), Javascriptu[15], REST[16] API a HTML a braly v potaz, že bude použita Javascriptová knihovna na vykreslování grafu. Obě varianty jsou zobrazeny na obrázku 4.2. Varianta označena písmenem `A` funguje jako jednotná, celistvá komponenta, u níž dochází k propojení uživatelského rozhraní a serverové části v tom smyslu, že se výsledná stránka generuje na serveru a ta je následně poslána uživateli. Zatímco druhá varianta `B` vystavuje REST API na serveru, které využívá (pomocí HTTP metod GET, POST, PUT a DELETE) např. AngularJS[17] s využitím klientského Javascriptu. Varianta `B` má výhodu v tom, že by bylo jednodušší propojení mezi javascriptovou knihovnou pro zobrazení grafu a komunikací mezi klientem a serverem (která by také byla v Javascriptu). Nicméně i přes tuto výhodu varianty `B` byla zvolena varianta `A` jakožto přímočařejší cesta z modelu napsaném v Javě a také proto, že javascriptový grafový editor nevyžaduje v aplikaci až tak velké nároky na implementaci.

[15] Skriptovací jazyk na straně klienta.
[16] Representational State Transfer (REST)
[17] Framework umožňující psát webové aplikace na klientské straně.



Obrázek 4.2: Porovnání dvou možných kombinací technologií

4.1.2 Popis architektury

Na aplikaci lze nahlížet jako na vícevrstvou webovou aplikaci, jejíž hierarchické (každá vrstva využívá tu pod ní) členění lze vidět na obrázku 4.3. Vykonání operací směrem dolů má tyto kroky:

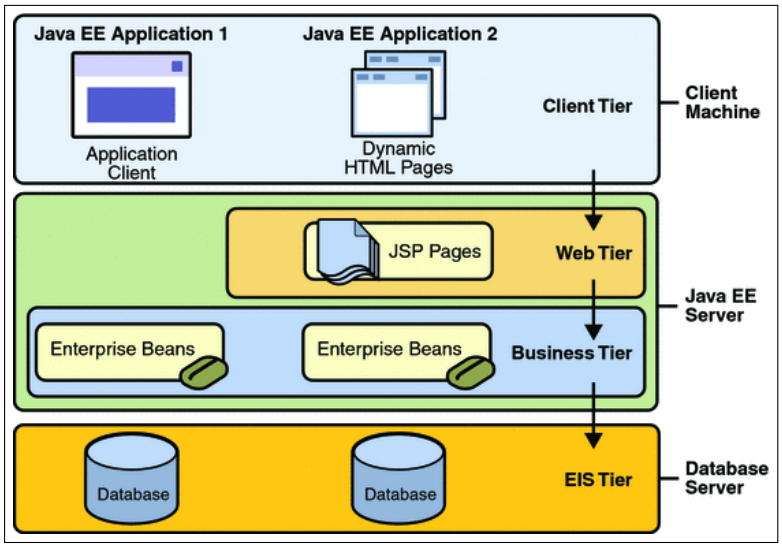
1. Klient zadá požadavek na konkrétní url.
2. Druhá (webová) vrstva obdrží požadavek a na základě url zvolí akci v business vrstvě a následně vybere konkrétní dynamickou stránku se všemi potřebnými daty.
3. Třetí (business) vrstva obstará logiku nad datovým modelem.
4. Čtvrtá (datová) vrstva poskytne perzistentní data.

Poté, co druhá vrstva vykoná business operaci, aplikace má veškerá data pro tvorbu dynamické JSP stránky, kterou pošle klientovi ve formě webové stránky. V následující sekci budou popsány konkrétní technologie využité pro každou ze zmíněných vrstev.

4.2 Prezentační část

Pro vytváření dynamických stránek (druhá vrstva z obrázku 4.3) byla zvolena technologie JSP[18], jenž je součástí Java EE (dříve J2EE) od jejího vzniku roku 1999 [15]. Výběr byl založen na pohodlnosti a jednoduchosti použití pro

[18] JavaServer Pages (JSP)



Obrázek 4.3: Vícevrstvá aplikace [2]

relativně malý počet stránek. I přesto, že je JSF[19] standardem Javy EE pro tvoření dynamických stránek a umožňuje lepší šablonovací systém, zvolil jsem čisté JSP s template taglibs[20]. Postup předání JSP stránky klientovi lze popsat v těchto krocích:

1. Klient zadá požadavek na konkrétní url.
2. Server se podívá zda již byla JSP stránka žádaná předtím, pokud ne, je přeložena do třídy dědící `HttpServlet`[21] a uložena na serveru pro další použití. Pokud již byla `HttpServlet` třída takto vytvořená, použije se znovu.

4.2.1 Grafická knihovna pro vizualizaci procesu

Pro vykreslování grafu byla zvolena knihovna `vis.js`, která disponuje těmito vlastnostmi:

- Snadná instalace knihovny.
- Jednoduchá struktura modelu grafu (termín graf není ve specifikaci použit, místo toho se používá network (síť)).
- Možnost změny vzhledu uzlů, hran či popisků uzlů.

[19] JavaServer Faces (JSF)
[20] Template Tag Library umožňuje tvořit jednoduché šablony pro JSP.
[21] Servlet je Java API běžící na serveru sloužící ke zpracování HTTP dotazů.

- Obsahuje základní nástroje pro interakci s uživatelem, kterými jsou přidání uzlu, přidání hrany, posun uzlu, smazání uzlu a smazání hrany.
- Schopnost automatického rozložení uzlů tak, aby se nepřekrývaly.
- Licencovaná pod Apache 2.0 License a MIT License.

Začlenění této knihovny do aplikace je také důvod, proč je v aplikaci použit Javascript spolu s AJAX. Neboť po vygenerování dat (grafu) na serveru je potřeba tyto data dostat do knihovny a zobrazit graf.

4.3 Serverová část

4.3.1 Spring

Výběr architektury umožnil zvolit jako pomocný open-source[22] framework Spring, který je v současné době velmi populární a to hlavně díky své komunitě. Jeho jádrem je princip zvaný *inversion of control*, který přesouvá zodpovědnost za vytváření objektů z aplikace na framework [16]. Někdy je také označován jako IoC kontejner. K vytváření objektů využívá *dependency injection*[23], která vytvoří daný objekt, když je ho potřeba. Konfigurace Springu se provádí skrz tzv. aplikační kontext (soubor XML) nebo pomocí anotací.

Pro správnou terminologii je nutné zmínit, jak vypadá typická URL servletu na Java serveru. Vypadá následovně (ukázka převzata z [18])

```
http://hostname.com/contextPath/servletPath/pathInfo
```

a popis jejích jednotlivých částí:

- *contextPath* - podle ní vybírá java server správnou aplikaci.
- *servletPath* - z dané aplikace vybere registrovaný servlet v souboru web.xml.
- *pathInfo* - popisuje adresářovou (hiearchickou) strukturu aplikace.

4.3.2 Spring MVC

Samotné jádro Springu lze použít i ve standalone[24] aplikacích a v naší aplikaci je také využito (např. pro konfiguraci databáze či jako IoC kontejner). Nicméně pro zpracování HTTP požadavků byl zvolen webový framework Spring MVC. Ten umožňuje díky anotaci `@Controller` oanotovat třídu, kterou bude Spring považovat za vstupní bod k obsluze požadavků. V této třídě lze použít anotaci `@RequestMapping`, která se může přidat na úroveň třídy nebo k samotným

[22] Označení otevřeného a dostupného zdrojového kódu.
[23] Technika pro vkládání závislostí mezi jednotlivými komponentami [17]
[24] Aplikace běžící lokálně na počítači na kterém byly nainstalovány.

metodám, které obslouží požadavek. Pokud je anotace nad třídou, určují kořenovou cestu k *servletPath* a její metody pak určují relativní cesty k této nadřazené kořenové cestě. Pokud je anotace pouze nad metodami, pak jsou její parametry absolutními cestami. Každá metoda vrací obvykle objekt typu `ModelAndView` nebo název konkrétní dynamické stránky (lze definovat prefixy - adresářovou strukturu a postfixy těchto stránek v konfiguračním XML). Metody můžou mít argument typu `Model` obsahující data, ke kterým lze přistupovat v dynamické stránce. Tento model lze libovolně modifikovat v průběhu zpracování požadavku (např. přidání nově vytvořené CRUD matice). Tyto činnosti jsou ve skutečnosti obsaženy ve webové vrstvě popsané v 4.1.2, po ní následuje business vrstva, která už se stará o logiku a využívá další vrstvy k ukládání dat. Následuje ukázka anotací nad třídou a metodou, která by měla validovat graf. Zároveň ukazuje, že je možné pomocí anotace `@PathVariable` získat hodnotu (uzavřenou do složených závorek v URL cestě) z URL.

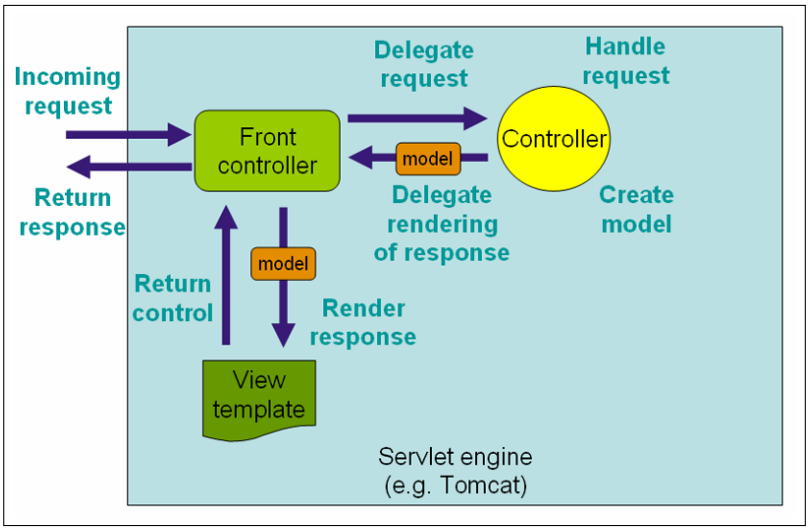
Listing 4.1: Ukázka anotací Spring MVC Controlleru

```
@Controller
@RequestMapping(value = "/workspace/{workspaceId}")
public class WorkflowController {

 @RequestMapping(value = "/workflow/{workflowId}/validateGraph",
  method = RequestMethod.POST)
 public String validateGraph(
  @PathVariable("workflowId") Long workflowId,
  @PathVariable("workspaceId") Long workspaceId, Model model) {
  // validate graph from the given workspaceId/workflowId identifier
  // add result of validation to the spring's model
  return "workflowView";
 }

}
```

Jak funguje Spring MVC na pozadí, znázorňuje obrázek 4.4 (převzatý spolu s informacemi o něm z [3]). První vrstvou je tzv. *Front controller*, který vyvolá `DispatcherServlet`, což je třída Springu, která dědí od třídy `HttpServlet`, jež je součástí Java EE. `DispatcherServlet` je definován jako každý jiný servlet spolu s URL mapováním (při kterém se servlet vyvolá) v souboru web.xml. Jeho úkol je delegování požadavků na `Controller`, který naplní `Model` objekty pro využití na dynamické stránce. Ten se předá vykreslovacímu modulu a vytvoří tak skutečnou stránku připravenou k odeslání klientovi.



Obrázek 4.4: Interakce na pozadí mezi servletem a Spring Controllerem [3]

4.3.2.1 Znovupoužití kódu

Jak už jsem popsal, k zpracování HTTP požadavku slouží „obsluhující“ metody s anotací `@RequestMapping` vztahující se ke stejné kořenové url v daném `Controller`u. Pokud nějaká stránka potřebuje zobrazovat tytéž informace (např. získané z databáze) s každým dotazem, bylo by nežádoucí (doporučující princip DRY[25], který vede k lépe udržitelnému kódu) psát stejný kus kódu v každé metodě. Pro tento problém lze definovat tzv. *interceptory*, neboli prvky, které dokáží zachytit požadavek před nebo po zavolání „obsluhující“ metody. Interceptor HTTP požadavků je třída dědící z `HandlerInterceptorAdapter`, která definuje dvě metody pro vykonání potřebných operací před (`preHandle`) zavoláním obsluhující metody a po (`postHandle`) zavolání. Interceptory se definují v konfiguračním XML v tagu `mvc:interceptors` a vždy se definují pro konkrétní relativní url, aby bylo jednoznačné, kdy se mají vyvolat.

4.3.2.2 Validace

Získání formulářových dat tak, aby byla možná jednoduchá validace, začne definováním třídy, která definuje vlastnosti formuláře. Následují tyto kroky:

1. Definice metody m, která má parametr a typu `Formular`.
2. Vložení objektu c typu `Formular` do `Modelu` springu např. v `preHandle` metodě interceptoru.

[25] Don’t Repeat Yourself (DRY) princip říká, že by kód neměl obsahovat duplicitní části kódu.

3. Poslání požadavku, který vyvolá metodu *m*.

4. Parametr *a* metody *m* obsahuje data z formuláře.

Aby byly vlastnosti formulářového objektu validovány, musí být specifikováno, jak má být validace provedena. To lze provést tak, že se k jednotlivým instančním proměnným přidají anotace z balíčku `javax.validation.constraints`. Samotná validace se sama od sebe nespustí, musí se ještě přidat anotace `@Valid` k argumentu obsluhující metody. Tím se zajistí automatická validace dat a pokud chceme vědět, jak validace dopadla, přidáme ještě jeden argument metody (musí následovat hned za argumentem formulářového objektu) typu `BindingResult`. Tento objekt obsahuje výsledek validace a voláním metody `hasErrors` jednoduše zjistíme, zda prošel validací či nikoliv.

4.3.2.3 Lokalizace

Lokalizace aplikace pomocí springu je velmi jednoduchá. Vše je založeno na tom, že se vytvoří dva či více souborů (koncovka `.properties`), které mají název definovaný jako `nazev_{lokalita_id}`, kde `{lokalita_id}` je kód určující jazyk, zemi a stát (pro angličtinu `en`, pro češtinu `cs`). V těchto souborech, jak je u souborů `.properties` zvykem, jsou obsaženy dvojice řetězců `klíč=hodnota`, kde `klíč` je unikátní pro daný soubor. V souborech se nadefinuje veškerý text použit v dynamických stránkách. Konfigurace probíhá opět v aplikačním kontextu (XML soubor), kde se nastaví počáteční jazyk a název parametru, který umožní měnit lokalizaci za běhu pomocí URL parametru. Ve stránkách se pak v místech, kde by se normálně psal text zadá tag `<spring:message code="klíč">`.

4.4 Popis nejdůležitějších balíčků v kódu

Nejprve uvedu tabulku se základními Java balíčky, které popíšu. Všechny balíčky mají stejný začátek `cz.cvut.fit`. Toto pojmenování vychází ze jmenné konvence, která se používá pro pojmenování balíčků. Balíček by totiž měl začít obráceným jménem domény, ke které se aplikace vztahuje. A jelikož je práce vedena na fakultě informačních technologií, byla zvolena takto.

V balíčku *backend* jsou obsaženy veškeré entity (`@Entity`), které používá Hibernate a služby (`@Service`, které obstarávají přístup k databázi a poskytují rozhraní možných operací s entitami). Dále pak obsahuje konvertory převádějící DTO[26] na entity a zpět. V podstatě všechny objekty (kromě samotných entit z balíčku *backend*) ztotožňující se s nějakou entitou jsou DTO objekty. Tento návrh byl zvolen pro větší rozvrstvení zodpovědností a zároveň může být jeden objekt odlehčenou verzí toho druhého. Na druhou stranu se platí za to,

[26]Data Transfer Object (DTO) je objekt, který slouží jako mezivrstva mezi databázovými entitami a zobrazovacími prvky.

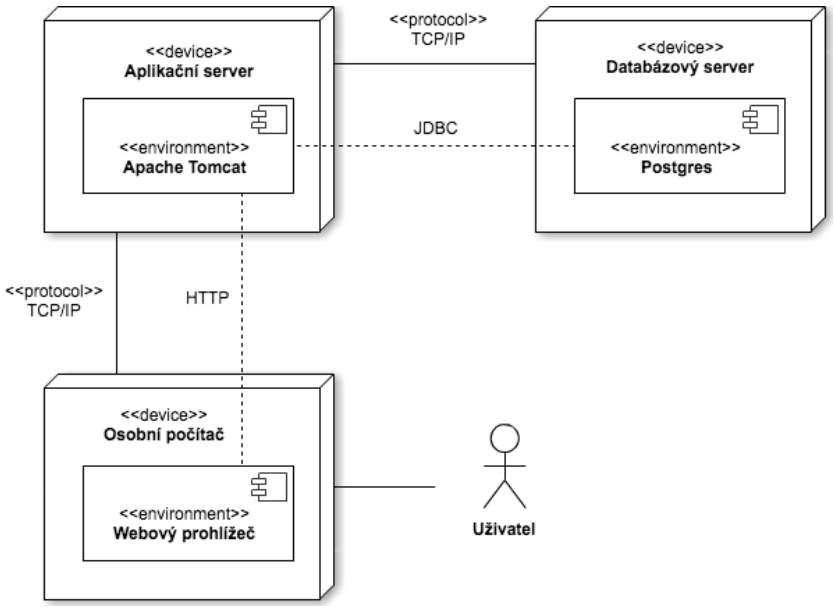
Název balíčku	Popis
backend	obsahuje veškeré třídy (např. entity) spojené s databází
core	v něm jsou definovány všechny použité struktury (graf, CRUD matice, ...)
data	obsahuje podporu pro práci s CSV soubory a pomocné třídy JSON formátu
mvc	veškeré třídy spojené se Spring MVC
utils	pomocné metody

Tabulka 4.1: Nejdůležitější balíčky v aplikaci

že se musí objekty konvertovat mezi sebou. Nicméně k tomuto kroku pomohla statická metoda `copyProperties(source, target)` z třídy `BeanUtils` (součástí `spring-beans` komponenty Spring frameworku), která dokáže zkopírovat (pomocí reflexe) jeden objekt na druhý. Tento přístup je vhodný pro základní typy proměnných, pro složitější struktury (typicky grafové, obsahující cykly) je nutné provést konverzi manuálně. V balíčku *core* jsou implementované struktury grafu, CRUD matice, definice projektu, procesu, chyb a umělé aplikace. Zároveň obsahuje třídy, ve kterých jsou implementovány algoritmy popsané v kapitole 3. Balíček *data* obsahuje pouze pomocné třídy, které slouží k převodu objektu (např. graf) do CSV formátu (konkrétně jsou to třídy implementující interface `Iterator`) a do formátu JSON (pro přenos dat mezi javascriptovou knihovnou a serverovou částí). K převodu objektu do JSON formátu je použita knihovna Jackson (konkrétně modul `jackson-annotations` obsahující anotace), která pomocí anotací určí, jaké proměnné objektu se nemají převádět (anotace `@JsonIgnore`) a také lze díky nim vytvářet i umělé nové vlastnosti JSON objektu (např. anotace `@JsonProperty("nazev_vlastnosti")` nad metodou vracející novou vlastnost). Předposlední balíček *mvc* obsahuje všechny controllery, interceptory a objekty použité ve formulářích v prezenční vrstvě. Poslední balíček obsahuje pomocné třídy, ve kterých jsou pouze statické metody obstarávající opakující operace (např. vyhledání prvku v seznamu podle jeho id).

4.5 Diagram nasazení

Následující schéma na obrázku 4.5 popisuje diagram nasazení, tedy propojení jednotlivých uzlů (hardware) a komponent (software) s nimi související.



Obrázek 4.5: Diagram nasazení

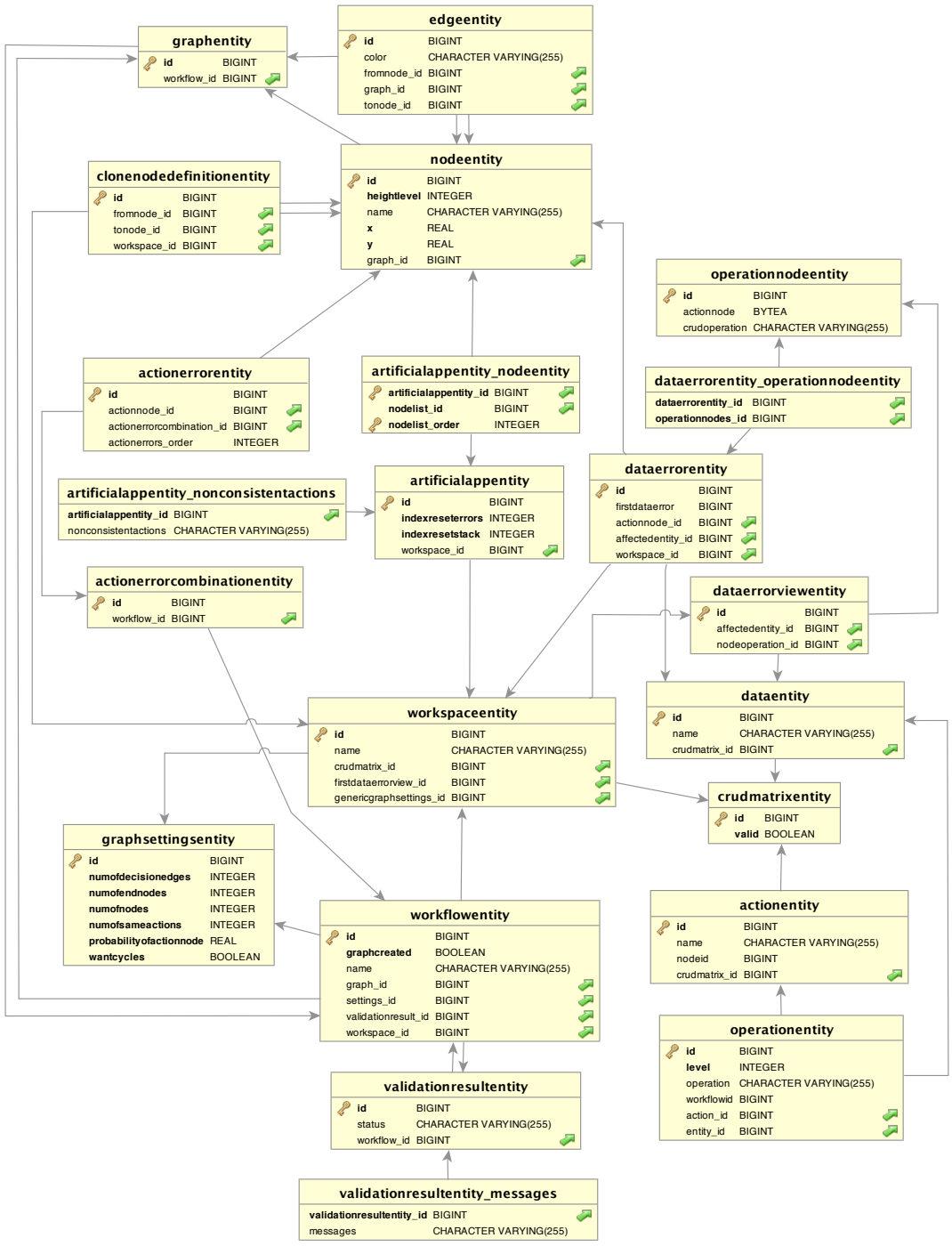
4.6 Databáze

Pro ukládání perzistentních dat byla zvolena databáze Postgres 9.2. Systém k manipulaci s databází používá framework Hibernate, který umožňuje ORM[27] mapování tříd (anotace `@Entity`) na databázové tabulky. Oprošťuje tedy vývojáře od znalosti SQL (nicméně se hodí například při debugování) a zjednodušuje práci (umí vytvořit databázi z definovaných entit).

4.6.1 Databázové schéma

Na obrázku 4.6 je databázové schéma. Tabulka může obsahovat primární klíč (obrázek klíče + tučný text) a cizí klíč (označen zelenou šipkou). Schéma bylo vygenerováno programem DbVisualiser, který obsahuje i editor databázových schémat. Názvy tabulek vycházejí z názvů Hibernate entit a názvy sloupců z názvů instančních proměnných těchto tříd, jejichž typy jsou dány mapováním typů proměnných na typy používané v Postgres.

[27] Object Relational Mapping (ORM) je technika umožňující mapování objektů na relační databázi.



Obrázek 4.6: Databázové schéma

4.7 Infrastruktura použitá při vývoji projektu

Při vývoji byla využita platforma Openshift, aby se vyzkoušelo reálné nasazení. Openshift poskytuje celou řadu (ať už placených či neplacených) funkcí pro správu různých aplikačních serverů (např. Tomcat, JBoss Application Server či WildFly Application Server), serverů pro spouštění testů (Jenkins) a v neposlední řadě poskytuje i databázové servery, na kterých běží např. MySQL, MongoDB či PostgreSQL použité v této práci. Nejjednodušší forma nasazení na Openshift vyžaduje, aby projekt využíval Maven, což je sestrojovací nástroj pro aplikace založené na Javě. Mimo jiné umožňuje jednoduchou správu závislostí v projektu a lze s ním pohodlně deployovat (nahrávat) aplikaci na server. Zároveň také definuje obecně používanou adresářovou strukturu projektu. Pro svou konfiguraci potřebuje pouze soubor `pom.xml`, který leží přímo v kořenovém adresáři aplikace. Pro nahrání na server pak stačí přidat profil (listing 4.2) s id `openshift`, který definuje v `build` fázi `plugin`, vytvářející výsledný war[28] s konfigurací, která specifikuje, jaký název má být použit pro výsledný war. Zde je použit ROOT, což umožní přistupovat k aplikaci na url (popsané v sekci 4.3.1) bez kontextu aplikace.

Pokud máme takto nakonfigurovaný projekt, nejsnažším způsobem, jak dostat aplikaci na Openshift server, je pomocí nástroje Git[29]. Tímto nástrojem byla od počátku verzovaná jak samotná aplikace, tak text této práce. Takže nebyl problém aplikaci na server nahrát.

Listing 4.2: Ukázka Maven profilu pro Openshift

```
<profile>
    <id>openshift</id>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.4</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                    <outputDirectory>webapps</outputDirectory>
                    <warName>ROOT</warName>
                </configuration>
            </plugin>
        </plugins>
    </build>
</profile>
```

[28] Web application ARchive (WAR) je archiv, který obsahuje webovou aplikaci.

[29] Nástroj umožňující sofistikované verzování libovolného textu (včetně zdrojových textů aplikace).

Kapitola 5

Testování

V této kapitole uvedu dva typy testů, které byly provedeny k ověření funkčnosti aplikace. Prvními jsou jednotkové testy, které ověřují funkčnost na nejmenší úrovni kódu. Druhými testy, které vykonávají testeři, jsou funkční testy. Ty by měly odhalit chyby ve funkcionalitách (vycházejících z funkčních požadavků) testované aplikace.

5.1 Jednotkové testy

Pro otestování základních funkcí aplikace byly zvoleny jednotkové testy, které mají za úkol otestovat jednotlivé části kódu aplikace. Testy by měly být srozumitelné a jednoduché, aby se v nich dalo snadno orientovat. Dobře napsaný test umožní neznalému rychlé pochopení toho, jak celá aplikace funguje (testy se mohou velice jednoduše převést na quickstarty[30] v další fázi vývoje). K samotným testům byl zvolen framework JUnit 4. JUnit umožňuje psát testy velmi jednoduše a pouze za pomocí jedné anotace `@Test` nad metodou spolu s využitím některé ze statických metod třídy `Assert` z balíčku `org.junit` k ověření správnosti vykonaného kódu. Ukázka takového testu:

Listing 5.1: Ukázka JUnit testu

```
/**
 * Tests for verifying the acyclic graph generation.
 */
public class AcyclicGraphTest {

 @Test
 public void testCreateAcyclicGraph() {
  Graph cyclic = ... // create graph with some cycles
```

[30] Ukázkové příklady užití aplikace.

```
    // check if has cycles
    Assert.assertTrue(containsCycles(cyclic));
    Graph acyclic = ... // create acyclic graph of the cyclicGraph
    // check if has no cycles
    Assert.assertFalse(containsCycles(acyclic));
    // check if the number of nodes is same
    Assert.assertSame(cyclic.getNodesSize(), acyclic.getNodesSize());
  }

}
```

5.1.1 Spojení JUnit se Spring MVC

Abychom byli schopni testovat i např. poslání požadavku na server, využijeme Spring obsahující komponenty pro testování, které jsou úzce spojeny s JUnit frameworkem. Třída musí mít alespoň dvě anotace, kterými jsou `@RunWith(SpringJUnit4ClassRunner.class)` (umožní použít *dependency injection*) a `@ContextConfiguration` (specifikuje, kde jsou uloženy konfigurační soubory Springu). Spolu s Mockito frameworkem tak vytváří ucelený nástroj testování takových případů užití. Bez Mockita by musela běžet reálná databáze, což je žádoucí např. pokud potřebujeme testovat databázové operace. Nicméně my chceme otestovat správné naplnění Spring Modelu po zaslání konkrétního požadavku. Mockito umožňuje dva základní konstrukty, jak pracovat s *mock* objekty. *Mock* objekt vznikne z objektu (nebo z třídy) typu *b* např. voláním statické metody `mock` z třídy Mockito. Takovému objektu můžeme nastavit, aby po zavolání jeho metody vrátil námi specifikovaný výsledek (anglický termín je *stubbing*). Na ukázce 5.2 se vytvoří mock `MainService` a specifikuje se `Workspace`, který by po zavolání metody `getWorkspaceById(1)` měla vrátit.

Listing 5.2: Ukázka Mocku

```
public void setup() {
  MainService mock = Mockito.mock(MainService.class);
  Workspace workspace = //specify which workspace should be returned
  Mockito.when(mock.getWorkspaceById(1).thenReturn(workspace);
}
```

Pro otestování Spring Controlleru (který využívá nějaký `Service` pro komunikaci s databází) je ho potřeba přidat do třídy s testy jako instanční proměnnou a oanotovat ji `@InjectMocks`, říkající, že by se do něj měly vložit mocky, které jsou také instančními proměnnými s anotací `@Mock`. Pro správné fungování je potřeba volat v metodě s anotací `@Before` (součást JUnit cyklu vykonání testu, která zaručí že bude metoda zavolána před každým testem) `MockitoAnnotations.initMocks(this)`, která vytvoří potřebné mocky

a vloží je do controlleru. Nakonec už stačí vytvořit pouze objekt typu `MockMvc`, který dokáže vyvolávat požadavky nad controllerem. Následující kód ukazuje nejjednodušší možný test, který pouze ověří, zda proběhne vyvolání požadavku na URL `/workspace` v pořádku.

Listing 5.3: Ukázka Spring MVC a Mock

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
//spring configuration files paths
})
public class SimpleSpringMVCTest {
 @InjectMocks
 private WorkspaceController controller;

 @Mock
 private MainService mockService;

 private MockMvc mockMvc;

 @Before
 public void setup() {
  MockitoAnnotations.initMocks(this);
  mockMvc = MockMvcBuilders.standaloneSetup(controller).build();
 }

 @Test
 public void testRecieveAllWorkspaces() throws Exception {
  mockMvc.perform(get("/workspace")).andExpect(status().isOk())
 }
}
```

5.1.2 Pokrytí aplikace pomocí jednotkových testů

Aplikace obsahuje 21 JUnit testů, kde 7 z nich testuje zpracování požadavků v controllerech (kontrola získání správných atributů v modelu vykonáním operací nad `Service`) a zbylých 14 testů se věnuje hlavně operacím s grafy, jelikož to je základ celé aplikace. Mezi ně patří detekce cyklů, tvorba acyklického faktoru grafu, vytváření jednoduchých cest v grafu (s čímž souvisí generování chyb) a v neposlední řadě generování CRUD matice.

Pro otestování algoritmů pracujících s grafy byly vytvořeny tři grafy, z nichž jeden obsahuje cykly, druhý a třetí jsou bez cyklů a liší se pouze v počtu akcí (v jednom případě je dostatečný počet akcí pro vytvoření CRUD matice). Pro otestování konzistence sledu CRUD operací se zvolí graf s větším počtem

akcí a vytvoří se CRUD matice. Ta se pak ve smyčce projde přes všechny entity a ověří se, zda operace splňují sekvenci CREATE, alespoň jeden z READ - UPDATE a na konec DELETE.

5.2 Funkční testy

Pro otestování základních funkcionalit (vycházející z funkčních požadavků) aplikace byly navrhnuty testovací scénáře (vybrané scénáře jsou v příloze B), které se předaly testerům. Scénáře obsahují krátký popis, prioritu, prerekvizity a jednotlivé kroky s očekávaným výstupem systému. Kroky jsou popsány dostatečně detailně, aby mohly být testy vykonávány samostatně.

5.2.1 Průběh testování

Systém byl ve třech případech spuštěn lokálně a předán testerům. V posledním případě byl spuštěn na Openshiftu. Testery se stali moji dva kamarádi a dva rodinní příslušníci.

5.2.2 Odhalené chyby

Nalezené chyby byly nejprve zmapované a následně opravené. Pro snadné odhalení chyb byla vytvořena chybová stránka (která se zobrazila, pokud byla v aplikaci vyhozena jakákoliv výjimka), do které se vepsal do komentáře celý výpis chybového zásobníku aplikace. Pokud došlo k vyhození výjimky v aplikaci, mohl jsem jednoduše zjistit, kde konkrétně chyba nastala. Již první test odhalil chybu. Všechny nalezené chyby:

- Pokud byla vytvořená umělá aplikace a přegeneroval se graf ze stejného projektu, opakovaně se vypsala chybová hláška v seznamu umělých aplikací.

- Při mazání procesu, se kterým byla spojena nějaká umělá aplikace, došlo k chybě v průběhu ukládání do databáze.

Další prohřešky byly vůči nevhodně zvoleným pozicím (a podbarvení) grafických elementů.

- Tlačítko „Vstoupit do aplikace“ se nacházelo na spodní části stránky a mělo bílou barvu (což ubíralo na důležitosti tlačítka). Navíc nebylo jasné, co je myšleno slovem „aplikace“.

- Základní nastavení parametrů pro generování procesního grafu zabíralo příliš mnoho místa na stránce s procesem.

Tlačítko „Vstoupit do aplikace“ bylo přesunuto do horní části stránky a bylo podbarveno zelenou barvou. Zároveň se změnil jeho text na „Vstoupit do umělé aplikace“, aby se význam více přiblížil k akci, kterou tlačítko provádí. Základní nastavení parametrů pro generování procesního grafu je ve výchozím stavu skryto a zobrazí se až po stisknutí tlačítka, kterým lze zároveň formulář opět skrýt.

Závěr

Podle zadání bylo úkolem navrhnout a implementovat systém, který umožní vytvořit umělou webovou aplikaci na základě předem vygenerovaného modelu procesů. Zároveň umožňuje zanesení umělých chyb a jejich detekci při průchodu umělou aplikací. Těchto požadavků bylo dosaženo a k nim bylo přidáno ještě několik dalších rozšíření jako například ruční modelování procesu či export do CSV. Při vývoji byl kladen důraz na využití co největšího počtu frameworků, které by ulehčily práci a já, jako vývojář, se staral pouze o programování zaměřené na jádro aplikace. Díky tomuto přístupu jsem byl oproštěn od složité manipulace s databází skrz SQL dotazy, velmi jednoduše jsem byl schopný naprogramovat zpracování požadavků od uživatele pomocí Spring MVC a díky knihovně vis.js jsem rozšířil aplikaci o možnost editace daných procesů.

Navržené algoritmy jsem se z počátku snažil vymýšlet sám, nicméně poté jsem zjistil, že vymýšlím to, co jiní už dávno prozkoumali (např. tvorba acyklického grafu, či počítání výšky uzlů). Proto jsem se začal soustředit spíše na teorii a hledal řešení ve skriptech prof. Koláře [9] a pak hlavně v „bibli“ algoritmů *Introduction to Algorithms* [12], ve které byly některé části popsané ještě více do detailu.

Přínosem je pro mě poznání nových frameworků, schopnost orientovat se v cizím JavaDocu a využití teorie grafů v praxi. Zároveň lituji toho, že jsem při vývoji přehlížel framework *Thymeleaf*[31], který by mi usnadnil tvorbu uživatelského rozhraní (především vzhled), protože nevyžaduje při každé změně v HTML či CSS přehrání aplikace na serveru. I když mi to znepříjemnilo návrh vzhledu a trvalo mi to více času, než bych očekával, jsem rád, že jsem nezavrhl technologii JSP, protože je to základ Java EE a je dobré tyto znalosti mít.

Aplikace obsahuje strukturu projektů s procesy, což umožňuje tvorbu nezávislých projektů. Nicméně do budoucna by bylo vhodné připsat k aplikaci přihlašovací rozhraní a rozdělit uživatele do rolí. To by utvořilo další vrstvu

[31] Thymeleaf umožňuje psaní dynamických stránek pomocí klasických HTML značek, takže je prohlížeč dokáže bez problému zobrazit.

a projekty by mohly být vázány na konkrétního uživatele. Zároveň by bylo vhodné k exportu dat přidat i import z jiných nástrojů (např. EA[32]) sloužících k tvorbě grafů, aktivity diagramů a podobných vizualizací procesů.

[32]Enterprise Architect (EA)

Literatura

[1] Petitit, F. a Tricot, M.: The new Web application architectures and their impacts for enterprises. 2014. Dostupné z: http://blog.octo.com/wp-content/uploads/2014/03/web-application-models-over-time.png

[2] Sun Microsystems: Distributed Multitiered Applications. 2010. Dostupné z: http://docs.oracle.com/javaee/5/tutorial/doc/figures/overview-multitieredApplications.gif

[3] Pivotal Software: Web MVC framework. 2016. Dostupné z: http://docs.spring.io/autorepo/docs/spring/4.1.6.RELEASE/spring-framework-reference/html/mvc.html

[4] Koomen, T. a kol.: TMap Next. 2010. Dostupné z: http://www.tmap.net/en/tmap-next

[5] Burnstein, I.: *Practical Software Testing: A Process-Oriented Approach*. Springer Science & Business Media, 2006, ISBN 978-0-387-21658-4.

[6] ISO/IEC 24765-2010, Systems and Software Engineering—Vocabulary. 2010. Dostupné z: http://www.iso.org/iso/catalogue_detail.htm?csnumber=50518

[7] TESTINGCUP: Testing Cup - Polish championship in software testing. Dostupné z: http://testingcup.com

[8] Finkelstein, C.: *An Introduction to Information Engineering: From Strategic Planning to Information Systems*. Addison-Wesley, 1990, ISBN 978-0201416541.

[9] Kolář, J.: *Teoretická informatika*. Praha: Česká informatická společnost, 2004, ISBN 80-900853-8-5.

[10] Bartík, V.: Grafy. 2008. Dostupné z: ftp://math.feld.cvut.cz/pub/bartik/Mek/Grafy.pdf

[11] Přispěvatelé Wikipedie: Souvislý graf. 2016. Dostupné z: https://cs.wikipedia.org/wiki/Souvislý_graf

[12] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to Algorithms, 3rd Edition*. Praha: The MIT Press, 2009, ISBN 978-0262033848.

[13] Lewis, W.: *PDCA/Test*. Auerbach Publications, 1998, ISBN 978-0201416541.

[14] Bureš, M.: *Techniky pro testování datových cyklů*. Praha: Zatím nepublikovaná práce.

[15] Přispěvatelé Wikipedie: Java EE version history. 2016. Dostupné z: https://en.wikipedia.org/wiki/Java_EE_version_history

[16] Přispěvatelé Wikipedie: Spring Framework. 2016. Dostupné z: https://cs.wikipedia.org/wiki/Spring_Framework

[17] Přispěvatelé Wikipedie: Vkládání závislostí. 2016. Dostupné z: https://cs.wikipedia.org/wiki/Vkládání_závislostí

[18] Eclipse Foundation: Configuring Contexts. 2016. Dostupné z: http://www.eclipse.org/jetty/documentation/current/configuring-contexts.html

Příloha A

Seznam použitých zkratek

PCT Process Cycle Test

UML Unified Modeling Language

DFS Depth First Search

BFS Breadth First Search

HTTP Hypertext Transfer Protocol

HTML HyperText Markup Language

AJAX Asynchronous JavaScript and XML

REST Representational State Transfer

JSP JavaServer Pages

JSF JavaServer Faces

DRY Don't Repeat Yourself

DTO Data Transfer Object

ORM Object Relational Mapping

WAR Web application ARchive

EA Enterprise Architect

Příloha B

Testovací scénáře

B.1 Vytvoření projektu

ID testu	TEST_1	
Název testu	Vytvoření projektu	
Hloubka detailu	Malá	
Shrnutí testu	Vytvoření projektu, pozitivní průchod	
Popis testu	Uživatel vytvoří nový projekt	
Vstupní podmínky		
Testovací data	Název projektu: Projekt1	
Priorita	Střední	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere stránku projektů	Systém zobrazí seznam projektů včetně formuláře pro vytvoření projektu
2	Uživatel vyplní název projektu a klikne na tlačítko „Vytvořit projekt“	Systém zobrazí seznam projektů včetně nově přidaného projektu

Tabulka B.1: Testovací scénář: Vytvoření projektu

B.2 Vytvoření procesu

ID testu	TEST_2	
Název testu	Vytvoření procesu	
Hloubka detailu	Malá	
Shrnutí testu	Vytvoření procesu, pozitivní průchod	
Popis testu	Uživatel vytvoří nový proces v projektu	
Vstupní podmínky	Vytvořený projekt	
Testovací data	Název procesu: Proces1	
Priorita	Střední	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere stránku projektů	Systém zobrazí seznam projektů včetně formuláře pro vytvoření projektu
2	Uživatel vybere vytvořený projekt	Systém zobrazí informace o vybraném projektu
3	Uživatel vyplní název procesu a klikne na tlačítko „Vytvoř proces“	Systém zobrazí seznam procesů včetně nově vytvořeného procesu

Tabulka B.2: Testovací scénář: Vytvoření procesu

B.3 Aplikování obecného nastavení pro generování procesu

ID testu	TEST_3	
Název testu	Aplikování obecného nastavení pro generování procesu	
Hloubka detailu	Malá	
Shrnutí testu	Aplikování obecného nastavení, pozitivní průchod	
Popis testu	Uživatel změní obecné nastavení procesu aplikováním na vybraný proces	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený prázdný proces	
Testovací data	Max. počet uzlů: 12 Max. počet hran vycházejicích z rozhodovacího uzlu: 3 Pravděpodobnost akce oproti rozhodovacímu uzlu: 0.5 Max. počet konců: 2 Max. společných akcí [%]: 0 Cykly: nezaškrtlé	
Priorita	Malá	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu
2	Uživatel stiskne tlačítko „Zobrazit/Skrýt hlavní nastavení“	Systém zobrazí formulářová pole vztahující se k hlavnímu nastavení
3	Uživatel vyplní pole testovacími daty a stiskne tlačítko „Aplikuj na proces“	Systém zobrazí stejné hodnoty v nastavení konkrétního procesu

Tabulka B.3: Testovací scénář: Aplikování obecného nastavení pro generování procesu

B.4 Vygenerování modelu procesu

ID testu	TEST_4	
Název testu	Vygenerování modelu procesu	
Hloubka detailu	Střední	
Shrnutí testu	Vytvoření modelu procesu, pozitivní průchod	
Popis testu	Uživatel vygeneruje model procesu obsahující cykly	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený prázdný proces	
Testovací data	Max. počet uzlů: 9 Max. počet hran vycházejicích z rozhodovacího uzlu: 2 Pravděpodobnost akce oproti rozhodovacímu uzlu: 0.7 Max. počet konců: 2 Max. společných akcí [%]: 0 Cykly: zaškrtlé	
Priorita	Vysoká	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces.	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu.
2	Uživatel vyplní pole (vztahující se k nastavení parametrů výsledného modelu procesu) testovacími daty a stiskne tlačítko „Generovat graf“.	Systém zobrazí model procesu splňující požadavky.

Tabulka B.4: Testovací scénář: Vygenerování modelu procesu

B.5 Validace procesu

ID testu	TEST_5	
Název testu	Validace procesu, pozitivní průchod	
Hloubka detailu	Střední	
Shrnutí testu	Validace procesu, pozitivní průchod	
Popis testu	Uživatel zvaliduje vytvořený proces	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený proces	
Testovací data	Vygenerovaný model s těmito parametry: Max. počet uzlů: 9 Max. počet hran vycházejicích z rozhodovacímu uzlu: 2 Pravděpodobnost akce oproti rozhodovacího uzlu: 0.7 Max. počet konců: 2 Max. společných akcí [%]: 0 Cykly: zaškrtlé	
Priorita	Střední	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces.	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu.
2	Uživatel vyplní pole (vztahující se k nastavení parametrů výsledného modelu procesu) testovacími daty a stiskne tlačítko „Generovat graf“.	Systém zobrazí model procesu splňující požadavky.
3	Uživatel stiskne tlačítko „Zvalidovat“.	Systém zvaliduje model procesu a zobrazí, že je graf konzistentní.

Tabulka B.5: Testovací scénář: Validace procesu

B.6 Vytvoření CRUD matice

ID testu	TEST_6	
Název testu	Vytvoření CRUD matice	
Hloubka detailu	Vysoká	
Shrnutí testu	Vytvoření CRUD matice, pozitivní průchod	
Popis testu	Uživatel vytvoří CRUD matici	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený proces	
Testovací data	Vygenerovaný model s těmito parametry: Max. počet uzlů: 9 Max. počet hran vycházejicích z rozhodovacímu uzlu: 2 Pravděpodobnost akce oproti rozhodovacího uzlu: 1 Max. počet konců: 2 Max. společných akcí [%]: 0 Cykly: nezaškrtlé	
Priorita	Vysoká	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces.	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu.
2	Uživatel vyplní pole (vztahující se k nastavení parametrů výsledného modelu procesu) testovacími daty a stiskne tlačítko „Generovat graf“.	Systém zobrazí model procesu splňující požadavky.
3	Uživatel stiskne tlačítko „Vytvoř CRUD matici“.	Systém zobrazí vygenerovanou CRUD matici.

Tabulka B.6: Testovací scénář: Vytvoření CRUD matice

B.7 Vstup do umělé aplikace bez validní CRUD matice

ID testu	TEST_7	
Název testu	Vstup do umělé aplikace bez validní CRUD matice	
Hloubka detailu	Malá	
Shrnutí testu	Vstup do umělé aplikace bez validní CRUD matice, negativní průchod	
Popis testu	Uživatel vytvoří proces bez CRUD matice a pak se pokusí vstoupit do umělé aplikace. Systém zobrazí chybovou hlášku, že není CRUD matice v pořádku.	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený model procesu bez CRUD matice (nebo s CRUD maticí, která není validní).	
Testovací data		
Priorita	Vysoká	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces.	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu.
2	Uživatel stiskne tlačítko „Vstup do umělé aplikace“	Systém nepovolí vstup do umělé aplikace a vypíše hlášku, že je nutné vytvořit novou validní CRUD matici.
3	Uživatel vybere projekt.	Systém zobrazí seznam umělých aplikací včetně nově vytvořené.

Tabulka B.7: Testovací scénář: Vstup do umělé aplikace bez validní CRUD matice

B.8 Vstup do umělé aplikace s validní CRUD maticí

ID testu	TEST_8	
Název testu	Vstup do umělé aplikace s validní CRUD maticí	
Hloubka detailu	Malá	
Shrnutí testu	Vstup do umělé aplikace s validní CRUD maticí, pozitivní průchod	
Popis testu	Uživatel vytvoří proces s validní CRUD maticí a pak vstoupí do umělé aplikace.	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený model procesu s validní CRUD maticí. Validní CRUD matici lze vytvořit z procesu s těmito parametry: Vygenerovaný model s těmito parametry: Max. počet uzlů: 9 Max. počet hran vycházejicích z rozhodovacímu uzlu: 2 Pravděpodobnost akce oproti rozhodovacího uzlu: 1 Max. počet konců: 2 Max. společných akcí [%]: 0 Cykly: nezaškrtlé	
Testovací data		
Priorita	Vysoká	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces s validní CRUD maticí.	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu.
2	Uživatel stiskne tlačítko „Vstup do umělé aplikace“	Systém zobrazí počátek procesu (toho, z kterého bylo tlačítko zmáčknuto) a jeho další možné akce.

Tabulka B.8: Testovací scénář: Vstup do umělé aplikace s validní CRUD maticí

B.9 Detekce přidané chyby v umělé aplikaci, pozitivní průchod

ID testu	TEST_9	
Název testu	Detekce přidané chyby v umělé aplikaci	
Hloubka detailu	Vysoká	
Shrnutí testu	Detekce přidané chyby v umělé aplikaci, pozitivní průchod	
Popis testu	Uživatel vytvoří proces s validní CRUD maticí, přidá umělé chyby v kombinaci akcí a pak vstoupí do umělé aplikace. Zkontroluje, zda umělá aplikace zobrazí chyby v definovaných akcích.	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený model procesu (s alespoň jednou akcí) s validní CRUD maticí.	
Testovací data		
Priorita	Vysoká	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces s validní CRUD maticí.	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu.
2	Uživatel přidá umělou chybu k akci a_1 pomocí manuální konfigurace.	Systém zobrazí všechny přidané chyby v akcích.
3	Uživatel stiskne tlačítko „Vstup do umělé aplikace“	Systém zobrazí počátek procesu (toho, z kterého bylo tlačítko zmáčknuto) a jeho další možné akce.
4	Uživatel prokliká pomocí tlačítek s názvy akcí umělou aplikaci tak, aby stiskl tlačítko akce a_1.	Systém zobrazí zvýrazněnou hlášku s informací, že v umělé aplikaci nastala chyba.

Tabulka B.9: Testovací scénář: Detekce přidané chyby v umělé aplikaci, pozitivní průchod

B.10 Úprava modelu umělé aplikace znemožní pokračovat dále v umělé aplikaci

ID testu	TEST_10	
Název testu	Úprava modelu umělé aplikace znemožní pokračovat dále v umělé aplikaci	
Hloubka detailu	Střední	
Shrnutí testu	Úprava modelu umělé aplikace znemožní pokračovat dále v umělé aplikaci, negativní průchod	
Popis testu	Uživatel vytvoří umělou aplikaci a následně změní strukturu modelu aplikace (např. přegeneruje proces, nebo ho smaže).	
Vstupní podmínky	Vytvořený projekt a v něm vytvořený model procesu (s alespoň jednou akcí) s validní CRUD maticí.	
Testovací data		
Priorita	Vysoká	
ID kroku	**Kroky testu**	**Očekávané výsledky**
1	Uživatel vybere projekt a v něm proces s validní CRUD maticí.	Systém zobrazí stránku obsahující editační nástroje vztahující se k procesu.
2	Uživatel stiskne tlačítko „Vstup do umělé aplikace“	Systém zobrazí počátek procesu (toho, z kterého bylo tlačítko zmáčknuto) a jeho další možné akce.
3	Uživatel se vrátí do seznamu procesů pomocí tlačítka „Vrátit se zpět do projektu“.	Systém zobrazí stránku s procesy a vytvořenými umělými aplikacemi.
4	Uživatel vybere libovolný proces a změní model umělé aplikace (přegeneruje graf nebo vygeneruje CRUD matici).	Systém zobrazí změny provedené uživatelem.
5	Uživatel přejde zpět na stránku projektu.	Systém znemožní výběr umělé aplikace a zobrazí u ní důvod této blokace.

Tabulka B.10: Testovací scénář: Úprava modelu umělé aplikace znemožní pokračovat dále v umělé aplikaci

Příloha C

Instrukce pro instalaci

C.1 Softwarové požadavky

Pro spuštění aplikace je potřeba mít nainstalovanou Javu SE s verzí alespoň 7. S nižší verzí aplikace nebude fungovat, neboť používá funkce (např. diamond operátory), které nejsou podporovány v nižších verzích. Dále je potřeba mít nainstalovaný Maven (vývoj využíval verzi 3.3.3). Pro funkční propojení aplikace s databází, je potřeba mít nainstalovanou PostgreSQL databázi (byla použita verze 9.5.0.0). Jako poslední požadavek je Apache Tomcat s verzí 7.

C.2 Hardwarové požadavky

Běh aplikace bude bezproblémový na jakémkoli novějším počítači, který obsahuje prohlížeč.

C.3 Instalační postup

Pro instalační postup předpokládáme, že je na cílovém stroji nainstalován potřebný software ze sekce C.1. Uvedu krok po kroku instalační postup aplikace. Pro popis využiji názvů kořenových adresářů, kam byl nainstalován software. Pro Apache Tomcat budu používat `TOMCAT_DIR`, pro Maven (typicky složka v domovském adresáři `.m2`) `MAVEN_DIR`, pro rozbalenou aplikaci `APP_DIR` a pro Postgres `POSTGRES_DIR`. Nejprve je nutné přidat uživatele (včetně jeho hesla) s rolemi do Apache Tomcat. Tyto údaje bude využívat Maven pro nahrávání aplikace.

1. editovat soubor `TOMCAT_DIR/conf/tomcat-users.xml`
2. přidat mezi tagy `<tomcat-users>` toto:

```
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
```

```
<user username="admin" password="password" roles="
    manager-gui,manager-script"/>
```

3. uložit soubor

Navíc ještě musíme nastavit proměnnou `dbLocal`, která říká, jaký konfigurační soubor databáze se má vybrat.

- editovat soubor `TOMCAT_APP/conf/catalina.properties`
- na konec souboru přidat `dbEnv=local`
- uložit soubor

Dále musíme přidat přihlašovací údaje i do nastavení Mavenu (abychom ho pak mohli jednoduše použít pro nahrání aplikací).

1. editovat soubor `MAVEN_DIR/settings.xml`
2. upravit ho tak, aby obsahoval záznam s tagem `<server>`
3. ukázka celého `settings.xml` (bez namespace v kořenovém elementu):

```
<settings ...>
     <servers>
          <server>
               <id>TomcatServer</id>
               <username>admin</username>
               <password>password</password>
          </server>
    </servers>
</settings>
```

Pro spojení s databází předpokládám, že Postgres běží na localhost s portem 5432. Přístupové údaje jméno a heslo `postgres`. Pokud se údaje v něčem liší, je potřeba pozměnit konfigurační soubor v aplikaci. Konkrétně v souboru `APP_DIR/src/main/resources/database-local.properties` (je možné vytvořit i nový konfigurační soubor, který by pojmenován `database-novyNazev` a v `TOMCAT_APP/conf/catalina.properties` se změní proměnná `dbLocal=novyNazev`). Pro vytvoření databázového schématu je potřeba vykonat tyto kroky:

1. spustit `psql` příkazovou řádku (např. `POSTGRES_DIR/bin/psql -p 5432`)
2. vytvořit novou databázi `DROP DATABASE hibernatedb; CREATE DATABASE hibernatedb;`

3. připojit se do nově vytvořené databáze `\c hibernatedb ;`
4. nahrát skript s tabulkami `\i APP_DIR/scripts/database/tables_script.sql`

Nyní už máme vše potřebné a stačí spustit server a následně aplikaci.

1. `cd TOMCAT_DIR/bin`
2. nastavení spustitelných skriptů `chmod +x *.sh`
3. spuštění serveru `./catalina.sh start`
4. zobrazení logu `tail -f ../logs/catalina.out`

Pro ověření, že byl server spuštěn správně stačí zkontrolovat zda je na konci logu `INFO: Server startup in xxx ms`, kde `xxx` je počet milisekund. Ještě uvedu, jak server vypnout. Ze stejné složky jako jsme spustili server, můžeme server vypnout pomocí `./shutdown.sh`.

1. vstoupit do složky `APP_DIR`
2. pro nahrání aplikace na server:
   `mvn clean tomcat7:deploy -Dmaven.test.skip=true`
3. pro spuštění testů:
   `mvn -DdbEnv=local -Dsurefire.useFile=false clean test`

Aplikace je následně dostupná na url:
`http://localhost:8080/ArtificialAppGenerator/app`

Příloha D

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `LICENSE-APACHE-2.0.txt` ..... licence Apache
- `LICENSE-MIT.txt` ..... licence MIT
- `manual.pdf` ..... uživatelská příručka
- `site` ..... dokumentace zdrojového kódu v HTML
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF