Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Šablony fragmentů Java EE aplikací

Plzeň, 2016 Bc. Ondřej Šimice

PROHLÁŠENÍ

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 20. 6. 2016

Bc. Ondřej Šimice

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat svému vedoucímu diplomové práce Ing. Petrovi Přibylovi ze společnosti CCA a Ing. Janě Varnuškové, Ph.D. z FAV ZČU za cenné rady a přívětivé vedení po celou dobu diplomové práce.

ABSTRACT

The templates of the fragments of the Java EE applications

The programmer's time is very valuable commodity during development of any application. This thesis deals with generation of the source code of Java EE applications from formalized design with using templates. The goal of the thesis wasn't to create whole application, but only fragments of the source code, which has sense to be generated. This effort aims to save programmers time. In this thesis you will became familiar with the CCA company methodology for module design of the user interface. This will be followed by design of input for the generator as XML file and description of the reference implementation for generator, which by using the templates generates fragments of the source code for modules of user interface. Text of this thesis emphasises options for potential modifiability and further expansion of the generator.

ABSTRAKT

Šablony fragmentů Java EE aplikací

Programátorův čas je velmi drahá komodita při vývoji jakékoliv aplikace. Tato práce se zabývá generováním zdrojového kódu Java EE aplikací z formalizovaného návrhu za využití šablon. Práce neměla za cíl vytvořit celou aplikaci, ale jen fragmenty zdrojového kódu, které má smysl generovat. Toto úsilí si klade za cíl ušetřit programátorův čas. V práci se nejdříve seznámíte s metodikou firmy CCA pro návrh modulů uživatelského rozhraní. Následovat bude návrh vstupu pro generátor ve formě XML souboru a popis referenční implementace generátoru, který s využitím šablon generuje fragmenty zdrojových kódů modulů uživatelského rozhraní. Text diplomové práce klade důraz hlavně na možnosti modifikovatelnosti a další rozšíření generátoru.

Obsah

1. ÚVOD 1

2. Úvod do problematiky generování zdrojového kódu Java EE aplikací z formalizovaného návrhu 2

2.1. Účel 2

2.2. Zásady pro generování 3

3. Metodika CCA pro návrh modulů uživatelského rozhraní 6

3.1. Moduly uživatelského rozhraní ve firmě CCA 6

3.2. Primefaces 7

3.2.1. Java Server Faces 7

3.2.2. PrimeFaces 7

3.3. Stránky modulu 9

3.3.1. Template.xhtml 10

3.3.2. Stránka Edit 11

3.3.3. Stránka Detail 12

3.3.4. Stránka List 12

3.4. Návrh aplikace v EA 14

3.4.1. Návrh uživatelského rozhraní aplikace 14

3.4.2. Návrh stránky v EA dle metodiky CCA 17

3.4.3. Závěr k metodice pro návrh v EA: 21

4. Dostupné nástroje pro generování kódu Java EE aplikací 22

4.1. Enterprise Architect 22

4.1.1. Nástroj export-data-model 23

4.2. Forge 2 24

4.2.1. Generování webové aplikace ve Forge 24

5. Návrh struktury konfigurace generátoru 26

5.1. Proces generování 26

5.1.1. XSLT transformace 27

5.2. Struktura dat vstupního XML generátoru 27

6. Referenční implementace generátoru na vybrané technologii 29

6.1. Implementace generátoru 29

6.1.1. Zpracování stránek ze vstupního XML 37

6.1.2. Datové objekty pro uložení komponent stránek 39
6.1.3. Parsery komponent stránek..42
6.2. Popis šablon 45
6.2.1. Fremarker[12] 46
6.2.2. Návrh šablon v pluginu 48
6.2.3. TemplateProcesor..50
6.3. Rozšíření pluginu 50
6.3.1. Obecná pravidla pro přidání komponenty 50
6.3.2. Přidání TabView 51
6.3.3. Přidání komponenty commandLink 55
7. Ověření funkčnosti generátoru na příkladu...58
8. ZÁVĚR 61
LITERATURA..62
PŘÍLOHA A – UŽIVATELSKÁ PŘÍRUČKA 63
PŘÍLOHA B – Export z EA, XSL soubor a výstup XSTL transformace 65
PŘÍLOHA C – Příklad vstupního XML generátoru 68

1. ÚVOD

Čas je jediná konstanta v našem životě. Den má 24 hodin, hodina 60 minut, a tak dále. Mnohdy se říká, že čas jsou peníze. Toto rčení platí i v softwarovém inženýrství. Programátorův čas stojí peníze. Pokud chcete urychlit vývoj softwaru, můžete na práci nasadit více programátorů, ale bude také stát více peněz. Navíc jen za předpokladu, že máte ve firmě dostatek programátorů. Z těchto důvodů je vhodné jejich časem zbytečně neplýtvat.

Při psaní programového kódu aplikací, a to zejména uživatelského rozhraní, jsou programátoři často nuceni psát spoustu podobných a opakujících se programových konstrukcí. Ve většině rozumně fungujících společností, které se zabývají vývojem softwaru, probíhá navíc nejdříve fáze návrhu aplikace, která bývá zpravidla nějakým způsobem zaznamenána buď ve formě textové specifikace, nebo v lepším případě v nějakém softwaru, ze kterého je pak možné velkou část specifikace generovat. Tyto informace musí někdo, zpravidla konzultant, sesbírat od zákazníka a někam zaznamenat a pak přijde na řadu softwarový architekt, který navrhne software a návrh opět musí někam zaznamenat, což se děje převážně elektronicky s využitím nějakého specializovaného softwaru. Po této fázi přijde na řadu programátor, který tyto informace musí promítnout při implementaci softwarového produktu. Také tu vidíte problém?

Ano, informace již minimálně jednou někde zaznamenané, musí být znovu zaznamenány programátorem. Na tento problém se snaží reagovat generování programového kódu.

Ve druhé kapitole Vás se seznámíme s problematikou generování Java EE aplikací z formalizovaného návrhu. Třetí kapitola popisuje metodiku CCA pro návrh modulů uživatelského rozhraní. Čtvrtá kapitola je věnována dostupným nástrojům pro generování kódu Java EE aplikací. V páté kapitole je představen návrh struktury konfigurace generátoru. Šestá kapitola se věnuje referenční implementaci generátoru pro generování fragmentů zdrojových kódů Java EE aplikací podle pravidel používaných ve firmě CCA z XML předpisu navrženém v kapitole pět. Kromě toho je zde uvedeno, jaké kroky je třeba provést pro přidání nové generované komponenty a demonstrace tohoto návodu na konkrétních případech. A sedmá kapitola pojednává o použití a ověření funkčnosti generátoru na konkrétním příkladu.

2. Úvod do problematiky generování zdrojového kódu Java EE aplikací z formalizovaného návrhu

2.1. Účel

Firma CCA se zabývá vývojem aplikací většinou pro větší subjekty, jako je např. státní správa či soudy. Ty pro své fungování potřebují ukládat velké množství dat o zaměstnancích, zákaznících, procesech, funkcích a podobně. Data jsou uložena obvykle v databázovém systému Oracle. Aplikace firmy CCA mají za úkol zajistit snadné ukládání, úpravu a vizualizaci těchto dat. Aplikace, kterými se bude zabývat tato práce, jsou psané v Javě s využitím PrimeFaces, což je nadstavba nad JavaServer Faces. Obojí je Java Enterprise Edition řešení pro vývoj uživatelského rozhraní webových aplikací. PrimeFaces se zabývá kapitola 3.2.

Pro dosažení cílů z předchozího odstavce je vývoj aplikací ve firmě CCA rozdělen do několika fází. Nejdříve probíhá důkladná komunikace se zákazníkem, aby bylo zjištěno, s jakými daty a procesy se bude pracovat. V další fázi systémový architekt vytvoří v systému Enterprise Architekt návrh aplikace. Pod návrhem aplikace je možné si představit: jaké webové stránky bude aplikace obsahovat, jaké jsou na stránce prvky (textová pole, tlačítka,…) a s jakými daty v databázi bude webová stránka propojena.

Když je hotový návrh, přichází na řadu programátor, který má za úkol jej implementovat ve vybrané technologii. V tomto případě Java a PrimeFaces.

Na základě mnohaletých zkušeností dospěli pracovníci firmy CCA ke zjištění, že při vývoji jejich aplikací a to zejména při vývoji uživatelského rozhraní se značná míra informací v různých fázích návrhu opakuje, případně se jen nepatrně liší. Toto zjištění vedlo k myšlence tento proces nějak zautomatizovat. Jednou z možností může být generování programového kódu. V současné době návrhář připraví návrh stránky včetně polí a jejich popisu a tuto informaci programátor využije. Nyní to přepisuje. Společnost CCA chce, aby to bylo generováno. Generovat celý kód se ukázalo příliš drahé. Je to několikanásobně dražší než, ruční přepis. Následovala tedy další myšlenka, negenerovat vše, ale generovat jen to, co má smysl. To znamená jen ty části, které půjdou popsat dostatečně obecně a jednoduše, aby jejich generování nestálo v celkovém měřítku větší úsilí než jejich napsání.

Problematikou generování kódu se zabývá kniha Code Generation in action [1], některé poznatky z ní jsou zmíněny v kapitole 2.2. Existují nástroje, které generování kódu do jisté míry podporují, ty budou rozebrány v kapitole 4.

2.2. Zásady pro generování

V softwarovém inženýrství existují dvě základní konstanty, se kterými se musíme potýkat [1]:

- Programátorův čas je cenný.
- Programátoři nemají rádi opakující se nudné úkoly.

Generování programového kódu, ať už částečné či úplné (na většině projektů je příliš drahé) do značné míry eliminuje psaní opakujícího se kódu a šetří tak čas programátorů. Kromě úbytku značné míry těžké a nezajímavé práce přináší do životního cyklu softwarového inženýrství několik výhod [1]:

- Produktivitu – generátory programového kódu, mohou vytvořit stovky tříd během několika sekund. To je produktivita, které se s ručním psaním nemůže dosáhnout. Záleží samozřejmě i na typu úlohy, kterou řešíme, a na kvalitě cílového řešení. Pokud chceme později takový kód upravit, změny můžou být promítnuty napříč systémem vcelku rychle.
- Kvalita – generovaný kód má jednotnou kvalitu napříč generovaným celkem. Pokud nalezneme bug (chybu), můžeme jej jednoduše opravit přegenerováním kódu.
- Konzistence – API (aplikační programové rozhraní), postavená užitím generování kódu jsou konzistentní v zápisu a názvech proměnných. To usnadňuje ruční úpravu kódu v budoucnosti, protože je přehlednější.
- Abstrakce – některé generátory dostávají na vstupu abstraktní model cílového systému. Tento model reprezentuje high-level business pravidla cílového systému. Takto jsou pravidla viditelná pro analytiky i programátory na rozdíl od pravidel schovaných v ručně psaném kódu. Navíc abstraktní model zvyšuje přenositelnost systému, protože šablony generátoru můžeme použít pro jiné programovací jazyky, technologie a frameworky.

Výhody generování kódu pro programátory [1]:

- Kvalita – Vysoká míra ručně psaného kódu vede k nekonzistentní kvalitě, jelikož programátoři během své práci nacházejí novější a kvalitnější přístupy. Využitím šablon při generování má tu výhodu, že se stále vytváří konzistentní kód. Při změně šablon a přegenerování kódu se vylepšení promítnout napříč celou bází programového kódu.
- Konzistence – Kód je tvořen, tak aby byl generátor kódu stavěn konzistentně v návrhu API a pojmenování proměnných. Díky tomu jsou rozhraní dobře pochopitelná a snadno použitelná.

- Jeden bod poznání – pokud uděláme změnu kódu ve vstupním souboru pro generátor, promítne se při přegenerování v celém systému, nebo alespoň v těch částech, které generátor ovlivňuje. Naopak i v sebelépe ručně psaném kódu si například změna názvu tabulky vyžádá spoustu manuálních změn.
- Více času na návrh – čas, který ušetříme tím, že za nás generátor vytvoří část kódu, můžeme využít na vytvoření lepšího návrhu.

Deset zásad generování kódu[1]:

1. Respektovat ruční psaní kódu – měli bychom přistupovat s respektem k ručně psanému kódu, protože často existují speciální případy integrované do programového kódu, které jsou snadno přehlédnutelné. Když nahrazujeme ručně psaný kód, potřebujeme se ujistit, že jsme nepřehlédli nějakou speciální situaci a máme pokryté všechny případy užití.
2. Nejdříve porozumět ručně psanému kódu – než začneme psát jakýkoliv generátor, je nutné dostatečně porozumět tomu, jak funguje nebo jak by fungoval, ručně psaný kód bez generování. Ideálně bychom měli nejprve ručně napsat dostatečné množství různých příkladů v dané technologii a pak tento kód využít, jako základ pro budoucí šablony generátoru. Druhá možnost je, že už to někdo napsal za nás a máme k ruce zdrojové kódy a někoho s kým se dají zkonzultovat.
3. Kontrola zdrojového kódu – ne dobré používat systémy pro správu verzí (např. SVN, GIT)
4. Dobře zvážit implementační jazyk – pro implementaci generátoru se může hodit jiný programovací jazyk, než bude cílový jazyk aplikace. Důvod je pro to ten, že generátor bude mnohdy řešit jiný druh úlohy, než cílová aplikace (pokud nebude cílová aplikace opět generátor). Proto je důležitá volba vhodného programovacího jazyka a knihovny, které nám mohou usnadnit hodně práce.
5. Integrování generátoru do vývojového procesu – jelikož bude generátor používán programátory při jejich práci, je vhodné, aby byl navržen tak, aby zapadal do standardního procesu vytváření kódu ve firmě. V některých případech může být vhodné, integrovat generátor přímo do vývojového prostředí (např. plugin do Eclipse). Zde je ale třeba být rozvážný, protože se tím zavádí silná závislost na verzi prostředí.
6. Varovné komentáře ke generovanému kódu – generovaný kód je vhodné obalit komentáři a řádně zdokumentovat, co je generováno, aby se předešlo tomu, že programátor upraví generovaný kód, spustí znovu generátor a ztratí své úpravy.
7. Důraz na použitelnost – snadné ovládání generátoru a výpis chybových výstupů může programátorovi značně usnadnit práci.
8. Dokumentovat – dobrá dokumentace by měla být součástí generátoru. Měla by zahrnovat hlavně, co generátor dělá, jak se instaluje, jak se spouští, jaké soubory ovlivňuje a také jak se dá rozšířit.

9. Nezapomínat na uživatele – je důležité budoucí uživatele zaškolit, jak generátor funguje, ukázat jeho výhody i slabá místa a správné způsoby použití. Lidé jsou skeptičtí k novým věcem a takto je možné jim nový nástroj lépe přiblížit a bude větší šance, že bude používán tak, jak bylo zamýšleno. Případně se takto dají odhalit slabá místa, která by mohly být ještě vylepšena, aby používání generátoru lépe korespondovalo s vývojovým procesem ve firmě.
10. Udržovat generátor – používáním generátoru budou vznikat požadavky na menší či větší úpravy, které by měly být pravidelně implementovány, aby se generátor zlepšoval a byl tak lépe využitelný ve vývojovém procesu.

3. Metodika CCA pro návrh modulů uživatelského rozhraní

3.1. Moduly uživatelského rozhraní ve firmě CCA

V první řadě je třeba si říci, co je myšleno slovem modul a co spojením uživatelské rozhraní. Ve spojení s uživatelským rozhraním je také třeba definovat, co bude rozuměno pod pojmem aplikace.

Aplikací bude v textu této diplomové práce myšlen program, který bude schopen provádět akce CRUD (create, read, update, delete) nad nějakými daty v databázi. Vše bude demonstrováno na aplikaci Ramses Akademie, kterou vytvořila firma CCA pro řízení svých vnitřních procesů. Tato aplikace uchovává informace o zaměstnancích a jejich úkolech.

Uživatelským rozhraním je myšleno grafické uživatelské rozhraní aplikace, které bylo u aplikace Ramses Akademie implementováno jako Java Enterprise webová aplikace ve frameworku PrimeFaces (více v kapitole 3. 2.). Uživatelské rozhraní je tedy v tomto kontextu sada webových stránek obsahující grafické vstupní prvky, jako jsou menu, formuláře, tlačítka a podobně. Uživatel tyto uživatelské prvky ovládá například pomocí myši a klávesnice.

Pod slovem modul bychom si měli v našem kontextu představit jednu část uživatelského rozhraní. Moduly uživatelského rozhraní aplikací CCA jsou specifické v tom, že jsou zpravidla spojeny s nějakou databázovou entitou, která má sadu parametrů. Pro názornou ukázku jsem si zvolil databázovou entitu Funkce. Funkce je zde chápána jako pozice pracovníka ve firmě. Může jí být například Vedoucí střediska, Personalista, Dělník, apod. Každá funkce má sadu parametrů – kód, název, platnost od, platnost do, popis, znalosti, odpovědnosti a přepínač, je-li funkce povolena k použití. Příklad funkce může být Inženýr hardware a sítí, kód funkce je unikátní identifikátor funkce. Popis obsahuje, co tato funkce obnáší. Znalosti uvádějí, jaké jsou požadavky na vykonávání této funkce a odpovědnosti, co má člověk v této funkci na starost.

Každý modul se skládá ze tří stránek detail, edit a list, které slouží k vytváření, vizualizaci a úpravě dat modulu (například modulu funkce). Interakci mezi stránkami modulu znázorňuje obrázek 2 (viz kapitola 3.3).

3.2. Primefaces

3.2.1. **Java Server Faces**

Java Server Faces (JSF) je Java specifikace pro vytváření komponentově-orientovaných uživatelských rozhraní pro webové aplikace. Byly formalizovány jako standard skrz Java Comunity Process jako součást Java Platform Enterprise Edition. JSF 2, používá Facelety jako svůj hlavní šablonovací systém.

Facelet

Facelet je webový framework pro tvorbu prezenční vrstvy ve webových aplikacích postavených na JSF[9]. Facelety jsou založeny na standardu XML, což je velká výhoda pro programátory, protože pro základní tvorbu JSF stránek se tak stačí naučit pár nových XML tagů. Obsah Facelets souborů je běžný xhtml kód mezi nímž, jsou tagy různých knihoven. Tagy knihoven se rozlišují podle jmenných prostorů (viz tabulka 1).

Knihovna	Prefix	Popis
JSF Facelets	ui:	Tagy pro tvorbu šablon
JSF HTMLTag Library	h:	Tagy komponent uživatelského rozhraní
JSF Core Tag Library	f:	Tagy s funkcionalitou nezávislou na RenderKitu
JSTL Core Tag Library	c:	Tagy jádra JSTL – cykly, podmínky atd.
JSTL Functions Tag Library	fn:	Tagy JSTL pro funkce – např. toUpperCase atd.

Tabulka 1 Typy jmenných prostorů knihoven

Velice důležitá je knihovna z namepsace ui, která dovoluje tvořit šablony stránek. Příkladem je template.xhtml, do které se pak z konkrétní stránky umisťují jednotlivé části jako hlavička, tělo, apod.

3.2.2. **PrimeFaces**

PrimeFaces jsou lightweight knihovna v jednom jar souboru, nulovou konfigurací a žádnými velejším závislostmi.

Většina dnešních webových aplikací klade důraz na interakci s uživatelem. JSF sice nabízí velké množství komponent, se kterými se snadno pracuje, ale jde o komponenty, které generují čisté HTML. Interakce se pak obvykle dosahuje kombinací CSS stylů a javascriptové knihovny JQuery.Výhodou PrimeFaces[11] je, že jsou postaveny na JQuery a obsahují velké množství propracovaných komponent, které pokrývají téměř všechny požadavky na vývoj moderních webových aplikací. Využitím komponent PrimeFaces tak programátorovi odpadá spoustu práce. Příklad takové komponenty je například *tabView Panel,* který reprezentuje panel s horizontálními záložkami, které mohou obsahovat další komponenty (viz obr 1).

Vše co je potřeba je přidání do *pom.xml* Mavenovského projektu závislost na Primefaces verze 4:

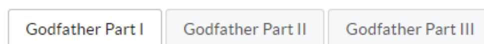
```xml
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>4.0</version>
</dependency>
```

Pro používání primefaces komponent v *xhtml* stránkách stačí do záhlaví přidat toto namespace:

```
xmlns:p="http://primefaces.org/ui"
```

Poté už je možné vyžívat libovolné komponenty PrimeFaces.

Basic



The story begins as Don Vito Corleone, the head of a New York Mafia family, oversees his daughter's wedding. His beloved son Michael has just come home from the war, but does not intend to become part of his father's business. Through Michael's life the nature of the family business becomes clear. The business of the family is just like the head of the family, kind and benevolent to those who give respect, but given to ruthless violence whenever anything stands against the good of the family.

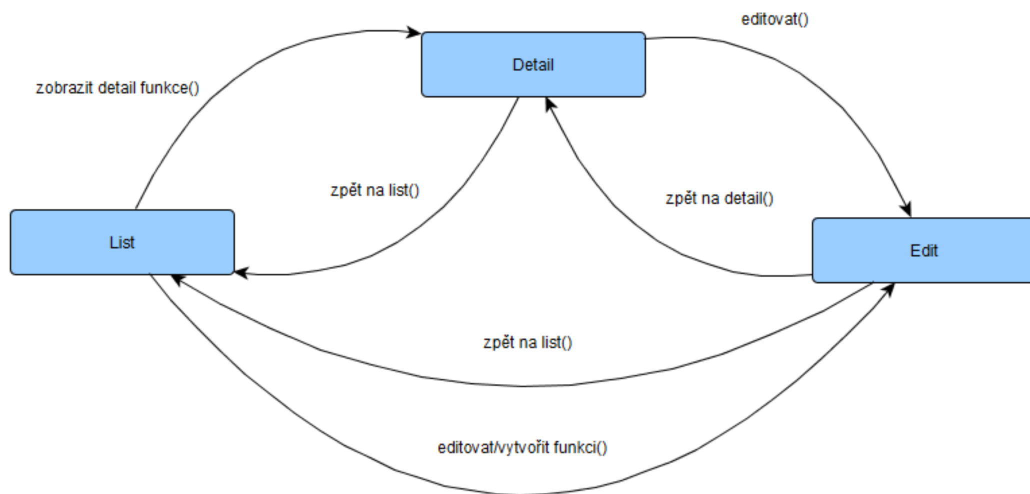
Obrázek 1 TabView komponenta PrimeFaces

3.3. Stránky modulu

Proto, abych byl schopen správně generovat požadované soubory, bylo nejprve nutné se seznámit s tím, jaké soubory to jsou, kde se nachází, jaký je jejich účel, z jakých částí se jednotlivé soubory skládají a opět jaký význam tyto jednotlivé stavební bloky. V následujících kapitolách tedy podrobně popíši strukturu stránek modulů.

Jak bylo zmíněno již dříve, jedná se o xhtml stránky napsané s využitím Primefaces. A mým úkolem bylo se s využitím generátoru, až na několik zjednodušení, dostat ke stejnému kódu, jako byl ručně napsán programátorem. Kromě modulů stránek jsem generoval ke každé stránce také Java Action Bean, které slouží jako controllery obsahující funkční logiku spojenou se stránkami.

Každý modul se skládá ze tří stránek. *Edit*, *Detail* a *List*. Na příkladu Funkce jsou tyto moduly reprezentovány soubory *funkceEdit.xhtml*, *funkceDetail.xhtml*, *funkceList.xhtml*. V dalším textu budou tyto stránky pro jednoduchost označovány pouze jako *Edit*, *Detail* a *List*. Tyto webové stránky mají společnou šablonu *template.xhtml*, ve které je hlavní struktura stránky. Obrázek 2 zachycuje interakci mezi stránkami.



Obrázek 2 Interakce mezi stránkami modulu

3.3.1. **Template.xhtml**

Soubor *template.xhtml* slouží jako šablona pro všechny stránky modulu (*Detail, Edit a List*). V tom to souboru je definována základní struktura stránky, která se skládá z hlavičky těla. Uvnitř hlavičky je umístěn *title* stránky, v těle bloky typu <*script*>, které obsahují funkce napsané v javascriptu, používané v aplikaci. Body se skládá ze čtyř <*div*> bloků: *header, menu, body* a *footer*.

Blok *header* představuje záhlaví stránky a je do něj vkládán blok *pageHeader* příslušné stránky *detail, edit či list.*

Blok *menu*, je společné menu pro každou stránku. Obsahuje dvě tlačítka (odkaz na domovskou stránku a odkaz na seznam komponent) a pod nimi obrázek.

Blok body reprezentuje tělo stránky a je do něj vkládán blok *body* příslušné stránky *detail, edit či list*.

Na obrázku 3 je vidět, jak se zobrazí stránka *Detail* z aplikace Ramses Akademie v prohlížeči.

Ramses Akademie - detail funkce

Ramses Akademie
Správa funkcí

Detail funkce	
Kód	KT31
Název	Inženýr hardware a sítí
Organizace	CCA Group a.s.
Platnost od	01.01.2014
Platnost do	
Popis	- správa hardware serverů a sítí v CCA i u zákazníků (návrh konfigurace, objednávání, instalace, údržba, upgrade) - správa telefonní sítě a telefonní ústředny CCA - údržba pasivních síťových prvků a konfigurace aktivních síťových prvků - správa komutovaného připojení z/do sítě CCA - internetového připojení - průběžný monitoring síťového provozu a zatížení - správa systému vybraných serverů - sledování vývoje v oboru - získávání a analýza informací - vyřizování reklamací - školení uživatelů
Znalosti	- požadované praktické zkušenosti s hardware RISC serverů a síťovými a komunikačními technologiemi - znalost komunikačních standardů a protokolů v oblastech LAN, WAN, Internetu, sériové a modemové komunikace - znalost OS UNIX výhodou - řízení osobního automobilu
Odpovědnosti	- odpovídá za funkčnost a optimální výkon: počítačové a telefonní sítě, připojení do Internetu, vzdáleného přístupu z/do sítě CCA přes modem (RAS), tel. ústředny a telefonů (s výjimkou mobilních telefonů). - Dále zodpovídá za funkčnost a vhodnou konfiguraci hardware RISC serverů a zabezpečení síťové komunikace proti zneužití
Použití povoleno	Ano

Přiřazení dovedností | Editovat | Zpět

© 2014 CCA Group a.s., Powered by Primefaces 4.0

Obrázek 3 Stránka Detail

3.3.2. **Stránka Edit**

Stránka *Edit* má dvě funkce jednou z nich je vytváření nového záznamu a druhá je editace existujícího záznamu. V závislosti na tom v jakém režimu se uživatel na stránku dostane, jsou či nejsou renderované některé texty či tlačítka na stránce.

Stránka *Edit* obsahuje dva základní bloky kompozice stránky: *pageheader* a *body*.

Blok *pageheader* obsahuje jen hlavičku stránky.

Bloku body je blok typu formulář *pageForm*. Uvnitř formuláře je možné nalézt tři panely:

- *homePanel*
- *dataPanel*
- *navPanel*

a navíc blok *confirmDialog*.

Panel *homePanel* obsahuje nadpis úrovně jedna a v něm se renderuje jeden z textových bloků *pageHeaderTextEdit* nebo *pageHeaderTextNew* podle toho, je-li stránka zavolána k editaci existujícího prvku nebo k vytváření nového. O tom, jaký text bude renderován, rozhoduje obslužná funkce *newRecord*, která vyhodnotí právě to, zdali se jedná o nový záznam či o editaci již existujícího.

Renderování

Ve spojitosti s otázkou renderování je nutné zmínit jeden důležitý fakt. Jednotlivé elementy, které se renderují jen na základně nějaké podmínky, jsou vždy obsaženy ve zdrojovém xhtml souboru příslušné stránky na serveru (tzn. ve výstupu mého generátoru). Podmínka, zdali budou renderovány, určuje pouze, jestli budou obsaženy v html souboru, který je poslán klientovi ze serveru, na základě http reqestu na cílovou stránku. Pro jednoduchost vypadá usecase takto: Klient zadá do prohlížeče url stránky Edit (vytvoří http regest na server), na serveru je zpracována stránka *Edit.xhtml*, jsou vyhodnoceny podmínky, co se má referovat a v http response je poslán klientovy soubor edit.html, kde jsou pouze ty části, které se mají renderovat.

Panel *dataPane*l obsahuje panelGrid a dvě tlačítka: uložit a smazat. Uložit slouží pro uložení změn a smazat pro smazání celé entity. Tlačítko pro smazání se renderuje pouze, je-li stránka v režimu edit. V panelGridu jsou pak facet, který obsahuje hlavičku formuláře – podle toho, zdali se vytváří nový prvek nebo se edituje existující prvek, se opět renderuje jeden z textů *tableHeaderTextNew* nebo *tableHeaderTextEdit*. Dále jsou v panelGridu jednotlivé atributy vytvářené či editované entity. Každý atribut má několik částí: *Label*, v němž je název atributu, dále pak panel, jenž obaluje prvek, do nějž je

zadávána hodnota atributu a prostor pro zprávu, kde můžou být zobrazeny výstupy validace vstupu. Prvky pro zadávání můžou být editovatelné textové pole, needitovatelné textové pole, text area, kalendář, či rozbalovací seznam, v němž je možný výběr z několika hodnot (např. ano/ne).

Panel *navPanel* je navigační panel, který standardně obsahuje pouze tlačítko zpět. Toto tlačítko je ve zdrojovém souboru xhtml dvakrát, jedno nás vrátí na stránku Edit druhé na stránku List. Tlačítka mají atribut rendered, který uvádí, jaká verze tlačítka se uživateli zobrazí. V případě, že jsme se na stránku *Edit* dostali ze stránky *Detail*, vrátí nás tlačítko *zpět* na stránku detail. V případě, že se na stránku *Edit* přišlo ze stránky *List,* nebo se vytváří nový záznam, vrátí nás tlačítko na stránku *List*. Interakce mezi stránkami byla zobrazena na obrázku 3 v úvodu kapitoly 3.3.

Dále je na stránce ještě blok confirmDialog, který slouží jako okno se dvěmi tlačítky *confirm* a *decline* pro potvrzování akcí.

3.3.3. **Stránka Detail**

Stránka detail slouží pouze pro zobrazení informací o entitě. Je možné se z ní dostat tlačítkem *editovat* na stránku *Edit* a zde editovat entitu, nebo tlačítkem *zpět* na stránku *List*.

Stránka *Detail* obsahuje opět dva základní bloky *pageheader* a *body*.

Blok *pageheader* má stejný význam jako u stránky *Edit*.

Blok *body* má uvnitř sebe, stejně jako u stránky *Edit*, formulář, který však obsahuje pouze panelGrid *detailPanel* a panel *navPanel*. Navigační panel *navPanel* má v sobě obsažené tlačítka *edit* a *zpět*, jejichž význam byl popsán výše.

Uvnitř panelGridu je opět facet *header* s textem hlavičky formuláře a pod ním už jsou jednotlivé atributy entity. Reprezentace atributu je zde oproti stránce *Edit* trochu jednodušší. Atribut je tvořen dvojicí label a text, kde label je popisek atributu a text hodnota atributu.

3.3.4. **Stránka List**

Stránka *List* slouží k zobrazení seznamu entit. Pro entitu funkce (funkce ve firmě) na ní můžeme vidět seznam všech funkcí ve firmě, které jsou zaneseny v databázi. Tlačítka *edit* na konci každého řádku slouží k přechodu na stránku *Edit* a editaci dané funkce. Na navigačním panelu dole jsou pak ještě tlačítka *Nová funkce, Smazat* a *Export*. Nová funkce opět vyvolá stránku *Edit* v kontextu vytváření nové funkce. Tlačítko Smazat smaže vybrané záznamy a tlačítko *Export* exportuje vybrané záznamy do souboru *xls,* což je formát pro Microsoft Excel. V horní části stránky je umístěn filtr funkcí, který

slouží pro filtrování záznamů, které budou zobrazeny na stránce *List* podle hodnot atributů jejich atributů.

Stránka *List* má opět podobnou strukturu jako měla stránka *Edit* či *Detail*. Přejdeme tedy rovnou k části *body*, která opět obsahuje formulář s id *pageForm*, který obsahuje čtyři bloky. Accordion panel, ve kterém je filtr pro stránku list, pak panel s tabulkou, ve kterém je zobrazen seznam funkcí a nakonec navigační panel a *confirmDialog* (potvrzovací panel). Navigační panel obsahuje tři tlačítka pospaná výše a *confirmDialog* plní podobnou funkci jako u stránky *Edit.* Zajímavý je pouze filtr a panel s tabulkou.

Filtr je tvořený komponentou Primefaces *accordinPanel*. Accordion panel je kontejner, který umožňuje sdružovat několik panelů vertikálně nad sebou to tzv. tabů, které pak můžou obsahovat jiné komponenty. Výhodou accordion panelu jet, že obsah každého tabu lze „srolovat“ a nezabírá tak na stránce tolik místa, když jej nepotřebujeme používat. Filtr je zde tvořen accordion panelem pouze s jedním tabem uvnitř něj jsou komponenty, podle kterých je možné filtrovat a dvě tlačítka: *Zobrazit* sloužící pro aplikaci zvoleného filtru a *Vymazat filtr* pro zrušení filtru a zobrazení všech záznamů bez filtru. Atributy podle nichž lze fitrovat jsou tvořeny labelem, v němž je název filtrovaného parametru a panelem v němž je komponenta pro zadání vstupu filtrovaného atributu a komponenta message pro zobrazení zprávy validace vstupu filtrovaného atributu.

Panel s tabulkou je tvořen panelem uvnitř něj je komponenta *dataTable*. Komponenta *dataTable* má sadu parametrů pro vstup dat, napojení na filtr, nastavení stránkování či nastavení stránkování (kolik záznamů bude vidět na stránce), které není naším cílem detailněji rozebírat, jelikož byly nějak jednotně zvoleny při návrhu a pro účely této práce jsou považovány za konstantní. Uvnitř komponenty *dataTable* je opět facet *header* s textem hlavičky tabulky a pak jednotlivé sloupce tabulky. Každý sloupeček tabulky je tvořen elementem *column*, který má atribut *headerText*, jenž, jak název napovídá, obsahuje text záhlaví tabulky pro tento sloupec. Uvnitř toho elementu je pak komponenta, v níž je obsah sloupce tabulky. Zpravidla je to *commandLink* (proniknutelný text s odkazem na jinou stránku – zde na stránku detail), element s běžným textem či tlačítko (např. tlačítko *edit* pro přechod na stránku *Edit* a editaci vybraného řádku tabulky).

3.4. Návrh aplikace v EA

Firma CCA využívá pro návrh svých aplikací nástroj *Enterprise Architect* (dále budu používat zkratku EA) od firmy *Sparx*. Tento nástroj je do značné míry konfigurovatelný. Umožňuje tvorbu profilů, což je implementace metodik firmy a umožňuje změnit chování směrem k uživatelům. Zadavatel (firma CCA) si vytvořil vlastní metodiku, jak v něm navrhovat uživatelská rozhraní a jeho části. Metodiku zachycuje *metamodel* na obrázku 4. Díky němu Enterprise Architect nabídne uživateli nové atributy, nad rámec standardních, aby je vyplnil.

V této kapitole je detailně popsán metamodel a jeho součásti. Z popisu jaká data jsou v EA uložena, bude těžit kapitola 4, jejíž součástí export dat z EA, který je součástí procesu, který si klade za cíl z dat v EA vygenerovat zdrojových souborů cílové aplikace (viz Obrázek 9 v kapitole 4.1.1).

3.4.1. Návrh uživatelského rozhraní aplikace

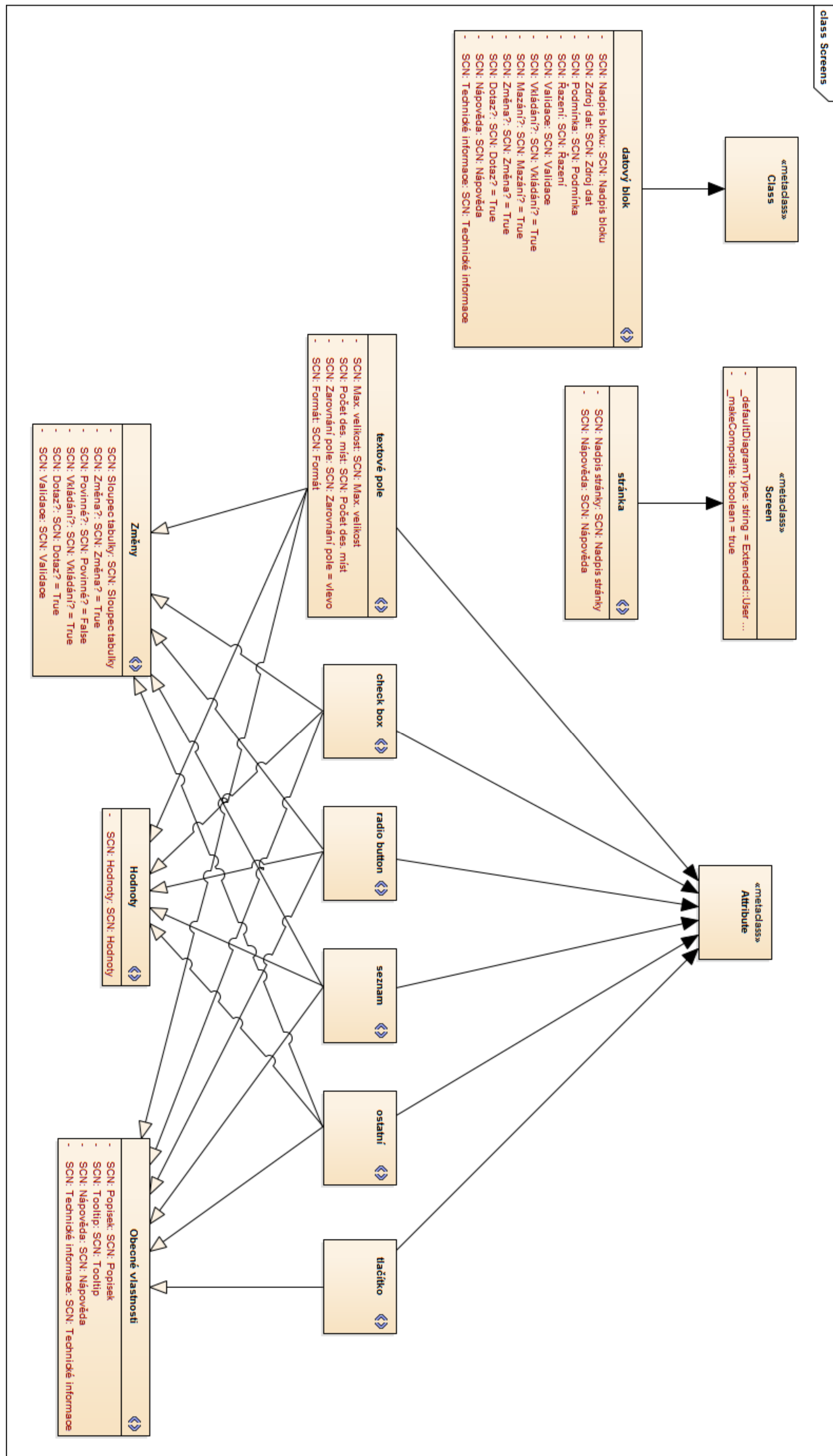
Jednou z věcí, která lze v EA navrhovat je uživatelské rozhraní. (Informace převzaty z [3]). Uživatelské rozhraní vedeno ve složce *Návrh aplikace\Moduly aplikace*. Každá stránka, formulář nebo tisková sestava je uložena v příslušné složce daného subsystému a má ve své struktuře vždy zahrnut diagram s uživatelský rozhraním a diagram s datovými prvky modulu.

Nahlédneme na metodiku CCA pro návrh uživatelského rozhraní nové stránky [3]:

1. V nabídce EA je potřeba vybrat složku *Moduly aplikace*, v ní se nachází diagram menu, kam bude nová stránka patřit. Tento diagram je třeba si zobrazit.
2. V panelu Toolbox v EA musíme zvolit nástroj *CCA: Screens*. Jeho metamodel je na obrázku 4.
3. Vybereme z něj element typu *stránka* a vložíme jej do diagramu. Po této akci se nám automaticky založí podřízený diagram pro uživatelské rozhraní.
4. Po tomto kroku je nutné založit volání na tuto stránku z menu nebo z jiných podřízených stránek. K tomuto účelu využijeme vazbu *navigate*.
5. Dále otevřeme podřízený diagram, který se vytvořil v bodu 3, a z programové záložka EA ProjectBrowser do tohoto diagramu vložíme nově založenou stránku.
6. Do stránky vložíme obrázek se skicou vzhledu uživatelského rozhraní.
7. Popíšeme obecné vlastnosti stránky.
8. Pod stránkou založíme nový diagram typu *Screens – Datový model stránky*
9. Založíme datové bloky a jejich pole.
10. Stránce přiřadíme případy užití, které realizuje.

11. Pro tyto případy užití vyznačíme v popisu příslušných polí, kde se tyto případy užití využívají.

Pokud se formulář v navrhovaném uživatelském rozhraní skládá z více obrazovek, vytvoříme pro něj další objekty typu *stránka* a řídíme se stejným postupem.



Obrázek 4 Metamodel pro EA firmy CCA (poskytnuto zadavatelem)

3.4.2. **Návrh stránky v EA dle metodiky CCA**

V **popisu stránky či sestavy** je třeba nastavit několik vlastností.

Stránka má tři základní skupiny vlastností:

- Vzhled
- Data
- Funkce

Vzhled je popsaný na diagramu vzhledu. Pro návrh skicy stránky se zadavatel používá nástroj *WireFrameSketcher*. Příklad jak vypadá skica, ukazuje obrázek 8. (Pozn.: data by měla být fiktivní, přesto byla pro jistotu anonymizována.)

Data jsou popsány objektem *datový blok*. Datový blok je skupina objektů na obrazovce, které mají stejný datový základ. V 99 % případů odpovídá datový blok jedné tabulce. Každý datový blok má své atributy.

Funkce říká, jaké interakce je možné se stránkou dělat, co se stane při stisknutí tlačítka a podobně.

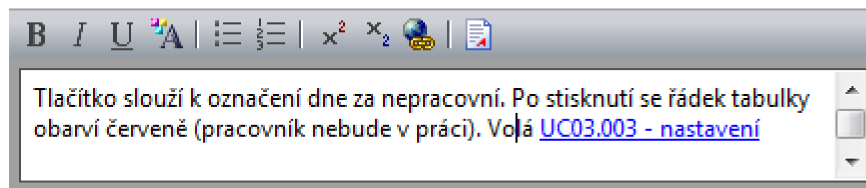
Ke každé stránce je třeba přiřadit sadu vlastností. Metodika CCA definuje vlastnosti uvedené v tabulce 1. Je třeba si všimnout rozlišení standardních vlastností pro stránku definovaných EA. Mezi ně patří Name a Notes. Na rozdíl od toho vlastnosti definované firmou CCA jsou odlišeny prefixem *SCN:*. Tento prefix označuje vlastnosti související s obecnými vlastnostmi pro stránku, či vlastnostmi pro datový blok, který je také součástí definice stránky, jak je možné si všimnout v tabulce 2.

Name	Kód a název stránky. Kód vychází ze standardu číslování stránek
Notes	Popis stránky, obecné informace o účelu stránky
SCN: Nadpis stránky	Nadpis okna se stránkou/formulářem, nadpis tiskové sestavy
SCN: Nápověda	Nápověda ke stránce

Tabulka 2 Vlastnosti stránky v EA (převzato z [3])

.

S každou stránkou jsou také spjaty činnosti, které je s ní možno vykonávat [3]. Činnost se ke stránce přiřadí realizací případů užití. Spolu s tím je třeba v popisu s tím spojených prvků stránky vyznačit, odkud se daný případ užití pouští. K tomuto účelu se využívá *hyperlink*. Na obrázku 5 vidíme, jak vypadá navázání pole na případ užití.



Obrázek 5 Navázání pole na případ užití (převzato z [3])

Obsah stránky je definován datovým blokem, u kterého je nutné vyplnit vlastnosti uvedené v tabulce 3.

Name	Název datového bloku - významový název
Notes	Obecný popis datového bloku, např. *Údaje o osob*
Stereotype	Vždy *datový blok*
SCN: Dotaz?	Je v bloku povolen dotaz? (pro forms aplikace)
SCN: Mazání?	Je v bloku dovoleno mazání záznamů?
SCN: Nadpis bloku	Nadpis bloku dat
SCN: Nápověda	Nápověda ke stránce
SCN: Podmínka	Podmínka pro výběr záznamů v bloku dat
SCN: Řazení	Podle čeho budou záznamy v bloku řazeny
SCN: Validace	Validační podmínka
SCN: Vkládání?	Je v bloku dovoleno vkládání?
SCN: Zdroj dat	Název zdroje dat, typicky název tabulky
SCN: Změna?	Je v bloku dovolená změna dat?

Tabulka 3 Vlastnosti nastavované pro Datový blok (převzato z [3])

Součástí stránky jsou pole, která můžou být různých typů. Polím jsou dále ještě přiřazeny vlastnosti, které pole blíže specifikují. Seznam vlastností, které jsou pro pole definovány, jsou uvedeny v tabulce 4. Metodika CCA definuje pro návrh stránky tyto typy polí (Typy polí na stránce převzaty z [3]):

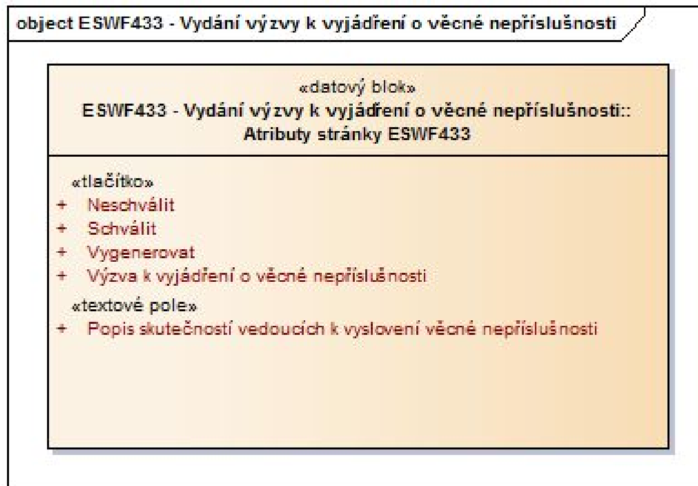
- textové pole (P) – je obecný text, který může být i na více řádek. Používá se i pro číselné hodnoty.
- seznam (S) – je pole typu seznam. Vzhled seznamu je zřejmý z grafického návrhu stránky, je-li to potřeba, může být popsán i v popisu prvku.
- radio button (R) - je výběrové tlačítko z několika možností.
- check box (C) - je zaškrtávací tlačítko.
- tlačítko (T) – představuje standardní tlačítko. Obvykle slouží ke spuštění Případu užití.
- ostatní (O) – sem jsou řazeny ostatní prvky obrazovky jako například obrázky, atd.

Některé vlastnosti uvedené v tabulce 4 jsou aplikovány na všechna pole, některé z nich pouze na nějaké typy polí. U právě uvedeného výčtu polí je za názvem pole velké písmeno, to koresponduje se sloupcem „Aplikuj na“ v tabulce 4. Toto spojení jasně definuje, pro jaké typy polí má být vlastnost vyplněna.

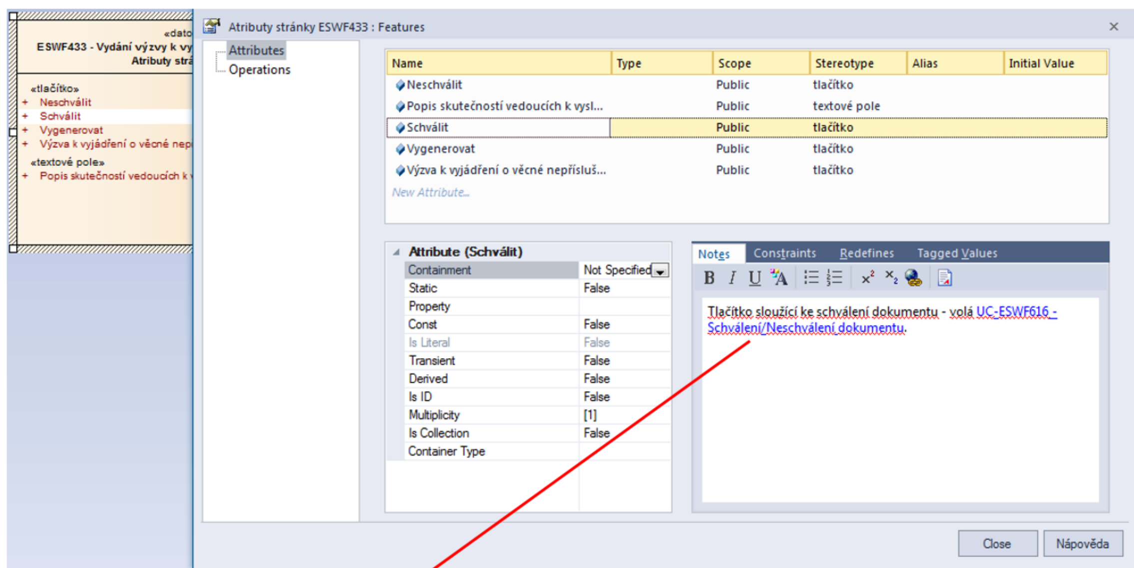
Vlastnost	**Označení**	**Popis**	**Aplikuj na**
Notes	Název pole	Název pole	Všechny
Type	Datový typ	Datový typ pole	P,S,R,C,O
Stereotype	Typ pole	typ ovládacího prvku	Všechny
Notes	Popis	Obecný popis prvku. Obsahuje všechno, co se nevejde do ostatních údajů	Všechny
SCN: Popisek	Popisek	Nadpis pole, text tlačítka	Všechny
SCN: Tooltip	Tooltip / hint	Hint pole, tooltip	Všechny
SCN: Nápověda	Nápověda	Nápověda	Všechny
SCN: Změna?	Operace	Je v poli dovolená změna dat?	P,S,R,C,O
SCN: Povinné	Operace	Povinné pole?	P,S,R,C,O
SCN: Vkládání?	Operace	Je v poli dovoleno vkládání?	P,S,R,C,O
SCN: Dotaz?	Operace	Dotazovatelné pole?	P,S,R,C,O
SCN: Validace	Validace pole	Validační podmínka	P,S,R,C,O
SCN: Hodnoty	Výčet hodnot	Hodnoty seznamu, check boxu, ...	S,R,C,O
SCN: Max. velikost	Velikost	Maximální velikost pole	P
SCN: Počet des. míst	Počet des. míst	Počet desetinných míst	P
SCN: Zarovnání	Zarovnání	Zarovnání pole (vlevo, vpravo, střed)	P
SCN: Formát	Formát	Formát pole	P

Tabulka 4 Vlastnosti polí (převzato z [3])

Podívejme se na obrázek 6, na kterém vidíme objektový diagram datového bloku obsahující dvě pole – textové pole a tlačítko. Abychom mohli zadat popis a vlastnosti datového bloku, musíme nejprve rozkliknout datový blok. Popis se vyplňuje v poli *Notes* (obrázek 7)a v záložce *Screens* se popíší vlastnosti datového bloku.

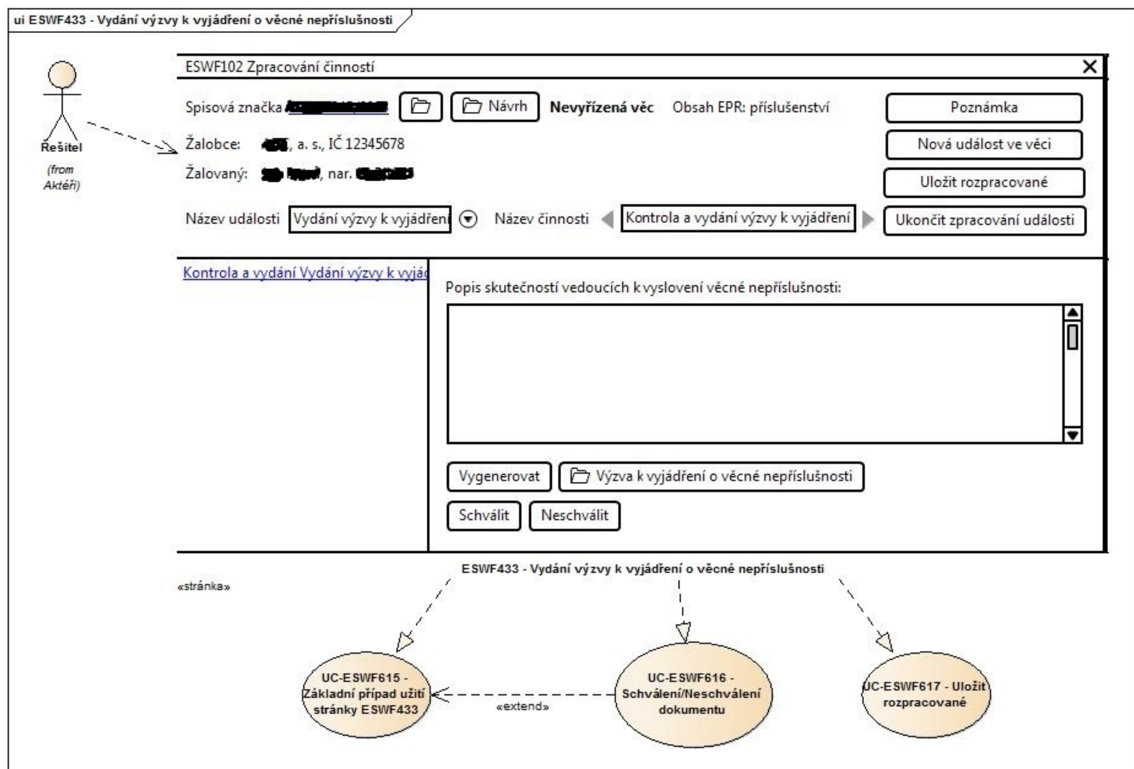


Obrázek 6 Objetkový diagram datového bloku (převzato z [3])



Název	Schválit	Typ		Popisek	Schválit
Popis	Tlačítko sloužící ke schválení dokumentu - volá UC-ESWF616 - Schválení/Neschválení dokumentu.			Nápověda	
Formát	«tlačítko» / - / -			Validace	
Data	- / / **vkládání změna dotaz povinné**				

Obrázek 7 Popis atributu datového bloku (převzato z [3])



Obrázek 8 Návrh vzhledu stránky (převzato z [3])

3.4.3. **Závěr k metodice pro návrh v EA:**

Z takovéhoto návrhu v EA lze vygenerovat dokumentaci nebo pak výstup ve formě xml (viz kapitola 5.1).

4. Dostupné nástroje pro generování kódu Java EE aplikací

Jako objekt zkoumání této kapitoly byly zvoleny dva nástroje. Oba na doporučení zadavatelské firmy, která s nimi má již zkušenosti. Jedním z nich je Enterprise Architect a druhým z nich je Forge ve verzi 2.

Důvodem pro výběr EA, je že se ve firmě CCA používá pro návrh aplikace a jeho zapojení do procesu generování kódu by bylo velice výhodné, protože by se tak mohl propojit proces návrhu a generování.

Nástroj Forge byl vybrán proto, že za ním stojí silná komunita, velice rychle se vyvíjí a společnost JBoss, která zastřešuje vývoj tohoto nástroje, je velký hráč na poli webových technologií. Forge je určený k tomu aby generoval kusy kódu a například z entity je schopen vygenerovat celou aplikaci. Nejenže ji umí vygenerovat, ale umí ji vygenerovat nejlépe podle pravidel pro psaní Java EE aplikací. Výsledný kód pak vypadá jako by jej psal velmi dobrý programátor. Tato možnost byla již ve verzi Forge 1 a jmenuje se *UI Scaffolding*.

4.1. Enterprise Architect

Enterprise Architect rozmanitý nástroj od firmy Sparx, který má mnoho funkcí. Je v něm možné zachytit proces návrhu aplikace ve formě různých diagramů dle normy UML a dalších. Na základě těchto diagramů je možné generovat dokumentaci či přímo zdrojový kód.

Ve firmě CCA se několik let zabývali možnostmi generování kódu přímo z EA. EA umí například z Class diagramu vygenerovat základní strukturu tříd a rozhraní, což jsem si ověřil již při tvorbě bakalářské práce. Generování zdrojového kódu je však omezeno počtem souborů, které je možné generovat. Na jeden průchod lze generovat pouze dva soubory. Proto je pro potřeby generování fragmentů kódu ve firmě CCA nevhodný. V současné době jich potřebují generovat minimálně sedm.

Podívejme se však na EA z jiného pohledu. Nemůže nás sice dovést přímo k tíženému cíli. Ale je možné se k němu přiblížit. Pro popis následujícího textu bude využit diagram na obrázku 9.

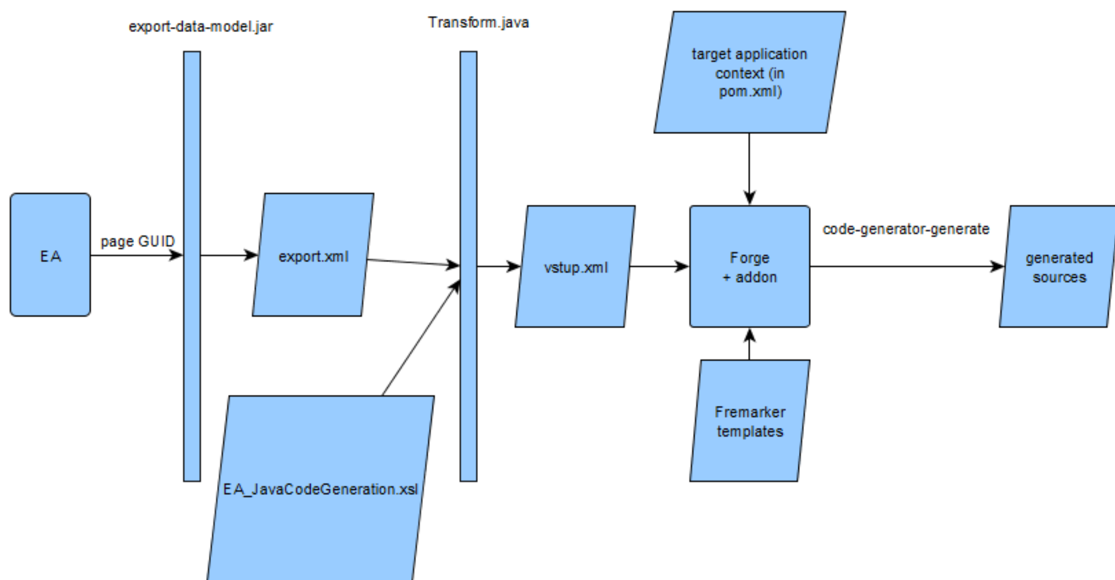
Existuje Java API podporující export dat z EA. Dále pak nástroj *export-data-model* napsaný v Javě ve firmě CCA, který využívá to API.

4.1.1. **Nástroj export-data-model**

Použití nástroje *export-data-model* bylo popsáno v [4]. Zadavatel mi poskytl jeho zdrojový kód i výsledný *export-data-model.jar* soubor, který šel použít spolu s EA API pro export. Jak probíhá export je součástí diagramu na obrázku 9.

Z EA se exportuje celá stránka, která obsahuje datové bloky (jak již bylo popsáno v kapitole 3). Vstupem pro export dat je jedinečný identifikátor stránky (GUID), který může vypadat například takto *A6BD624A-D785-4004-941A-098938D48513*. Výstupem je pak XML soubor. Příklad, jak takový soubor může vypadat, je vidět na obrázku 36 v přípoze B. Na diagramu na obrázku je nazván jako *export.xml*.

Podíváme-li se na strukturu souboru *export.xml* zjistíme, že odpovídá struktuře stránky popsané v kapitole 3. Soubor *export.xml* obsahuje kořenový element stránka. Stránka má jeden *datový blok*, a tak dále. Zajímavé jsou elementy *list*, *detai* a *edit* uvnitř datového bloku. Například element *list* je generován na základě *atributu SCN: Dotaz?*.



Obrázek 9 Process generování

Shrneme-li to, EA umožňuje generování kódu, které není pro potřeby CCA dostatečné. Nicméně poskytuje Java knihovnu s API, které umožňuje napsat si nástroj na export dat například do formátu XML. To jaké data budou generována a v jaké formě lze ovlivnit úpravou zdrojového kódu nástroje *export-data-model*.

4.2. Forge 2

Každý, kdo se nějaký čas zabývá vývojem Java EE aplikací, s velkou pravděpodobností strávil hodně úsilí a energie vytvářením layoutů projektu, nastavováním závislostí nebo správných class path pro jeho kompilaci a spuštění. Přestože Maven umožňuje snížit tuto zátěž v porovnání s udržováním konfigurace projektu ručně, je typicky celkem dost konfiguračního kódu definující závislosti projektu, který je pro běh aplikace nutno napsat do pom.xml.[2]

Podle [2] JBoss Forge "nabízí inkrementální zlepšení projektů Java EE aplikací". Protože je implementován jako příkazový shell, nabízí nám možnost upravovat projektové soubory a složky. Mezi typické případy užití Forge patří:

- přidání Java Persisetnce API (JPA) entit a popis jejich modelu,
- konfigurace Maven závislostí,
- deploy aplikace na server.

Celkově je cílem Forge usnadnit nastavení projektu ve všech jeho fázích vývoje.

Velikou výhodou Forge je, že je do něj možné vytvářet vlastní pluginy, které tak můžou využívat výhod jádra Forge, které stále přibývají, protože za Forge stojí aktivní komunita, která ho neustále vyvíjí.

Nejzajímavější součástí součástí Forge je UI Scaffolding, sloužící pro generování kódu, jehož jádrem se inspirovala i původní vezte pluginu, na jehož rozvoji jsem se podílel.

4.2.1. Generování webové aplikace ve Forge

Ve Forge lze vygenerovat aplikaci použitím UI Scaffoldingu[6] z entity během minuty. Na obrázku 10 jsou všechny příkazy, které musí být postupně zavolány a na obrázku 11 je pak vidět průběh vytváření aplikace.

```
1  forge
2  project-new --named conference
3  jpa-new-entity --named Speaker
4  jpa-new-field --named firstname
5  jpa-new-field --named surname
6  jpa-new-field --named bio --length 2000
7  jpa-new-field --named twitter
8  scaffold-generate --targets org.conference.model.Speaker
```

Obrázek 10 Postup pro vytvoření aplikace (podle[7])

Nyní stručně popíši, co těchto pár příkazů dokáže:

- Prvním příkazem se pustí forge.
- Druhým příkazem se vytvoří nový projekt, který obsahuje pouze prázdnou strukturu podle standardu Java EE 6
- Třetí příkaz vytvoří novou entitu, hned jsme přepnuti do kontextu entity
- Příkazy 4 až 7 entitě nastavíme pár atributů, aby to bylo možné musíme být v kontextu entity. Nyní jsme do něj byli přepnuti příkazem vytvoření entity, kdybychom chtěli entitu upravovat později, musíme se do ní přepnout příkazem *cd Speaker.java*. Příkaz cd slouží jak pro přechod mezi složkami, tak pro vstoupení do kontextu entity.
- Posledním příkazem už jen vygenerujeme zdrojové soubory applikace, kde parametr `--targets` určuje cestu k entitě
- Pro entitu Speaker se nám vytvoří stránky *create, view, search* a další obslužné soubory.

Tímto krátkým příkladem jsme si ukázali jaká síla se ve Forge 2 skrývá. Jedinou nevýhodou UI Scaffoldingu je, že nám neumožňuje zadat entity ve formě XML souboru, jak by firma CCA potřebovala.

```
C:\Data\pokus>forge
Using Forge at c:\devTools\tools\forge-distribution-2.20.2.Final

JBoss Forge, version [ 2.20.2.Final ] - JBoss, by Red Hat, Inc. [ http://forge.jboss.org ]

[pokus]$ project-new --named conference
***SUCCESS*** Project named 'conference' has been created.
[conference]$ jpa-new-entity --named Speaker
***SUCCESS*** Persistence (JPA) is installed.
***SUCCESS*** JPA Entity org.conference.model.Speaker was created
[Speaker.java]$ jpa-new-field --named firstname
***SUCCESS*** Field firstname created
[Speaker.java]$ jpa-new-field --named surname
***SUCCESS*** Field surname created
[Speaker.java]$ jpa-new-field --named bio --length 2000
***SUCCESS*** Field bio created
[Speaker.java]$ jpa-new-field --named twitter
***SUCCESS*** Field twitter created
[Speaker.java]$ scaffold-generate --targets org.conference.model.Speaker
***SUCCESS*** CDI has been installed.
***SUCCESS*** EJB has been installed.
***SUCCESS*** Servlet API has been installed.
***SUCCESS*** JavaServer Faces has been installed.
***SUCCESS*** Scaffold was generated successfully.
[Speaker.java]$
```

Obrázek 11 Ukázka průběhu vytvoření aplikace

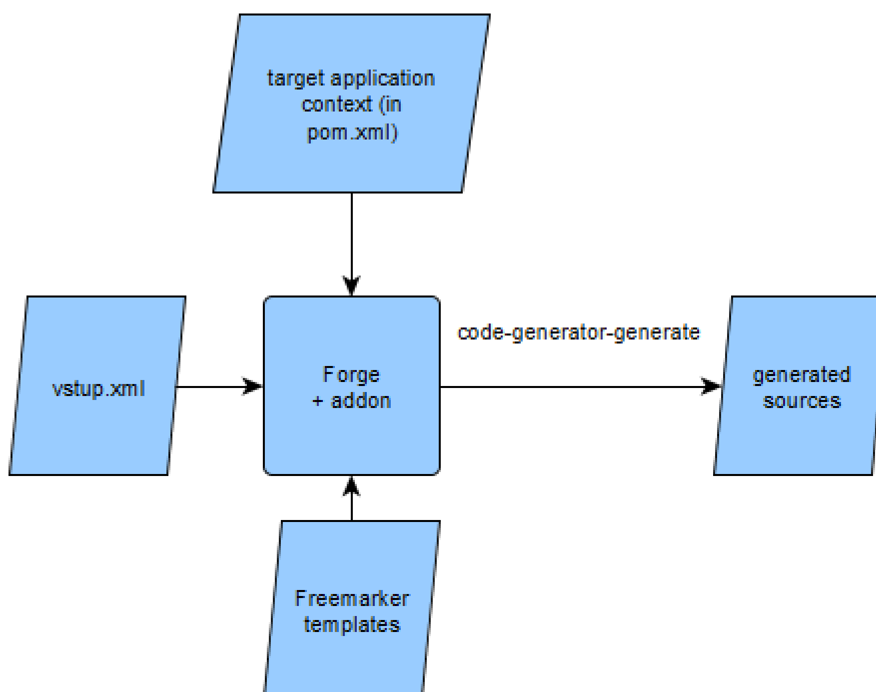
5. Návrh struktury konfigurace generátoru

5.1. Proces generování

Ze všeho nejdříve je nutné si říci, jak celý proces generování probíhá, včetně jeho okolí. Tedy co se děje před ním, během něj a po něm. Toto znázorňuje diagram na obrázku 9 (uvedený v kapitole 4.1.1.). A dále říci jakými částmi z tohoto procesu se zabývá tato diplomová práce a jakými ne.

Na začátku celého procesu je zákazník (člověk či skupina lidí, pro kterého je aplikace vyvíjena). Od něj se sesbírají požadavky na aplikaci, poté systémový architekt na základě těchto požadavků vytvoří návrh aplikace v programu Enterprise Architekt. Z EA se vyexportují data do XML souboru, který budu nazývat *export.xml*. Export dat byl již popsán v kapitole 4.1.1. Ze souboru *export.xml* se XSLT transformací vytvoří soubor, který nazvu *vstup.xml*. Zde je hranice teoretické a praktické části této diplomové práce.

Praktická část této práce je znázorněna na obrázku 12. Bylo třeba navrhnout vstupní XML pro generátor a sadu šablon, pro generované zdrojové soubory ve FreeMarker Template Language. Navrhnuté vstupní XML je popsáno v kapitole 5.2. Šablony jsou popsány v kapitole 6, protože jsou prozatím součástí projektu s generátorem.



Obrázek 12 Proces generování v kontextu DP

Vstupní soubor pro generátor bylo třeba navrhnout tak, aby data v něm obsažené bylo možné získat z EA. Pro účely této práce byla vstupní data zvolena ručně. Úpravou programu pro export a XSL souboru pro XSLT transformaci, by však tato data mohla být získána i cestou z EA.

5.1.1. XSLT transformace

XSLT transformace[8] (eXtensible Stylesheet Language Transformations) slouží k převodu zdrojových dat ve formátu XML do jiného formátu. Nejčastěji je to XML, HTML, ale může to být i jiná libovolná datová struktura. Samotná transformace se provádí za pomocí procesoru XSLT. Tento program si můžeme napsat podle pravidel XSLT transformace v libovolném programovacím jazyce nebo využít již napsanou XSLT knihovnu pro vybraný programovací jazyk. Pro XSLT transformaci jsou zapotřebí dva soubory. V prvním souboru budou zdrojová data, která chceme transformovat. Tento soubor musí splňovat jen obecné vlastnosti XML, žádné další požadavky na něj neklademe. Druhý soubor musí obsahovat vzorec pro transformaci a být napsán v jazyce XSL.

XSL (eXtensible Stylesheet Language) je jazyk, který umožňuje popsat, jak má být XML převedeno na jinou datovou strukturu. V příloze B, uvádím příklad, jak by mohl vypadat exportovaný soubor z Enterprise Architektu, k němu XSL soubor na převod a výstup po transformaci. Příklad je pouze ilustrační, pro nasazení s mým generátorem by musely být lehce upravena jak generovaná data, tak transformační soubor. Pro vytvoření si obrázku o tom, jak to funguje, je však postačující.

5.2. Struktura dat vstupního XML generátoru

Vstupní soubor pro generátor je uložen v souboru ve formátu XML. Na obrázku 13 je k nahlédnutí jeho fragment, který bude k popsání struktury postačovat. Celé je pak na obrázku 59 v příloze C. Kořenovým elementem je *generator-unit*, reprezentující jeden vstup pro generátor. XML v současné době může obsahovat pouze jednu *generator-unit.* Ta může obsahovat další elementy. Ty může rozdělit na dvě skupiny. První skupinou jsou elementy, které určují globální vlastnosti pro celou *generator-unit*. Druhou z nich jsou elementy pro generované stránky.

Do první skupiny patří tyto elementy:

- *entityName* – název generované entity (např. Funkce, ve smyslu pozice pracovníka ve firmě),
- *pageTitle* – titulek stránky – pro všechny stránky společný,
- *pageTemplate* – relativní cesta umístění šablony stránek modulů (*template.xml*),
- *dateTimePattern* – formát data a času.

Mezi elementy stránek patří:

- *detail, edit, list* – elementy v nichž jsou údaje pro generování modulů *Detail, Edit* a *List.*

Elementy uvnitř elementů stránek pak definují, jaké komponenty budou na stránce. Jedná-li se o stránku list, některé z nich upravují i její strukturu.

V současné době můžou stránky ve vstupním XML obsahovat tyto komponenty:

- *displayItem* – needitovatelný text
- *textItem* – editovatelný text
- *textArea* – textové pole
- *dateItem* – vstup pro datum
- *selectOneMenu* – menu pro výběr jedné z více položek
- *button* - tlačítko
- *commandLink* - odkaz
- *accordion* – panel s vertikálními záložkami
- *tabView* – panel s horizontálními záložkami

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <generator-unit>
3    <entityName>Funkce</entityName>
4    <pageTitle>Funkce pracovníka</pageTitle>
5    <pageTemplate>/templates/template.xhtml</pageTemplate>
6    <dateTimePattern>dd. MM. yyyy</dateTimePattern>
7    <detail>
8      <displayItem id="kodFunkce" label="Kód" field="kodFunkce"/>
9      ...
10     <button id="prirazeniDovednostiButton" label="Přiřazení dovedností">
11       <description>Otevře okno pro výběr dovedností, která mají být funkci přiřazeny</description>
12     </button>
13   </detail>
14
15   <edit>
16      ...
17     <textArea id="popisFunkce" label="Popis" maxlength="4000" rows="5" field="popisFunkce"/>
18     ...
19     <selectOneMenu id="pouzitiPovoleno" label="Použití povoleno" field="pouzitiPovoleno">
20       <option value="T">Ano</option>
21       <option value="F">Ne</option>
22     </selectOneMenu>
23   </edit>
24
25   <list selectionMode="multiple" header="Customers">
26    ...
27   </list>
28 </generator-unit>
```

Obrázek 13 Vstupní XML - fragment

6. Referenční implementace generátoru na vybrané technologii

6.1. Implementace generátoru

Ve firmě CCA se již dříve pokoušeli o implementaci generátoru. Byl k tomu zvolen Forge 2 od firmy JBoss. Znamenalo to napsat addon pro Forge 2, který se pak do tohoto nástroje nainstaloval a bylo jej pak možné pouštět jako nový příkaz Forge2. Generátor uměl načíst vstupní XML a vygenerovat stránky *Detail*, *Edit* a *List* a na nich pár základních komponent. Problém byl v tom, že tento návrh neumožňoval generovat složitější komponenty, jako je například *accordion* či *tabView*, kde jedna komponenta může obsahovat další komponenty. Ve své diplomové práci jsem tedy nezačínal „na zelené louce", ale využil jsem již existujícího projektu, který jsem musel upravit a rozšířit, aby vyhovoval požadavkům zadavatele.

Addon do Forge 2 je v podstatě jednoduchý Maven projekt, který musí splňovat několik podmínek. Struktura addonu vychází z inspirace zdrojovými kódy jádra Forge a již existujících addonů. Typicky jsou v projektu obsaženy tyto tři maven moduly:

- *addon* – obsahuje nutné nastavení Mavenu pro správné sestavení addonu
- *api* – obsahuje rozhraní výkonného kódu, který se používá v příkazech
- *impl* – obsahuje implementace výše uvedených rozhraní a samotné třídy s příkazy

Ve Forge 1.x se v addonu používala vstupní třída. Ve Forge 2 tomu již tak není, ale každý jeho příkaz se definuje jako třída, která je potomkem:

org.jboss.forge.addon.projects.ui.AbstractProjectCommand

nebo nějaké z rodičů této třídy.(Příklad na obrázku 14)

```
01. public class CodeGenerateCommand extends AbstractProjectCommand {
02.
03.     ...
04.
05.     @Inject
06.     private ProjectFactory projectFactory;
07.
08.     @Override
09.     protected boolean isProjectRequired() {
10.         return true;
11.     }
12.
13.     @Override
14.     protected ProjectFactory getProjectFactory() {
15.         return projectFactory;
16.     }
17.
18.     @Override
19.     public UICommandMetadata getMetadata(UIContext context) {
20.         return Metadata
21.                 .forCommand(CodeGenerateCommand.class)
22.                 .name("Code Generator: Generate")
23.                 .description("Generuje JSF a podpurny Java kod na zaklade vstupnich parametru.")
24.                 .category(Categories.create("CCA Code Generator"));
25.     }
26.
27.     @Override
28.     public void initializeUI(UIBuilder builder) throws Exception {
29.         builder.add(inputType).add(inputParametersFile);
30.     }
31.
32.     @Override
33.     public Result execute(UIExecutionContext context) throws Exception {
34.         // Sem přijde samotná logika toho co má příkaz dělat
35.         return Results.success();
36.     }
37.
38. }
```

Obrázek 14 Code generator command

Důležitá je tato metoda na obrázku 15:

```
1. @Override
2. public UICommandMetadata getMetadata(UIContext context) {
3.     return Metadata
4.             .forCommand(CodeGenerateCommand.class)
5.             .name("Code Generator: Generate")
6.             .description("Generuje JSF a podpurny Java kod na zaklade vstupnich parametru.")
7.             .category(Categories.create("CCA Code Generator"));
8. }
```

Obrázek 15 UIMetadata

V této metodě se nadefinuje vše potřebné spojené se zobrazováním jména a popisu pluginu v textovém GUI Forge 2. Ve Forge 2 bude výsledné jméno příkazu *code-generator-generate* (jméno se odvodí z toho, co je uvedeno v jako parametr metody *name()*, tedy z *"Code Generator: Generate")*.

Dále je potřeba příkazu přidat parametry. Ty se definují jako CDI (Context Dependency Injection) injekce potomků třídy *InputComponent*. V následujícím příkladu je vidět použití anotace *@WithAttributes*, díky níž lze vyplnit informace o parametru (popis, povinnost, …). Následující příklad vypíše pouze hodnoty parametrů. (viz obrázek 16).

```
01. public class CodeGenerateCommand extends AbstractProjectCommand {
02.
03.    @Inject
04.    @WithAttributes(
05.          label = "Context provider",
06.          description = "Zdroj vstupnich parametru.",
07.          required = true
08.    )
09.    private UISelectOne<ContextProviderType> inputType;
10.
11.    @Inject
12.    @WithAttributes(
13.          label = "Input parameters file",
14.          description = "Soubor se vstupnimi parametry.",
15.          required = true
16.    )
17.    private UIInput<FileResource<?>> inputParametersFile;
18.
19.    ...
20.
21.    @Override
22.    public Result execute(UIExecutionContext context) throws Exception {
23.       System.out.println("INPUT TYPE: " + inputType.getValue());
24.       System.out.println("INPUT PARAMETER FILE: " + inputParametersFile.getValue());
25.       return Results.success();
26.    }
27.
28. }
```

Obrázek 16 CodeGeneratorCommand příklad

První anotace říká, že parametr *inputType* bude typu*UISelectOne*, což je výběr z hodnot uvedených v enumu *ContextProviderType*. Druhý parametr reprezentuje jednoduchý vstup typu soubor.

Dále je ještě zajímavá metoda *initializeUI*, která inicializuje uživatelské rozhraní. Je v ní možné přidat již zmiňované vstupní atributy. Hodnotu vstupního atributu pro další použití lze získat metodou *getValue()* a účelně použít například takto při volání příkazu pro generování (viz obrázek 17).

```
@Override
public Result execute(UIExecutionContext context) throws Exception {
    codeGenerator.generate(getSelectedProject(context), inputType.getValue(),
    inputParametersFile.getValue());
    return Results.success();
}
```

Obrázek 17 Metoda execute

V metodě *execute* se pak volá funkční logika toho, co má příkaz dělat. Vrátíme-li se ke konkrétnímu příkladu generátoru, je volána metoda *generate* rozhraní *CodeGenerator*, které je do třídy *CodeGeneratorComand* injektováno a implementuje ho třída *CodeGeneratorImpl*.

Třída *CodeGeneratorImpl* tedy obsahuje vše důležité pro vytvoření a zavolání akcí spojených s novým příkazem do Forge 2. Návrh této metody dělal pan Ing. Miroslav Král, který se zabýval vývojem pluginu přede mnou. Tato třída je pro celý plugin

klíčová, proto popíši, co se v ní děje. Nejdříve je však ještě nutné zmínit kontext v jakém se nový příkaz bude volat.

Výsledkem má být jeden nový příkaz do Forge 2, který se bude jmenovat *code-generator-generate*. Do uživatelského rozhraní Forge 2 se dostane tak, že ve složce obsahující hlavní *pom.xml* Maven projektu s pluginem, zavoláme příkaz *mvn clean install* (případně jen *mvn install*). Tímto plugin nainstalujeme do lokálního Maven repozitáře. Poté se zavoláme příkaz *forge --install cz.cca.forge:code-generator,2.5.1-SNAPSHOT,* který zajistí, že se plugin z lokálního repozitáře přidá do uživatelského rozhraní Forge 2. Pokud sestavení a nainstalování pluginu dopadne úspěšně, přibude při dalším puštění Forge 2 nový příkaz *code-generator-generate*.

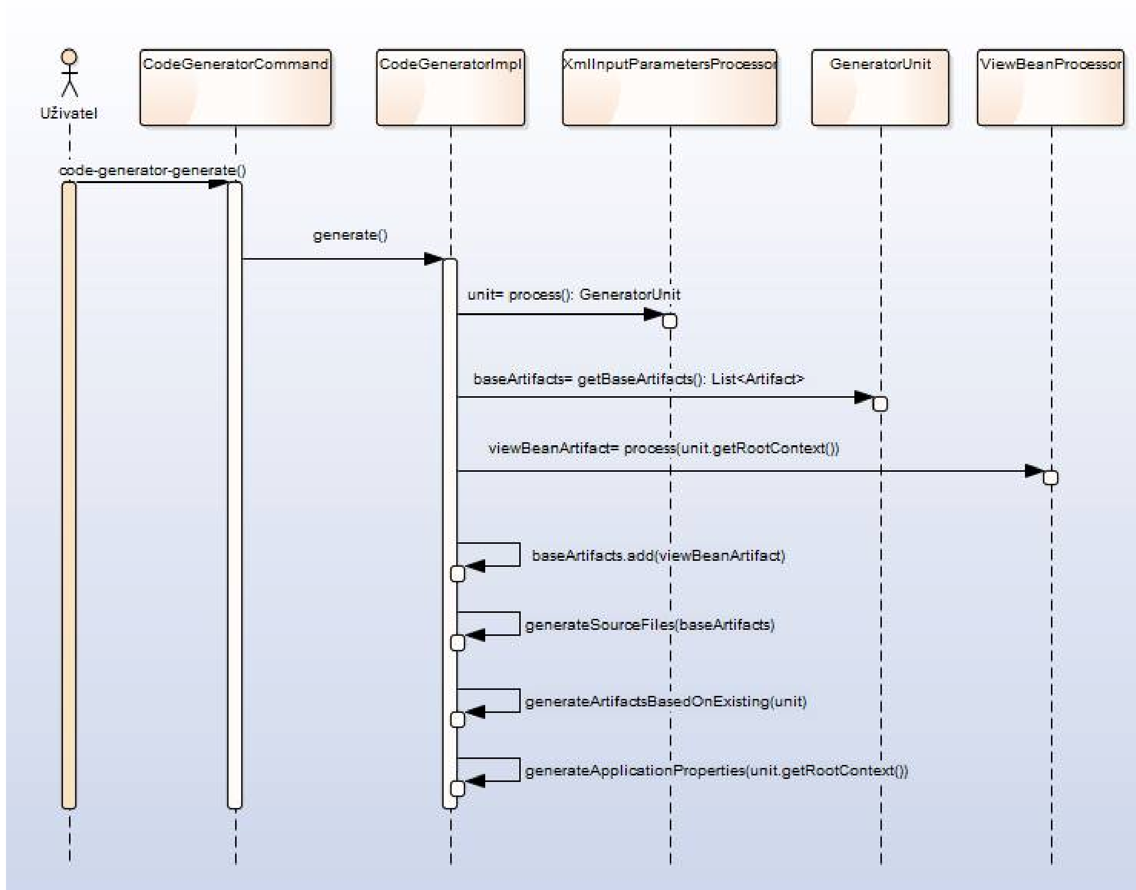
Tento příkaz je možné volat jen v určitém kontextu. Cílová aplikace je založena na PrimeFaces, což je Java EE technologie, a tak bude tímto kontextem kořenový adresář modulu *war*. V tomto adresáři se musí nacházet soubor *pom.xml* pro modul *war* a soubor se vstupním XML. Nazvěme ho pro jednoduchost například *vstup.xml*. Z *pom.xml* Forge 2 získá informace o adresářové struktuře modulu na základě specifikace Java EE aplikace.

Generátor se pustí tak, že se v adresáři *war* pustí příkazová řádka a v ní se zavolá příkaz *forge*, který nás přepne do testového uživatelského rozhraní Forge 2 (konzole). V konzoli Forge 2. V konzoli Forge 2, je teď možné volat jeho základní příkazy a úspěšně nainstalované addony/pluginy. Příkaz pro generování bude vypadat takto:

```
code-generator-generate --inputType XML --inputParametersFile vstup.xml
```

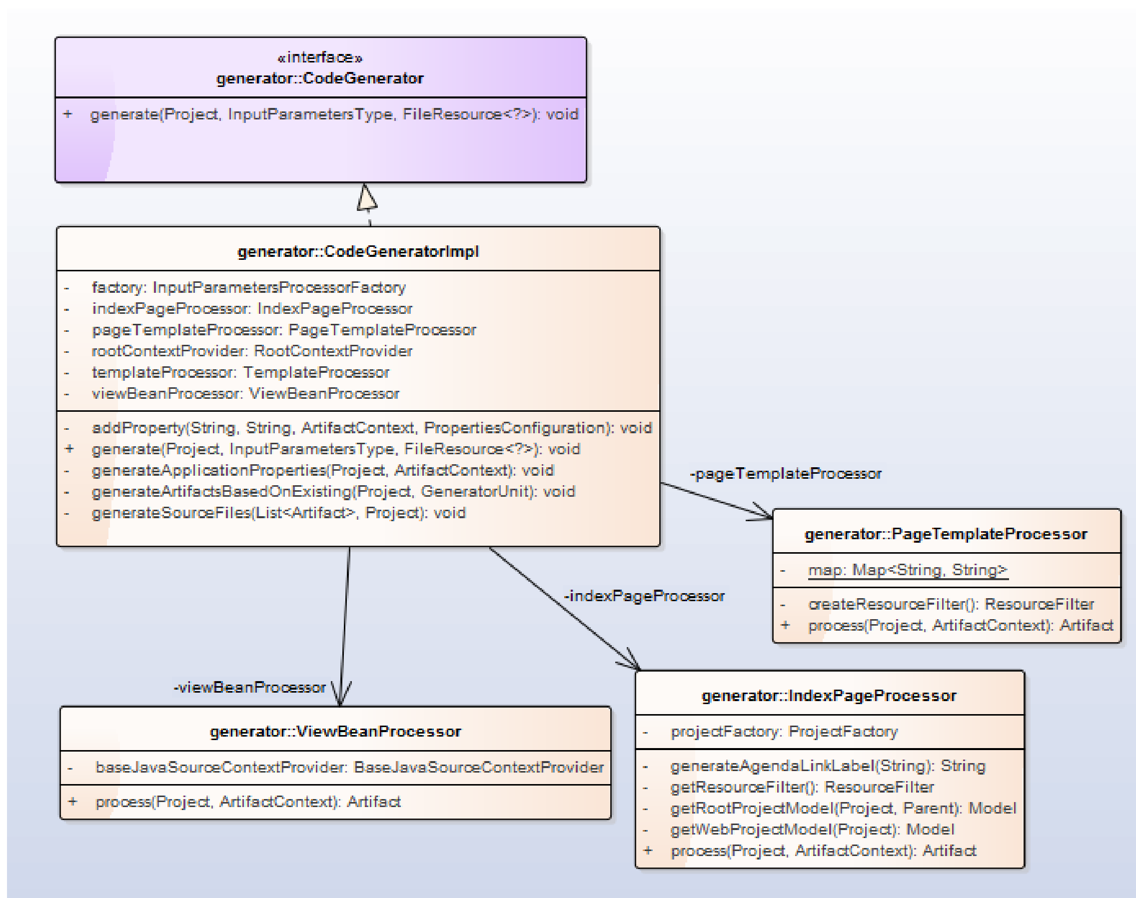
Parametr *inputType XML* říká, že vstup pro generátor bude XML soubor. Možné hodnoty pro *inputType* jsou *EA* a *XML*. Možnost *EA* je připravena pro případ, že by vstupem generátoru byl přímo výstup z Enterprise Architektu, není však implementovaná a je mezi vzdálenějšími cíli pro další vývoj pluginu. Nicméně konstrukce *UISelectOne<InputParametersType>*, kde enum *InputParametersType* obsahuje výčet všech hodnot, nastiňuje, jak snadno přidat další typ vstupu. Hodnota parametru *inputParametersFile* určuje cestu ke vstupnímu souboru.

Co se děje uvnitř addonu po zavolání příkazu *code-generator-generate*, nastiňuje diagram na obrázku 18. Některé implementační detaily zde popíši podrobněji, protože jejich pochopení bude zdrojem poznání pro další podkapitoly, které popisují, jak je možné addon dále rozšířit a jaké kroky se k tomu musí podniknout.



Obrázek 18 Volání coge-generator-generate

Nyní je možné se opět vrátit k popisu třídy *CodeGeneratorImpl*. Tato třída obsahuje implementaci metody *generate* volané v metodě *execute* třídy *CodeGenerateComand*, dále obsahuje tři privátní metody, které jsou volány z těla této metody, a čtvrtou volanou v poslední z těchto tří. Pro lepší pochopení přikládám class diagram zachycený na obrázku 19.



Obrázek 19 CodeGenerator class diagram

Níže uvádím hlavičku metody *generate* a popíši její parametry:

```
@Override
public void generate(final Project webProject,
                     final InputParametersType inputParametersType,
                     final FileResource<?> inputFile)
        throws CodeGeneratorException {
…
}
```

Parametr *webProject* typu *org.jboss.forge.addon.projects.Project* je důležitý v tom, že skrze něj jsou do aplikace vnesené informace o Java EE aplikaci, do níž budou generovány výstupní soubory, tyto informace jsou získané z *pom.xml* v modulu *war* cílové Java EE aplikace, o němž už jsem se zmiňoval v předchozím textu. Standardem Java EE aplikace je jasně daná adresářová struktura aplikace, součástí níž, je například to, kde mají být umístěné JSF stránky *(*.xhtml* zdrojové soubory) a kde podpůrné java soubory, které slouží jako controlery k těmto stránkám a podobně. Tento parametr tedy uchovává kontext cílové Java EE aplikace. Další dva parametry *inputParametersType* a *inputFile* jsou již vysvětlené vstupní parametry generátoru.

Nyní nahlédneme na chování uvnitř metody *generate*. Ze všeho nejdříve je potřeba získat správný *InputParametersProcessor*. Ten se získá na základě hodnoty vstupního

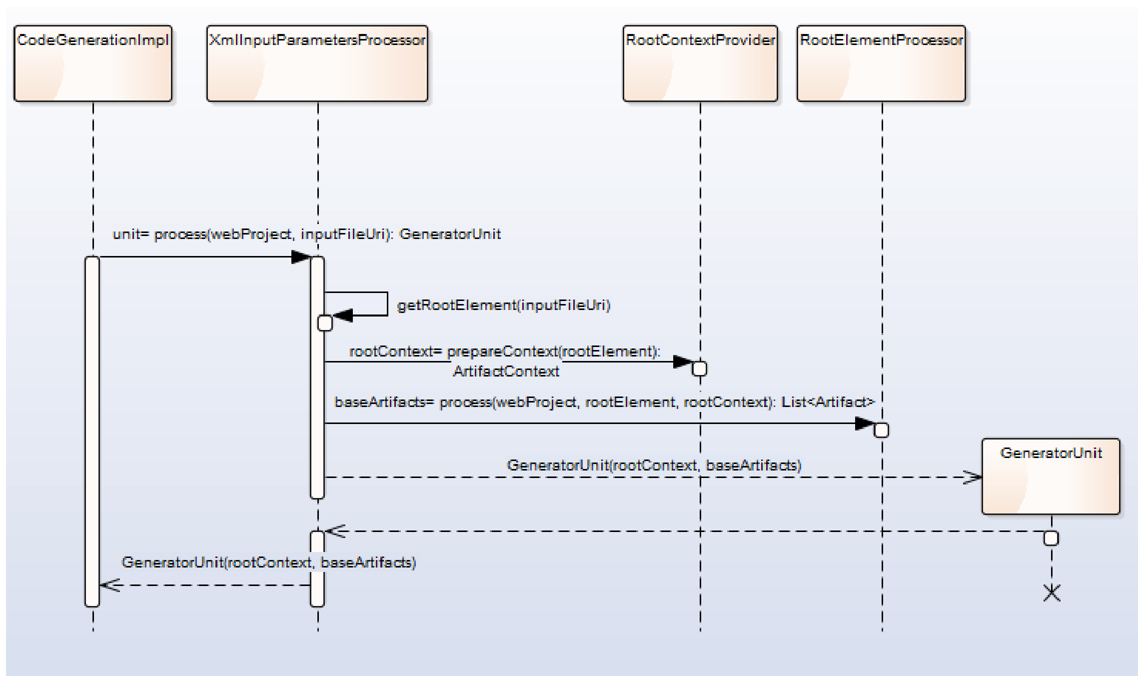
parametru *inputType* z tovární třídy *InputParametersProcessorFactory*. V současné době je implementován pouze jeden procesor vstupních parametrů, který zastřešuje vstup z xml souboru a je implementován ve třídou *XmlInputParametersProcessor*. Nad *InputParametersProcessorem* se zavolá metoda *process*, která má na vstupu kontext cílové aplikace a cestu ke vstupnímu *xml*. Tato metoda vrátí objekt typu *GeneratorUnit*. (viz obrázek 20)

Obrázek 20 GneratorUnit

GeneratorUnit obsahuje kolekci *baseArtifact*, což jsou základní Artefakty pro generované stránky (*Edit, Detail, List)*. Pro každou z nich obsahuje jeden Artefakt definující *xhtml* soubor pro webovou stránku a jednu Java Action Bean (opět pro všechny tři). Atribut *rootContext* obsahuje základní údaje o stránce (*entityName, title, apod.)*.

GeneratorUnit je datová reprezentace toho, co je ve vstupním *xml*. její obsah je následně použit jako zdroj dat pro šablony, ze kterých se pak generují soubory, které jsou výstupem generátoru.

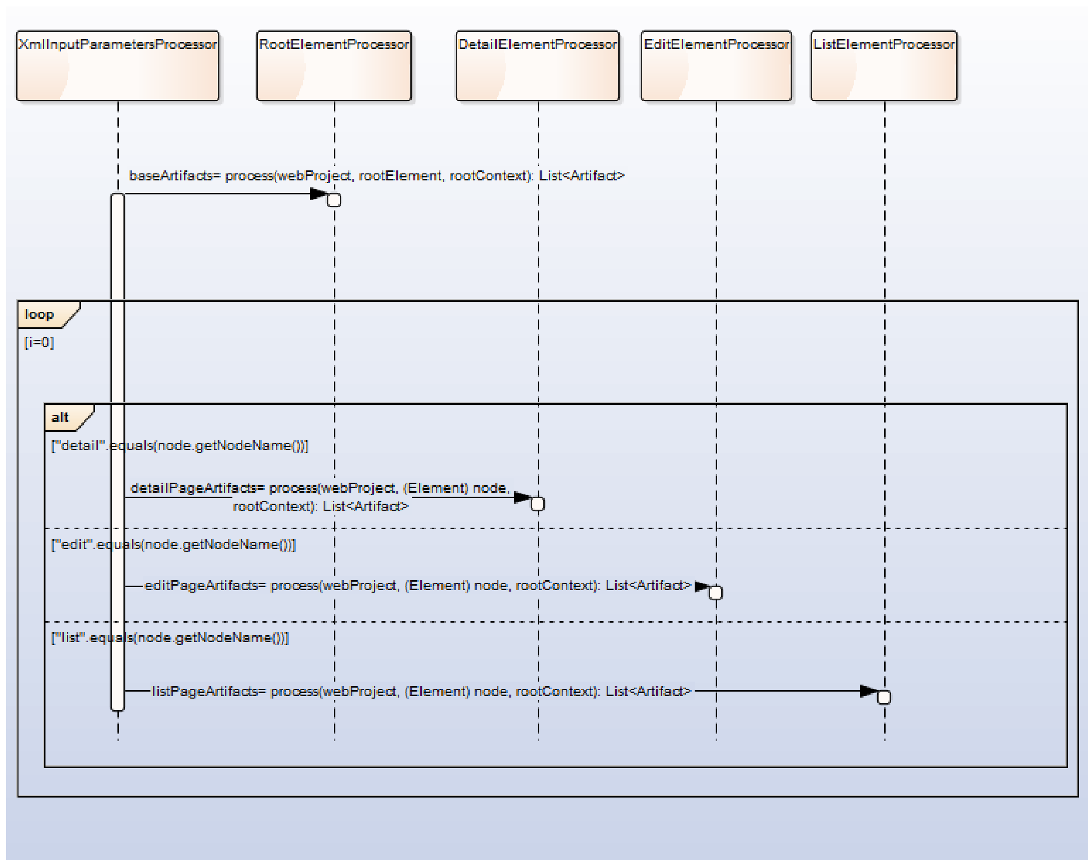
Jak se tyto data získají, ukazují následující diagramy. První z nich (obrázek 21) ukazuje, co se děje, když se v metodě *generate* třídy *CodeGeneratorImpl*, zavolá nad třídou *XmlInPutParametersProcessor* metoda *process.*



Obrázek 21 metoda process

Další diagram (obrázek 22) detailněji ukazuje, jak se získají *baseArtifacts*, tedy detail volání metody *process* nad instancí třídy *RootElementProcessor*. V cyklu se prochází *rootElement* a konstrukcí:*final Node node = rootElement.getChildNodes().item(i);*

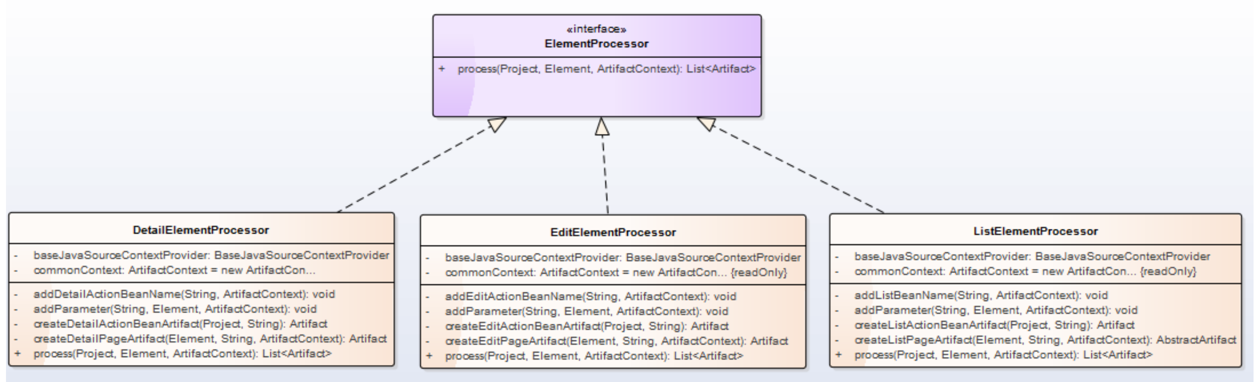
se získá aktuální uzel představující ve vstupním xml jednu stránku (detail, edit, list). Pro každou stránku je samostatný *ElementProcessor*, který zpracuje stránku.



Obrázek 22 Root Element Processor

6.1.1. **Zpracování stránek ze vstupního XML**

Je potřeba zpracovat informace o stránkách *Detail, Edit a List*. Každou stránku má na starosti samostatná třída, která je potomkem třídy *ElementProcessor*. Pro zpracování stránky je třeba implementovat metodu *process*. V této metodě se volají další metody, které mají na starosti jednotlivé akce. Výsledkem má být kolekce se dvěma Artefakty. Jeden s informacemi o JSF stránce a druhý s informacemi o Java Action Beaně.



Obrázek 23 ElementProcessor

Strukturu tříd a jejich metod pro zpracování stránek znázorňuje class diagram na obrázku 23.

Pro zpracování stránky je třeba udělat tyto čtyři kroky:

1. Získat název entity pro, kterou jsou stránky generovány (například Funkce, ve významu pozice pracovníka ve firmě). Její název je ve vstupním elementu uložen v uzlu *entityName*
2. Vytvořit *Artifact* pro JSF stránku, to má za úkol metoda *create<NázevStránky>Artefakt*. Pro stránku detail to bude například *createDetailPageArtifact*. V případě entity Funkce a stránky detail, bude toto zdroj informací pro generování stránky *FunkceDetailPage.xhtml*.
3. Vytvořit *Artifact* pro Java Action Beanu – java třídu s obslužnými metodami, tzn. controller pro JSF stránku. Pro stránku detail se je to metoda *createDetailActionBeanArtifact*, pro ostatní stránky je to obdobné. V případě entity Funkce a stránky *Detail*, bude toto zdroj informací pro generování souboru *FunkceActionBean.java*, ve kterém bude java třída *FunkceActionBean*.
4. Artefakty z bodů 2 a 3 jsou uloženy do kolekce typu List a vráceny na výstupu metody *process*. V postupné hierarchii volání metody *process* i v třídách předtím, jsou tyto dva artefakty přidány do kolekce *baseArtifact* v *GeneratorUnit*.

Pozn.: Každý *Artifact* má svůj *ArtifactContext*, ve kterém jsou uložena zpracovaná data ze stránky, jako je například *entityName* z bodu 1 a komponenty na stránce, název souboru, do kterého bude *Artifact* generován (např. *FunkceDetailPage.xhtml* či *FunkceDetailActionBean.java*) a také šablona, která má být použita pro generování šablony (např. *detail.ftl* pro JSF stránku detail či *detail-action.ftl* pro Java Action Beanu stránky *Detail*). Tyto informace jsou pak využity při generování stránek na základě šablon ve třídě *TemplateProcesor*.

Ve své diplomové práci jsem použil základní strukturu tříd pro zpracování stránek, kterou navrhl pan Ing. Miroslav Král přede mnou, ale obsah metod pro zpracování komponent na jednotlivých stránkách bylo nutné téměř celý napsat znovu, včetně vytváření jmen souborů, do kterých jsou generovány a šablon pro generování, které byly kompletně přepsány. Veškeré komponenty na jednotlivých stránkách jsou nově ukládány v rekurzivní struktuře, která umožňuje uchovat informaci o tom, jako jsou do sebe komponenty vnořeny. Návrh datových objektů pro uložení získaných dat stránek je blíže popsán v kapitole 6.1.2 a k nim příslušné parsery, které je plní, jsou popsány v kapitole 6.1.3.

6.1.2. **Datové objekty pro uložení komponent stránek**

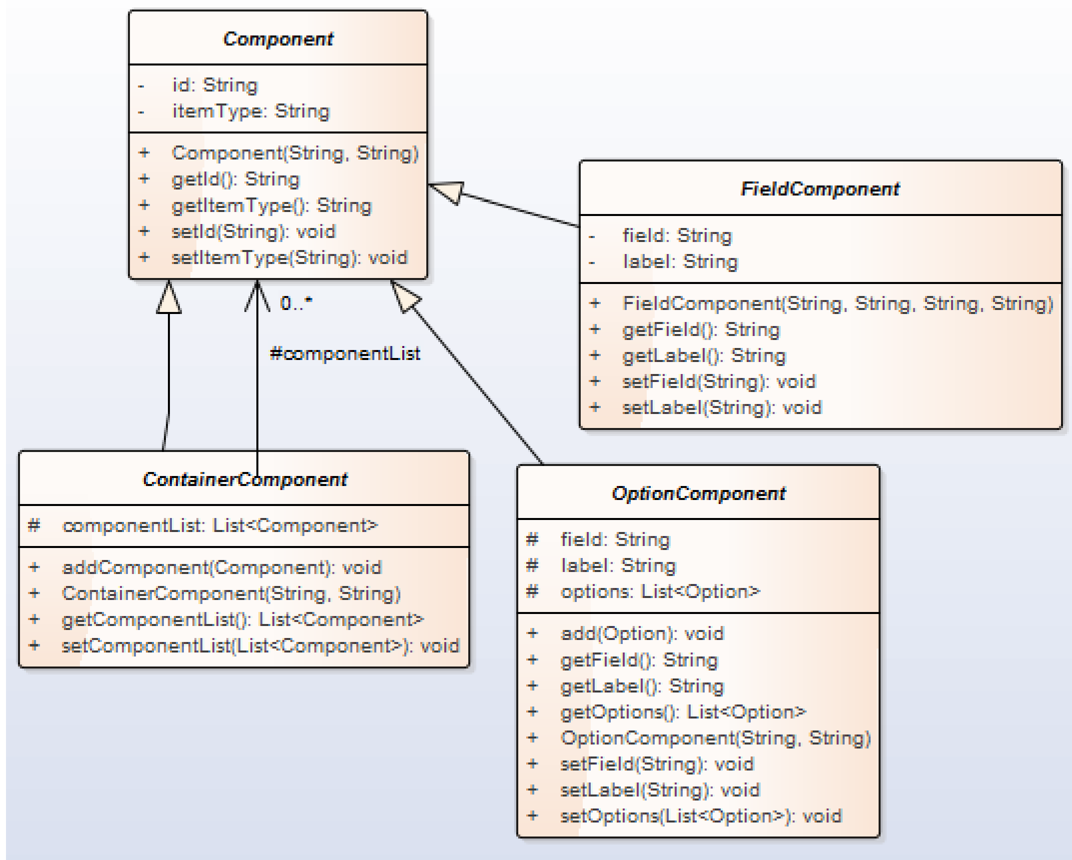
Při návrhu datových objektů pro uložení komponent jsem musel zohlednit několik faktů:

- Každá komponenta na stránce má název a atributy. Mělo by být snadné přidat novou komponentu nebo nějaký atribut komponenty.
- Některé komponenty mohou obsahovat další komponenty. Pokud je komponenta zanořena uvnitř nějaké jiné komponenty, je třeba tuto informaci nějak jednoduše uchovat.
- Datové objekty, do kterých budou komponenty uloženy, budou použity při generování souborů za využití šablon. Protože pro generování obsahu zdrojových souborů, které jsou výstupem generátoru, využívám šablonovací nástroj FreeMarker, bylo nutné navrhnout datové objekty tak, aby s nimi FreeMarker byl schopen pracovat. Zároveň jsou navrženy, dostatečně obecně a jednoduše, aby mohl být FreeMarker v případě potřeby vyměněn za nějakou jinou knihovnu.

Pro splnění podmínek z předchozích bodů, jsem podnikl tyto kroky:

- Vytvořil jsem abstraktní třídu *Component*, která reprezentuje jakoukoliv komponentu na stránce. Každá komponenta tedy musí dědit tuto třídu nebo některého jejího potomka.
- Dále jsem vytvořil abstraktní třídu *ContainerComponent*, která reprezentuje komponentu, která v sobě může mít zanořené další komponenty. Třída *ContainerComponent*, je potomkem třídy *Component*, protože taková komponenta je stále komponentou.
- Na základně požadavků zadavatele by mělo být možné, aby každá komponenta měla své id. Najdou se sice komponenty, u kterých je momentálně id nevyužité, ale to je dané pouze strukturou vstupního xml, které se bude určitě ještě rozšiřovat. Výsledkem tohoto požadavku je atribut *id* ve třídě *Component*.
- Pro potřeby zpracování ve FreeMarkeru je nutné nějaký způsobem z datového objektu rozlišit o jaký typ komponenty se jedná. Datové objekty jsou třídy napsané v Javě a FreeMarker s nimi do jisté míry umí pracovat, nicméně je nutné si uvědomit, že FreeMarker není Java a některé konstrukce, které jsou možné v javě FreeMarker neumožňuje. Z tohoto důvodu má třída *Component* atribut *itemType* a tím, že každý datový objekt reprezentující nějakou komponentu je buď přímým, nebo nepřímým potomkem této třídy, je zajištěno, že na základě tohoto atributu FreeMarker rozpozná, o jakou komponentu se jedná. *ItemType* je plněn při vytváření komponenty v příslušném parseru pro danou komponentu, aby byly všechny údaje o datovém objektu komponenty zadávány na jednom místě.
- Protože FreeMarker pracuje převážně s řetězci, přestože má v sobě zabudované funkce, které do jisté míry umí pracovat i s čísly, jsou všechny atributy datových objektů ukládány jako řetězec. Takže každý atribut je typu *String*.

- Atributy datových objektů komponent jsou všechny *protected*, aby bylo možné v případě potřeby od komponenty vytvořit další komponentu využitím dědičnosti.
- Třída je standardní JavaBean, tzn. je nutné napsat gettery, settery a dodržovat *CamelCase* notaci. Pokud se na to zapomene, FreeMarker bude mít problém se dostat k hodnotám atributů.



Obrázek 24 Component Class Diagram

Na obrázku 24 je vidět class diagram pro komponenty. Všechny čtyři třídy jsou abstraktní třídy, od kterých se dědí další komponenty. V úplné obecnosti bych si nejspíš vystačil s předky *Component* pro jednoduchou komponentu a *ContainerComponent* pro komponentu, která může obsahovat další komponenty, ale některé komponenty měli specifické společné vlastnosti, tak jsem se rozhodl takovým komponentám přidělit společného předka, který je opět potomkem *Component*.

Mezi komponenty typu *FieldComponent* jsem zařadil (použitá jména odpovídají názvům, ze vstupního xml):

- *commandLink*, která reprezentuje *commandLink*, což je text, který se chová jako tlačítko
- *dateItem,*
- *displayItem,*
- *textArea,*
- *textItem.*

Tyto komponenty mají společné to, že jsou spojeny vždy s nějakým atributem typu *field*, kde je uložena jejich hodnota (odtud název skupiny) a pak každá z nich má *label*, v němž je uložen popisek (label) komponenty.

Mezi komponenty typu *OptionComponent* se momentálně řadí pouze komponenta *selectOneMenu*, ale patřila by sem například i komponenta *selectOneRadio*, kterou jsem neimplementoval, ale v dalším rozšíření generátoru bude nejspíš implementována. Tyto komponenty mají společné to, že mají seznam možností, ze kterých se vybírá právě jedna.

Mezi komponenty typu *ContainerComponet* se řadí komponenty, které můžou obsahovat libovolného potomka *Component*. Zařadil jsem mezi ně:

- komponenty *detail, edit, list*, které reprezentují generované stránky
- *tab*, který reprezentuje záložku v komponentě *accordion* či *tabView*, po předchozí analýze a konzultaci se zadavatelem jsem se rozhodl, že *accordion* i *tabView* budou využívat pro reprezentaci záložky (tabu) jeden společný typ komponenty. Druhou uvažovanou možností bylo mít separátní komponenty *accordionTab* a *tabViewTab*, ale nepřineslo by to žádný užitek naopak jen opakování kódu.
- Komponenty *dataTable* a *filter*, které jsou v současné použité jen na stránce list.

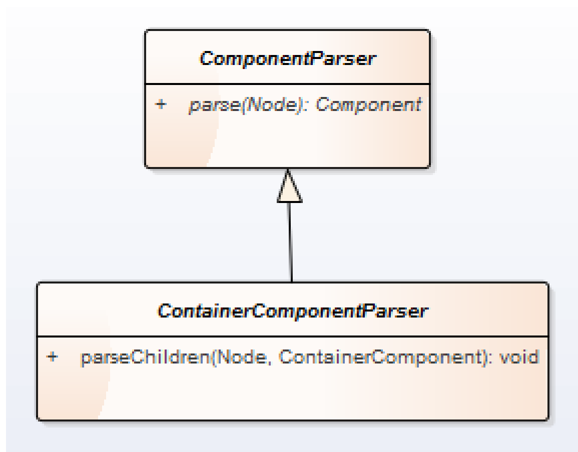
Dále jsou tu komponenty, které jsou potomky přímými potomky třídy *Component*, patří mezi ně:

- *accordion* a *tabView*, obě mají kolekci *tabList*, která obsahuje kolekci záložek (komponent typu *tab*)
- *button (*reprezentován jako commandButton*)*

6.1.3. **Parsery komponent stránek**

Parsery komponent jsou potomky *ComponentParser* nebo *ContainerComponentParser*, který je potomek *ComponentParser*. Obojí jsou to abstraktní třídy. *ComponentParser* obsahuje abstraktní metodu *parse*. Tím je zajištěno, že jí musí každý konkrétní parser (neabstraktní třída) implementovat. Tato třída slouží k parsování příslušné komponenty. ContainerComponentParser obsahuje navíc metodu *parseChildren*, ve které je implementováno parsování komponent, které jsou vnořeny do komponenty typu *ContainerComponent.* Toto parsování probíhá vždy stejně a tímto se předejde redundanci programového kódu a urychlí se implementace parserů.

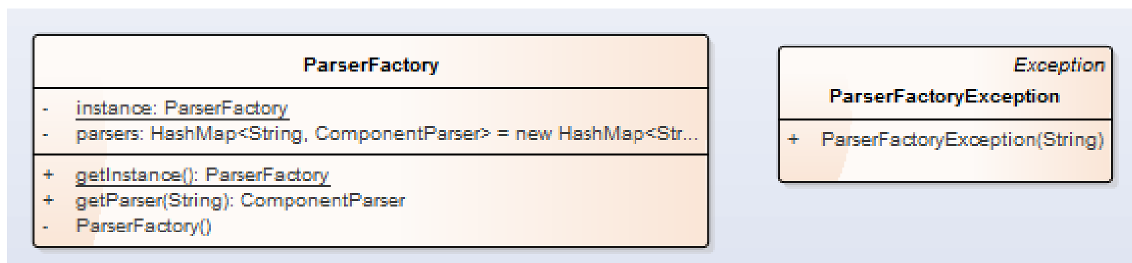
Parsery jsou tedy vytvořeny podle jednoduchého pravidla: Pokud jde o parser pro komponentu jenž je potomkem *ContainerComponent*, bude parser potomkem *ContainerComponentParser*, je nutné implementovat metodu *parse* a uvnitř ní zavolat metodu *parseChildren*, která zajistí parsování vnořených komponent. Parsery pro všechny ostatní komponenty tedy komponenty, které jsou potomky *Component*, ale nejsou potomky *ContainerComponent*, budou potomky *ComponentParser* a je u nich nutné pouze implementovat metodu *parse*. Na obrázku 25 je class diagram zachycující strukturu předků pro parsery.



Obrázek 25 ComponentParser class diagram

Aby se při každém použití parseru nemusela vytvářet nová instance parseru, spravuje parsery tovární třída *ParserFactory*, která je navržena podle návrhového vzoru Singleton, aby byla v aplikaci pouze jedna instance této třídy. Pokud chceme tedy použít některý z parserů, je nejprve třeba získat instanci třídy *ParserFactory*, což zajišťuje její statická metoda *getInstance*, ta při prvním zavolání zavolá privátní konstruktor třídy a uloží instanci *ParserFactory* do privátní statické proměnné *instance*, při dalším volání se již nová instance nevytváří, ale vrátí se reference na ní právě z proměnné *instance*. Instance parserů, které je možné použít, jsou uložené v privátní

HashMapě *parsers*. Do této mapy jsou vkládány v privátním konstruktoru třídy, kdy klíčem do mapy je řetězec s názvem komponenty ze vstupního xml souboru a hodnotou je reference na instanci parseru. Pro získání instance parseru slouží metoda *getParser*, jejímž vstupem je název komponenty ze vstupního xml. Díky tomu je pak možné jednoduše získat správný parser na základě názvu parsované komponenty. Případ, kdy bychom chtěli parsovat objekt, jehož parser není přidán do *ParserFactory (*obrázek 26*)*, je ošetřen výjimkou *ParserFactoryException*, která nám oznámí, že se jedná o nepodporovaný objekt spolu s názvem nepodporovaného objektu, který jsme chtěli parsovat.



Obrázek 26 ParserFactory a ParserFactoryException

Při parsování vnořených komponent ze vstupního XML jsem narazil na jeden problém, který byl třeba vyřešit. Podívejme se na fragment kódu (obrázek 27), který by mohl být obsažen ve vstupním XML. Máme tu *accordion* s jednou záložkou (*tab*), ve které může být více dalších různých komponent dále pak *selectOneMenu*, které v sobě může mít vnořeny jen elementy *option*. Intuitivní řešení by bylo projít všechny vnořené elementy a ty pak zprasovat. Nastává tu však problém, že mezi elementy, které nás zajímají, se často vyskytují různé bílé znaky, jako jsou mezery či tabulátory.

```
1  <accordion>
2    <tab title="titulek">
3      <textItem id="nazevFunkce" label="Název" field="nazevFunkce"/>
4      <textArea id="znalosti" label="Znalosti" maxlength="4000" rows="5" field="znalosti"/>
5    </tab>
6  </accordion>
7  <selectOneMenu id="pouzitiPovoleno" label="Použití povoleno" field="pouzitiPovoleno">
8    <option value="T">Ano</option>
9    <option value="F">Ne</option>
10 </selectOneMenu>
```

Obrázek 27 Fragment vstupního xml

V prvním případě (obrázek 28), kdy chci získat vnořené komponent v uzlu *tab*, mi pomůže vlastnost uzlu *nodeType*, kterou porovnám s konstantou *org.w3c.dom.Node. ELEMENT_NODE*, díky tomuto triku se zbavím bílých znaků. A dostanu jen uzly s komponentami, které můžou mít různá jména.

```
1  for (int i = 0; i < node.getChildNodes().getLength(); i++) {
2          final Node childNode = node.getChildNodes().item(i);
3          if(childNode.getNodeType() == Node.ELEMENT_NODE) {
4
```

Obrázek 28 Výběr elementů jen typu ELEMENT_NODE

Existuje ještě jeden případ, kdy mám například *selectOneMenu*, které v sobě může mít zanořené pouze možnosti, ze kterých lze vybírat (pouze uzel nazvaný *option*). Nemůže v sobě například obsahovat jinou komponentu jako například *displayItem*. V tomto případě se hodí použít vlastnost *nodeName* a porovnat jí na jméno uzlu, který může obsahovat. Ostatní komponenty a bílé znaky budou ignorovány (viz obrázek 29).

```
1  for(int i = 0; i < node.getChildNodes().getLength(); i++) {
2          Node optionNode = node.getChildNodes().item(i);
3          if("option".equals(optionNode.getNodeName())) {
4
```

Obrázek 29 Výběr elementů podle názvu

6.2. Popis šablon

Plugin používá pro generování Java kódu a JSF(resp. Primefaces) knihovnu FreeMarker. Je to jednoduchý šablonovací nástroj určený pro generování libovolného obsahu a pro naše účely se skvěle hodí (navíc jej používá i originální Forge plugin UI Scaffolding). V tomto pluginu jsou šablony uloženy v adresáři:

```
/src/main/resources/templates
```

Celá myšlenka je založena na jednoduchém nahrazování proměnných hodnotami. Zde je jednoduchá ukázka šablony pro ViewBeanu (obrázek 30):

```
1  package ${package};
2
3  import cz.cca.common.wf.ui.model.AbstractEditableViewBean;
4  import cz.cca.ra.db.model.CcalFunkce;
5  import ${entity};
6
7  public class ${entityName}ViewBean extends AbstractEditableViewBean<Ccal${entityName}> {
8
9      private static final long serialVersionUID = 1L;
10
11     public ${entityName}ViewBean(final Ccal${entityName} view) {
12         super(view);
13     }
14
15 }
```

Obrázek 30 Šablona pro ViewBean.java

Ostatní šablony jsou trochu komplikovanější. Zejména šablony pro generování JSF(Primefaces) stránek. Návrhem šablon, který jsem dělal, celý od znova, včetně vysvětlení potřebných znalostí o FreeMarkeru se budu zabývat v následujících podkapitolách.

Všechen inicializační kód FreeMarkeru je v modulu *impl* a třídě:

```
cz.cca.forge.addon.code.generator.template.TemplateProcessor
```

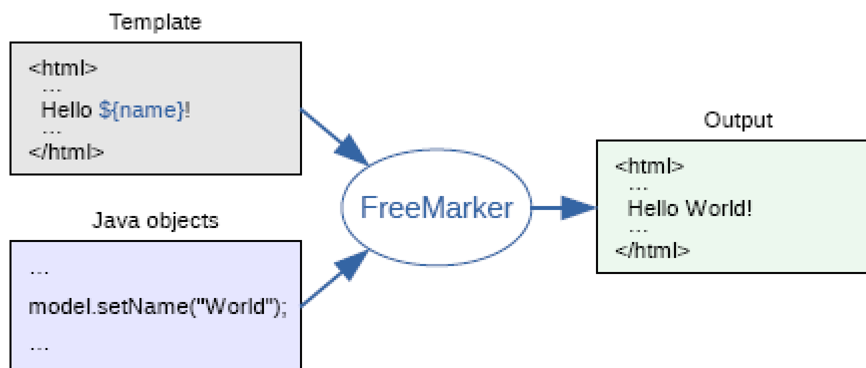
6.2.1. **Fremarker[12]**

Jak jsem již zmínil v předchozím textu, knihovna FreeMarker je šablonovací nástroj využívající princip nahrazování proměnných hodnotami. V případě, že chceme do generování šablon zavést nějakou složitější logiku, nebude nám pouze prosté nahrazování proměnné hodnotou stačit.

Šablony jsou psány ve *FreeMarker Template Language (FTL)*, což je jednoduchý, specializovaný jazyk. Data, která chceme zobrazit, je třeba si předem předpřipravit v některém jiném programovacím jazyce jako je například java. Šablony už pak zobrazují jen připravená data. Uvnitř šablony se zaměřujeme na to, jak budou data prezentována, a mimo šablonu na to jaká data budeme prezentovat.

Pozn.: Tam, kde je v textu této práce použito například: *"Šablony jsou psány ve FreeMarkeru"*, je tím myšleno, že jsou napsány právě v jazyce *FTL*.

Na obrázku 31 je vidět jakým princip jakým způsobem *FreeMarker* funguje. Na vstupu dostane připravená data a šablonu a výstupem je charater stream uložený do objektu třídy *java.io.StringWriter*. Nad tímto objektem jen zavoláme metodu *toString()* a dostaneme obsah generovaného souboru. Toto se děje uvnitř třídy *TemplateProcessor*.



Obrázek 31 Případ užití FreeMarkeru (z[12])

FreeMarker je šablonovací nástroj.

Pro jeho použití v Maven projektu je nutné přidat tut dependency do pom.xml:

```
<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
</dependency>
```

Na příkladu HelloWorld na obrázku 32 si ukážeme, jak se píší funkce ve FreeMarkeru. Funkce se uvozuje klíčovým slovem *function*, za kterým následuje jméno funkce a za ním můžou následovat parametry funkce. Každá funkce musí obsahovat *return*, kterým definujeme návratovou hodnotu funkce. Uvnitř funkce lze používat lokální proměnné uvozené klíčovým slovem *local*.

Volání funkce závisí na tom, kde ji voláme. Volání funkce uvnitř další funkce je vidět v definici funkce *concatDemo*, kde voláme funkci *sayHello*() nebo funkci *sayWorld()*.

Funkce *concatDemo* mimo jiné ukazuje, jak funguje spojování řetězců uložených v lokálních proměnných. Pokud potřebujeme použít výstup funkce v šabloně, použijeme volání ${concatDemo()}, tím do šablony dostaneme hodnotu, kterou funkce vrací ve formě řetězce.

```
1  <#function concatDemo>
2   <#local str = "">
3   <#local str += sayHello()>
4   <#local str += " ">
5   <#local str += sayWorld()>
6   <#local str += "!">
7   <#return str>
8  </#function>
9
10 <#function sayHello><#return "Hello"></#function>
11 <#function sayWorld><#return "World"></#function>
12
13 ${concatDemo()}
```

Obrázek 32 Hello world ve FreeMarkeru

Výstup volání funkce z obrázku 32 bude: Hello world!

6.2.2. **Návrh šablon v pluginu**

Šablony jsou v pluginu umístěné v adresáři:

```
/src/main/resources/templates
```

V tomto adresáři jsou rozděleny do tří dalších složek:

- java – Obsahuje šablony pro action beany.
- jsf – Obsahuje šablony pro JSF(Primefaces stránky).
- lib – Obsahuje podpůrné funkce ve FreeMarkeru, které jsem si napsal, aby bylo možné generovat komponenty z datových objektů, které můžou být zanořeny v rekurzivní struktuře. Tyto objekty byly popsány v kapitole 6.1.2.

V adresáři java se nacházejí šablony *detail-action.ftl, edit-action.ftl, list-action.ftl* a *view-bean.ftl*. Pakliže bude ve vstupním xml souboru nastaveno *entityName* jako *Funkce* (myšleno funkce pracovníka ve firmě), budou z tichto šablon generovány soubory: *FunkceDetailAction.java, FunkceEditAction.java, FunkceListAction.java FunkceViewBean.java.*

V adresáři jsf se nacházejí šablony *edit.ftl, detail.ftl, list.ftl, template.ftl* a jsou z nich generovány zdrojové soubory JSF(Primefaces) stránek: *funkceDetail.ftl, funkceEdit.ftl, funkceList.ftl, template.ftl*. Šablony *edit.ftl, detail.ftl a list.ftl* využívají knihoven ze třídy lib.

Ve třídě *lib* se nachází tyto soubory s podpůrnými funkcemi, které jsem napsal ve Fremarkeru:

- *accordion.ftl,*
- *commandButton.ftl,*
- *commandLink.ftl,*
- *containerComponent.ftl,*
- *dateItem.ftl,*
- *displayItem.ftl,*
- *optionComponent.ftl*
- *selectOneMenu.ftl,*
- *tabComponent.ftl,*
- *tabView.ftl,*
- *textArea.ftl,*
- *textItem.ftl.*

Každý soubor ve složce lib obsahuje jednu funkci napsanou ve FreeMarkeru. Pro lepší přehlednost dodržuji tento soubor pravidel:

- V každém souboru je umístěna pouze jedna funkce.
- Jméno souboru s funkcí je shodné se jménem funkce, začíná malým písmenem a dodržuje *camelCase* notaci.
- Názvy funkcí jsou odvozeny od názvů komponent ze vstupního xml. Funkce *accordion* má tedy na starosti připravení šablony pro komponentu *accordion*, které v sobě může mít zanořeno více *tabů*. Ty jsou řešeny uvnitř jejího těla funkcí *tabComponent*. V dodržování této jmenné konvence jsem udělal zatím dvě výjimky a to pro funkce *tabComponent* a *optionComponent*, protože *option* a *tab*, jsou ve FreeMarkeru již použita.

Podívejme se nyní na obrázek 33 s fragmentem kódu funkce *containerComponent* a na obrázek 31 s fragmentem kódu, jak je tato funkce volána v šabloně *edit.ftl*. Šabloně edit je předána java kolekce *List<Component>components*, kde *Component* je datový objekt popsaný v kapitole 6.1.2. Detaily, jak je toho docíleno jsou popsány u popisu třídy *TemplateProcessor*, v souvislosit s touto kapitolou se spokojíme pouze s tím, že v proměnné *components* je obsah proměnné *componentList* z třídy *EditPageCoponent,* která je potomkem třídy *ContainerComponent* a jsou v ní uchována data o stránce *edit*. V proměnné *components*, jsou tedy všechny kořenové komponenty ze vstupního xml pro stránku *edit* a v některých z nich můžou být zanořené další komponenty, na další úrovni. Máme tedy rekurzivní strukturu, kterou je třeba projít. K tomuto účelu byla napsána funkce *containerComponent* a další podpůrné funkce ve složce lib.V šabloně *edit.ftl*, je procházen seznam všech komponent první úrovně v seznamu *components.* Ve funkci *containerComponent* se na základě atributu *itemType* (popsaného v kapitole 6.2.1) rozhodne, o jakou komponentu se jedná a ta je pak zpracována funkcí, které tomu přísluší. Z fragmentu kódu je vidět, že například komponenta *displayItem* je zpracována funkcí *displayItem*. Volání funkce *continerComponent* ze šablony je na obrázku 34.

```
1   <#include "displayItem.ftl">
2   <#include "textItem.ftl">
3           ...
4   <#function containerComponent component context>
5       <#local ret ="">
6       <#if component.itemType?? && component.itemType="displayItem">
7           <#local ret = ret + displayItem(component,context) >
8       </#if>
9       <#if component.itemType?? && component.itemType="textItem">
10          <#local ret = ret + textItem(component,context) >
11      </#if>
12          ...
13       <#return ret>
14  </#function>
15
```

Obrázek 33 Funkce containerComponent

```
<#if components??>
    <#list components as component>
        ...
        ${containerComponent(component,"${bean}.detailViewBean.edit.")}
        ...
    </#if>
</#list>
```

Obrázek 34 Volání funkce containerComponent

6.2.3. TemplateProcesor

Tato třída má na starosti vytvořit z dat o uložených v kolekci List<*Artifact*> *baseArtifacts* ve spojení se šablonami výsledný generovaný soubor. Podle atributu *artifactType,* který může nabývat hodnoty JSF nebo JAVA se zavolá příslušná obslužná metoda (generateJsfPage(project, artifact) nebo generateJavaSource(project, artifact).

Uvnitř těchto metod se pak na základě atributu project, který uchovává kontext cílové aplikace, do které jsou zdrojové soubory generovány, vytvoří buď soubor typu java nebo webfacet, ty v sobě mají uloženy informace, kde mají být v adresářové struktuře umístěny. Obsah souboru získá z instance StringWriteru, jak bylo popsáno již v kapitole 6.2.1 popisující FreeMarker.

6.3. Rozšíření pluginu

6.3.1. Obecná pravidla pro přidání komponenty

Pro přidání komponenty je nutné udělat tyto kroky:

- Navrhnout reprezentaci komponenty ve vstupním XML
- Navrhnout datový objekt pro uložení komponenty
- Navrhnout parser pro uložení komponenty
- Navrhnout obslužnou funkci pro použití komponenty v šablonách

6.3.2. **Přidání TabView**

Komponenta *TabView* představuje panel s horizontálními záložkami. Každý *TabView* panel může obsahovat jeden nebo více tabů (záložek) a každá záložka může obsahovat libovolné množství komponent.

Jak bude komponenta reprezentována ve vstupním XML je vidět na obrázku 35.

```
<tabView>
    <tab title="tabView1Tab1">
        <displayItem id="organizace1" label="Organizace1" field="organizace1" />
    </tab>
    <tab title="tabView1Tab2">
        <displayItem id="organizace2" label="Organizace2" field="organizace2" />
    </tab>
</tabView>
```

Obrázek 35 tabView - struktura ve vstupním xml

TabView se skládá ze dvou elementů. Je třeba vytvořit dva datové elementy a dva parsery. Jak budou vypadat, si ukážeme na následujících fragmentech kódů.

Na fragmentech kódu nejsou uvedeny definice getterů a setterů pro atributy datových objektů, které je samozřejmě nutné také vytvořit.

Datový objekt *TabView* bude potomek *Component* (obrázek 36). Bude potřebovat seznam *tabList*, pro uchování *tabů*, které obsahuje a metodu pro přidání nového *tabu*, do seznamu. Seznam *tabList* nesmíme zapomenout inicializovat v konstruktoru.

```
1  public class TabView extends Component {
2
3      /** List of tabs in tabView */
4      protected List<Tab> tabList;
5
6      public TabView(String itemType, String id) {
7          super(itemType, id);
8          tabList = new ArrayList<Tab>();
9      }
10     public void addTab(Tab tab) {
11         this.tabList.add(tab);
12     }
13     //get, set
14 }
```

Obrázek 36 TabView datový objekt

Datový objekt *Tab* (obrázek 37) bude potomek *Container*Component, protože může obsahovat libovolné další komponenty. Od rodičovské třídy zdědí seznam komponent *componentList*. Není třeba jej inicializovat, je již inicializován v rodiči. Už je potřeba přidat jen atribut *title*.

```java
public class Tab extends ContainerComponent {

    /** Title of tab. */
    protected String title;

    public Tab(String itemType, String id) {
        super(itemType, id);
    }

    //get,set
}
```

Obrázek 37 Tab datový objekt

TabView je potomek *Component*, takže *TabViewParser* (obrázek 38) bude potomek *ComponentParser*. Je třeba implementovat metodu *parser*. Jejím vstupem je parsovaný element.

Podle toho, jak nastavíme atribut *itemType*, bude rozlišován *TabView* ve funkci pro generování šablony pro tento element. Funkce *element.hasAttribute* vrátí, jestli má element ve vstupním XML daný atribut, pokud ano, jeho hodnotu získáme funkcí *getAttribute*. Takto získáme všechny atributy elementu.

Protože *tabView* může obsahovat elementy typu *tab*, je potřeba projít v cyklu všechny podřízené elementy.

Podmínkou *childNode.getNodeName() == ParserFactory.TAB_INPUT_XML_NAME,* získáme jen elementy typu *tab*. V *ParserFactory* jsou uchovány konstanty se jmény elementů, které mohou být ve vstupním XML a existuje pro ně parser.

```java
public class TabViewParser extends ComponentParser {

    @Override
    public Component parse(Node node) throws ParserFactoryException {
        final Element element = (Element) node;

        String itemType = "tabView";
        String id = "";
        if (element.hasAttribute("id")) {
            id = element.getAttribute("id");
        }

        TabView tabView = new TabView(itemType,id);
        ParserFactory parserFactory = ParserFactory.getInstance();

        for (int i = 0; i < node.getChildNodes().getLength(); i++) {
            final Node childNode = node.getChildNodes().item(i);
            //tabView can contain only tab elements
            if(childNode.getNodeName() == ParserFactory.TAB_INPUT_XML_NAME) {
                ComponentParser tabParser = parserFactory.getParser(childNode.getNodeName());
                Tab tab = (Tab) tabParser.parse(childNode);
                if(tab != null) {
                    tabView.addTab(tab);
                }
            }
        }
        return tabView;
    }
}
```

Obrázek 38 TabViewParser

Instanci ParserFactory získáme metodou getInstance(). Zavoláním metody *getParser(childNode.getNodeName())* získáme správný parser, pro element, který ze kterého chceme získat data. V tomto případě to může být jen *tabParser.* Metodou *parse()* z něj opět získáme data. Když vše dopadne pořádku a výsledek není *null*, tak daostaneme datový objekt typu *Tab* a přidáme ho do seznamu záložek metodou *addTab()*.

Tab je potomkem *ContainerComponent*, takže *TabParser* bude potomkem *ContainerComponentParser* (viz obrázek 40). Attributy se získají podobně, jako u předchozího parseru a zpracování vnořených komponent, zajistí metoda *parserChildren()*.

```
1  public class TabParser extends ContainerComponentParser {
2      @Override
3      public Component parse(Node node) throws ParserFactoryException {
4
5
6          Tab tab = new Tab("tabComponent", null);
7          //set attributes
8          final Element element = (Element) node;
9          if(element.hasAttribute("id")) {
10             tab.setId(element.getAttribute("id"));
11         }
12         if(element.hasAttribute("title")) {
13             tab.setTitle(element.getAttribute("title"));
14         }
15         //parse child elements
16         parseChildren(node, tab);
17
18         return tab;
19     }
```

Obrázek 39 TabParser

Ještě je třeba upravit *ParserFactory*.(viz obrázek 40). Přidáme do ní dvě konstanty se jmény nových elementů ve vstupním XML. Dále přidáme v konstruktoru *ParserFactory* mapování na nově vytvořené parsery do mapy *parsers*. Toto je vše, co musíme udělat pro získání hodnot nového komponenty.

```
1  public static final String TAB_INPUT_XML_NAME = "tab";
2  public static final String TAB_VIEW_INPUT_XML_NAME = "tabView";
3
4  private ParserFactory() {
5      parsers.put(TAB_VIEW_INPUT_XML_NAME, new TabViewParser());
6      parsers.put(TAB_INPUT_XML_NAME, new TabParser());
7      ...
8  }
```
.

Obrázek 40 Změny v ParserFactory pro přidání tabView

Nyní nám chybí už jen vytvoření šablony pro novou komponentu. Je třeba vytvořit dvě nové funkce ve FreeMarkeru. Butou to *tabView.ftl* a *tabComponent.ftl*.

A do *containerComponent.ftl* přidáme fragment kódu z obrázku 41. Kód nám říká, že pokud komponenta není null a je typu *tabView*, zavolej funkci, která nám vrátí šablonu pro *tabView* a přidá ji do výstupního řetězce funkce *containerComponent*.

```
1  <#include "tabView.ftl">
2  <#function containerComponent component context bean entityName>
3
4    <#if component.itemType?? && component.itemType=="tabView">
5          <#local ret = ret + tabView(component, context, bean, entityName) >
6    </#if>
7          ...
8    <#return ret>
9  </#function>
```

Obrázek 41 Úpravy v containerComponent

Funkce, která generuje komponentu *tabView* je na obrázku 42. V lokálních proměnných uvozených slovem *local* se postupně skládá kód komponenty. Pokud *tabList,seznam* komponent *tabView,* není prázdný, projde se a záložky v něm se zpracují funkcí *tabComponent*. Cyklus je ve Freemarkeru uvozený klíčovým slovem *list* a je to cyklus přes všechny komponenty kolekce.

Za klíčivým slovem *function* pro funkci a *tabView* jménem funkce je několik parametrů. Ty jsou zde uvedeny, protože *tabComponent* může obsahovat libovolnou komponentu a některé z nich tyto proměnné potřebují a není jiný způsob, jak jim hodnoty v nich uložené předat.

```
1  <#include "tabComponent.ftl">
2  <#function tabView component context bean entityName>
3        <#local str1="<p:tabView>\n" >
4        <#local str2="">
5        <#if component.tabList??>
6            <#list component.tabList as tab>
7                <#local str2= str2 + tabComponent(tab, context, bean, entityName)>
8            </#list>
9        </#if>
10       <#local str3 ="</p:tabView>\n" >
11       <#return str1 + str2 + str3>
12 </#function>
```

Obrázek 42 tabView šablona

Funkce, která generuje komponentu *tab* je na obrázku 43. Opět se postupně skládá po částech v lokálních proměnných, které se pak pospojují. Pokud je seznam komponent neprázdný, projde se v cyklu a komponenty v něm se zpracují funkcí *containerComponent*. Ta v sobě totiž zavolá příslušnou funkci na obsloužení vstupující komponenty. Tyto funkce ale musí být definovány v *containerComponent.ftl*, podobně jako funkce *tabView*, jak je vidět na obrázku 41.

```
<#function tabComponent component context bean entityName>
    <#local str1="<p:tab">
    <#if component.title?? >
        <#local str1= str1 + " title=\"" >
        <#local str1 = str1 + component.title >
        <#local str1 = str1 + "\" ">
    </#if>
    <#local str1= str1 + ">\n">
    <#local str2="">
    <#if component.componentList??>
        <#list component.componentList as component>
            <#local str2 = str2+ containerComponent(component, context, bean, entityName) >
        </#list>
    </#if>
    <#local str3="</p:tab>\n">
    <#return str1+str2+str3>
</#function>
```

Obrázek 43 tabComponent šablona

Generovaný výstup pro vstup z obrázku 35, je na obrázku 44:

```
<p:tabView>
    <p:tab title="tabViewTab1" >
      <h:outputText id="organizace1" value="#{funkce_01DetailAction.detailViewBean.view.organizace1}"/>
    </p:tab>
    <p:tab title="tabViewTab1" >
       <h:outputText id="organizace2" value="#{funkce_01DetailAction.detailViewBean.view.organizace2}"/>
    </p:tab>
</p:tabView>
```

Obrázek 44 Generovaný výstup tabView

6.3.3. Přidání komponenty commandLink

Komponenta *commandLink* je hypertextový odkaz, který se chová jako tlačítko. Reprezentace ve vstupním XML je vidět na obrázku 45.

```
<commandLink id="kodFunkce" label="Kód" field="kodFunkce" link="detail"/>
```

Obrázek 45 Příklad commandLinku ve vstupním XML

Klikneme-li na *commandLink,* dostaneme se na stránku definovanou atributem *link*. Bude třeba vytořit jeden datový objekt a jeden parser. Datový objekt bude potomkem *FieldComponent* (z něj zdědí *field* a *label*), zbývá přidat atribut *link* (viz obrázek 46).

```
public class CommandLink extends FieldComponent {
    /**
     * Name of page where command link links.
     * e.g.: detail, edit, list
     */
    protected String link;

    public CommandLink(String itemType, String id, String label, String field) {
        super(itemType, id, label, field);
    }
    //get,set
}
```

Obrázek 46 CommandLink datový objekt

Parser bude potomkem *ComponentParser.* (obrázek 47). Zbytek bude stejný jako u předchozích parserů.

```
public class CommandLinkParser extends ComponentParser {
    @Override
    public Component parse(Node node) throws ParserFactoryException {

        final Element element = (Element) node;

        String itemType = "commandLink";
        String id = "";
        String label = "";
        String field = "";

        if (element.hasAttribute("id")) {
            id = element.getAttribute("id");
        }

        //... nastavení další attributů

        CommandLink commandLink = new CommandLink(itemType,id,label,field);

        if(element.hasAttribute("link")) {
            commandLink.setLink(element.getAttribute("link"));
        }

        return commandLink;
    }
}
```

Obrázek 47 CommandLinkParser

Do *ParserFactory* je třeba přidat novou proměnou a parser (Obrázek 48), princip je stejný jako u tabView.

```
public static final String COMMAND_LINK_INPUT_XML_NAME = "commandLink";
private ParserFactory() {
    parsers.put(COMMAND_LINK_INPUT_XML_NAME, new CommandLinkParser());
    ...
}
```

Obrázek 48 commandLink ParserFactory

Dále je třeba vytvořit funkci ve FreeMarkeru pro generování komponenty *commandLink*. Bude se jmenovat *commandLink.ftl* (viz obrázek 49). Uvnitř funkce se v lokální proměnné postupně poskládá výsledná generovaná komponenta. Konečně zde využijeme množství parametrů funkce zmiňované u tabView, které se využije pro vygenerování volání metody pro přechod na stránku detail. Dále si můžeme všimnout, že některé znaky jako například levá složená závorka je třeba escapovat, jinak mají speciální význam.

Dále můžeme u řetězcových proměnných používat různé funkce pro práci s řetězci.

- proměnná?un_capfirst, zajistí malé první písmenko
- proměnná?cap_first, Zajití velké první písmenko

```
1  <#function commandLink component context bean entityName>
2      <#local str1="<p:commandLink">
3
4      <#local str1 = str1 + " action=\"" >
5      <#local str1 = str1 + "#\{" + bean + ".open"+component.link?cap_first+
6      "Page("+component.field +",\'"+entityName?uncap_first+ component.link?cap_first+"\')}" >
7      <#local str1 = str1 + "\"">
8
9      <#local str1 = str1 + " value=\"" >
10     <#local str1 = str1 + "#\{" + context + component.field + "}" >
11     <#local str1 = str1 + "\"">
12     <#local str1=str1 +" />">
13     <#return  str1 >
14 </#function>
```

Obrázek 49 commandLink.ftl

Co bude nutné přidat do *containerComponent* je na obrázku 50. Princip opět stejný jako u *tabView*. Na obrázku 51 je pak generovaný výstup.

```
1  <#include "commandLink.ftl">
2  <#function containerComponent component context bean entityName>
3      <#local ret ="">
4      <#if component.itemType?? && component.itemType=="commandLink">
5          <#local ret = ret + commandLink(component,context,bean,entityName) >
6      </#if>
7      ...
8   <#return ret>
9  </#function>
10
```

Obrázek 50 úprava containerComponent

```
1  <p:commandLink action="#{funkce_01ListAction.openDetailPage(kodFunkce,'funkce_01Detail')}"
2                 value="#{funkce_01.view.kodFunkce}" />
```

Obrázek 51 Generovaný výstup commandLinku

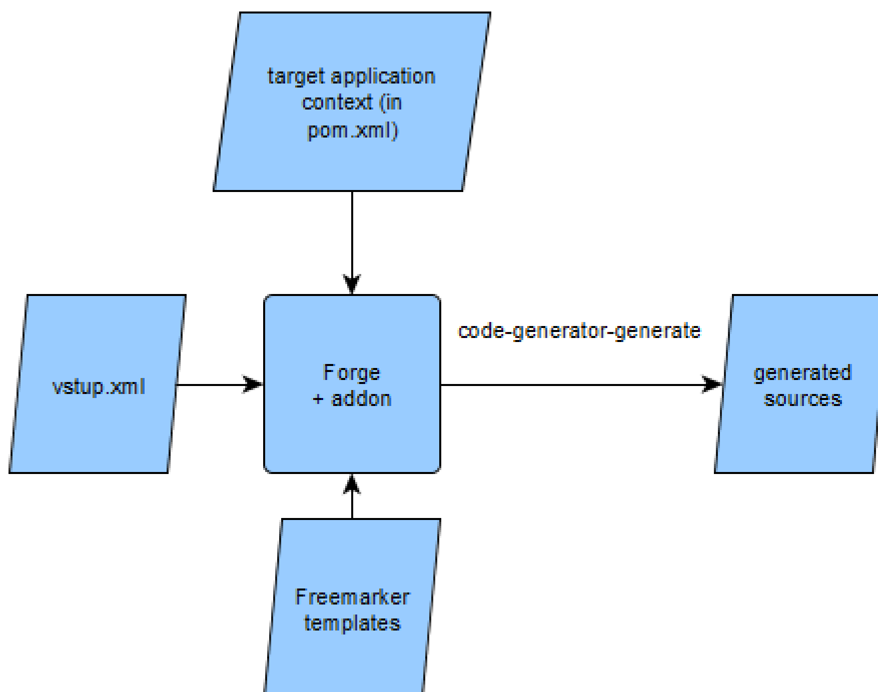
.

7. Ověření funkčnosti generátoru na příkladu

Jedním z cílů této diplomové práce bylo vytvořit generátor, který bude ze vstupního XML souboru generovat zdrojové soubory cílové aplikace. Od zadavatele jsem dostal referenční ručně psanou implementaci projektu nazvaného Ramses akademie. Tuto aplikaci ve firmě CCA používají například pro správu informací o zaměstnancích. Dostal jsem jen výřez této aplikace jen se zdrojovými sobory pro entitu funkce. Konfigurace generátoru – tedy vstupní XML a šablony pro strány Detail, Edit a List, byly navrženy tak, aby byly schopné generovat stránky *Detail, Edit a List* blížící se ručně psaným zdrojovým kódům. Pro testování byla zvolena entita Funkce, ve smyslu pozice pracovníka ve firmě. Proces generování je znázorněn na obrázku 52.

Bylo třeba vygenerovat tyto soubory:

- FunkceDeatailAction.java,
- FunkceEditAction.java
- FunkceListAction.java
- FunkceViewBean.java
- funkceDetail.xhtml
- funkceEdit.xhtml
- funkceList.xhtml
- template.xhtml



Obrázek 52 Proces generování

Provedl jsem dva typy ověření funkčnosti generátoru. První postup byl takový, že jsme se zadavatelem připravili obsah navrženého vstupního XML tak, aby popisoval entitu funkce natolik, aby z něj bylo možné generovat téměř stejný výstup jako ručně psaný kód. Při generování bylo dohodnuto několik drobných zjednodušení. Například v generátoru nebyly řešeny klíčové atributy, obslužné funkce ze souborů funkce*ActionBean.java byly generovány některé celé a některé třeba jen z části s komentářem, že mají být implementovány. Příklad vstupního XML je na obrázku v příloze C. Při porovnání referenčních ručně psaných a generovaných souborů došlo až na domluvené výjimky ke shodě.

Dále bylo provedeno ověřeno generování všech navržených komponent. Byl vytvořen vstup pro demonstraci využít všech komponent (obrázek 53) a k němu výstup generátoru (obrázek 54). Všechny komponenty byly správně vygenerovány.

```
1  <accordion>
2    <tab title="accordionTab1">
3      <displayItem id="kodFunkce" label="Kód" field="kodFunkce"/>
4      <textArea id="znalosti" label="Znalosti" maxlength="4000" rows="5" field="znalosti"/>
5    </tab>
6    <tab title="accordionTab2">
7      <displayItem id="kodFunkce" label="Kód" field="kodFunkce"/>
8      <textArea id="popisFunkce" label="Popis" maxlength="4000" rows="5" field="popisFunkce"/>
9    </tab>
10 </accordion>
11 <tabView>
12     <tab title="tabViewTab">
13         <commandLink id="kodFunkce" label="Kód" field="kodFunkce" link="detail"/>
14         <textItem id="nazevFunkce" label="Název" field="nazevFunkce"/>
15         <dateItem id="platnostOd" label="Platnost od" field="platnostOd" type="date" format="dd.MM.yyyy"/>
16         <dateItem id="platnostDo" label="Platnost do" field="platnostDo" type="date" format="dd.MM.yyyy"/>
17         <selectOneMenu id="pouzitiPovoleno" label="Použití povoleno" field="pouzitiPovoleno">
18             <option value="T">Ano</option>
19             <option value="F">Ne</option>
20         </selectOneMenu>
21     </tab>
22     <tab title="tabViewTab2">
23         <displayItem id="kodFunkce" label="Kód" field="kodFunkce"/>
24         <displayItem id="platnostOd" label="Platnost od" field="platnostOd" type="date" format="dd.MM.yyyy"/>
25         <displayItem id="platnostDo" label="Platnost do" field="platnostDo" type="date" format="dd.MM.yyyy"/>
26         <displayItem id="organizace" label="Organizace" field="organizace"/>
27     </tab>
28 </tabView>
29 <button id="testovaciTlacitko" label="Otestuj tlačítko">
30     <description>Spustí nějakou velice užitenou činnost.</description>
31 </button>
```

Obrázek 53 Vstup

```
1  <p:accordionPanel>
2      <p:tab title="accordionTab1" >
3        <h:outputText id="kodFunkce" value="#{testDetailAction.detailViewBean.view.kodFunkce}"/>
4        <p:inputTextarea id="znalosti" value="#{testDetailAction.detailViewBean.view.znalosti}" rows="5" maxlength="4000"/>
5      </p:tab>
6      <p:tab title="accordionTab2" >
7        <h:outputText id="kodFunkce" value="#{testDetailAction.detailViewBean.view.kodFunkce}"/>
8        <p:inputTextarea id="popisFunkce" value="#{testDetailAction.detailViewBean.view.popisFunkce}" rows="5" maxlength="4000"/>
9      </p:tab>
10 </p:accordionPanel>
11 <p:tabView>
12     <p:tab title="tabViewTab" >
13       <p:commandLink action="#{testDetailAction.openDetailPage(kodFunkce,'testDetail')}"
14                      value="#{testDetailAction.detailViewBean.view.kodFunkce}" />
15       <p:inputText id="nazevFunkce" value="#{testDetailAction.detailViewBean.view.nazevFunkce}"/>
16       <p:calendar id="platnostOd" value="#{testDetailAction.detailViewBean.view.platnostOd} maxlength="30" pattern="dd.MM.yyyy"
17       navigator="true" display="inline"effect="fadeIn" yearRange="1991:2030" required="true"requiredMessage="#{msg.required}" >
18         <f:convertDateTime pattern="dd.MM.yyyy" />
19         <p:ajax event="dateSelect" listener="#{funkceEditAction.validatePlatnostOd}" oncomplete="window.scrollTo(0, 0);" />
20       </p:calendar>
21       <p:calendar id="platnostDo" value="#{testDetailAction.detailViewBean.view.platnostDo} maxlength="30" pattern="dd.MM.yyyy"
22       navigator="true" display="inline"effect="fadeIn" yearRange="1991:2030" required="true"requiredMessage="#{msg.required}" >
23          <f:convertDateTime pattern="dd.MM.yyyy" />
24          <p:ajax event="dateSelect" listener="#{funkceEditAction.validatePlatnostDo}" oncomplete="window.scrollTo(0, 0);" />
25        </p:calendar>
26       <p:selectOneMenu id="pouzitiPovoleno" value="#{testDetailAction.detailViewBean.view.pouzitiPovoleno}" >
27          <f:selectItem itemLabel="Ano" itemValue="T" />
28          <f:selectItem itemLabel="Ne" itemValue="F" />
29        </p:selectOneMenu>
30      </p:tab>
31      <p:tab title="tabViewTab2" >
32         <h:outputText id="kodFunkce" value="#{testDetailAction.detailViewBean.view.kodFunkce}"/>
33         <h:outputText id="platnostOd" value="#{testDetailAction.detailViewBean.view.platnostOd}">
34            <f:convertDateTime pattern="dd.MM.yyyy" />
35         </h:outputText>
36         <h:outputText id="platnostDo" value="#{testDetailAction.detailViewBean.view.platnostDo}">
37            <f:convertDateTime pattern="dd.MM.yyyy" />
38          </h:outputText>
39         <h:outputText id="organizace" value="#{testDetailAction.detailViewBean.view.organizace}"/>
40     </p:tab>
41 </p:tabView>
42 <p:commandButton action="#{testDetailAction.testovaciTlacitko()}" value="Otestuj tlačítko" />
```

Obrázek 54 Generovaný výstup

8. ZÁVĚR

Během řešení diplomové práce jsem se seznámil s problematikou generování zdrojového kódu Java EE aplikací z formalizovaného návrhu a s metodikou CCA pro návrh modulů uživatelského rozhraní. Informace o generování zdrojového kódu jsem čerpal hlavně z knihy Code Generation in action [1] a ze zkušeností pracovníků firmy CCA při konzultacích s panem Ing. Petrem Přibylem.

Prozkoumal jsem dostupné nástroje pro generování kódu a usoudil, že Forge 2, který již dříve ve firmě CCA použili při vývoji první verze generátoru, bude tím pravým nástrojem. Dostal jsem zdrojové kódy, jejichž fragmenty měly být výstupem generátoru, podrobně jsem se seznámil jak s nimi, tak s první verzí implementace generátoru. Navrhl jsem strukturu konfigurace generátoru, kterou je xml soubor, který obsahuje název generované entity a popis komponent, které budou generovány na stránkách detail, edit a list.

Navrhl jsem a implementoval úpravy na generátoru, aby bylo možné generovat i složitější typy komponent. Navrhl jsem rekurzivní strukturu sady objektů, do kterých jsou informace o komponentách parsovány ze vstupního xml souboru. Taktéž jsem navrhl a implementoval sadu parserů, které získání těchto informací zajistily. Dále jsem kompletně přepsal šablony pro generování požadovaných zdrojových souborů a napsal sadu funkcí v jazyku FreeMarker Template Language, které umožňují dynamicky generovat složitější komponenty uložené v rekurzivní struktuře.

Sepsal jsem postup pro přidání nové komponenty a demonstroval jej na několika příkladech požadovaných zadavatelem. Dále jsem nastínil, co by bylo třeba udělat pro přidání další generované stránky. Dále jsem vytvořil uživatelskou příručku pro instalaci a použití generátoru, kterou lze nalézt v příloze A. Tato diplomová práce by měla sloužit jako zdroj poznání pro další rozšiřování generátoru.

Funkčnost generátoru jsem otestoval pro entitu Funkce, kdy je míněna funkce pracovníka ve firmě. Společně s panem Ing. Petrem Přibylem z firmy CCA jsme přizpůsobili obsah komponent ve vstupním xml tak, aby vyhovoval požadavkům na generované stránky. Pustil jsem generátor s tímto vstupním xml souborem a porovnal generované soubory s požadavky zadavatele. Ověřil jsem tímto, že generátor funguje tak, jak má. Tímto jsem splnil všechny body zadání.

Do budoucna navrhuji tyto rozšíření a úpravy:

- Šablony pro generování stránek jsou v současné době součástí addonu. Tento přístup byl navržen v první verzi generátoru, ze které jsem vycházel. Se zadavatelem jsme se dohodli, že tento přístup zatím zachováme, ale při další úpravě generátoru by bylo vhodné se zamyslet, jak by bylo možné šablony oddělit od generátoru.

LITERATURA

[1] HERRINGTON, Jack. *Code generation in action*. Greenwich, CT: Manning, c2003. ISBN 1930110979.

[2] RUBINGER, Andrew Lee. a Aslak. KNUTSEN. *Continuous enterprise development in Java*. ISBN 1449328296.

[3] Enterprise Architect - Návrh aplikace pro VV [CCapedia (lokální wiki firmy CCA)]. In: . [cit. 2016-06-07].

[4] Export data model [CCapedia (lokální wiki firmy CCA)]. In: . [cit. 2015-08-08].

[5] *JBoss Forge* [online]. JBoss Forge, 2016 [cit. 2016-06-20]. Dostupné z: forge.jboss.org

[6] Write a Java EE Web Application - Basic. GONCALVES, Antonio. *JBoss Forge* [online]. JBoss Forge, 2016 [cit. 2016-06-20]. Dostupné z: http://forge.jboss.org/document/write-a-java-ee-web-application---basic

[7] *JBoss Forge* [online]. JBoss Forge [cit. 2016-06-20]. Dostupné z: http://forge.jboss.org/1.x/docs/important_plugins/ui-scaffolding.html

[8] XSLT. *Wikipedia* [online]. [cit. 2016-06-20]. Dostupné z: https://cs.wikipedia.org/wiki/Extensible_Stylesheet_Language_Transformations

[9] Facelets. *Wikipedie* [online]. 2013 [cit. 2016-06-20]. Dostupné z: https://cs.wikipedia.org/wiki/Facelets

[10] AJAX. *Wikipedie* [online]. [cit. 2016-06-20]. Dostupné z: https://cs.wikipedia.org/wiki/AJAX

[11] *PrimeFaces* [online]. [cit. 2016-06-20]. Dostupné z: http://www.primefaces.org/

[12] *FreeMarker* [online]. [cit. 2016-06-20]. Dostupné z: http://freemarker.org/

PŘÍLOHA A – UŽIVATELSKÁ PŘÍRUČKA

Generátor je plugin do Forge 2. Nástroj vyvíjen a testován v prostředí Windows 8.1. Měl by však běžet na všech verzích od Windows 7 a novějších.

V tomto návodu je popsáno, jak si zprovoznit Forge s pluginem a jak jeho funkcionalitu vyzkoušet na projektu CCA ramses-akademie-project.

Pro použití generátoru bude potřeba toto:

- Apache Maven 3.2.3, dostupný na https://maven.apache.org,
- JBoss Forge 2.20.2, dostupný na cd k diplomové práci v archivu:
  forge-distribution-2.20.2.Final-offline.zip, případně na http://forge.jboss.org/
- Java 8

Postup instalace:

1. Zkontrolujte, že máte nainstalovaný Maven 3.2.3 a Javu 8. Pokud ne, nainstalujte si je.
2. Na cd s diplomovou prací je archiv forge-distribution-2.20.2.Final-offline.zip. Tento archiv si rozbalte na pevný disk někam, kde budete mít práva zápisu. Například na cestu:
   C:/devTools/forge/
3. Dále je třeba přidat systémovou proměnnou s umístěním Forge, to se provede takto:
   i. Přidejte si systémovou proměnnou FORGE_HOME, která obsahuje cestu k adresáři s rozbaleným Forgem.
   ii. Do proměnné PATH(path) přidejte *%FORGE_HOME%/bin.*
4. instalaci ověřte spuštěním příkazu ***forge*** z konzole.
5. Na cd s diplomovou prací je složka s maven projektem generátoru, jmenuje se code-generator-addon, její obsah zkopírujte například do
   C:/devTools/code-generator-addon
6. V kořenové složce generátoru, tedy na cestě z bodu 4, zavolejte příkaz:
   mvn clean install
   Tím se nainstaluje addon do lokálního maven repositáře.
7. Příkazem *forge --install cz.cca.forge:code-generator,2.5.1-SNAPSHOT* nainstalujete addon z lokálního repozitáře do forge. 2.5.1-SNAPSHOT verzi addonu, která je uvedena hlavním *pom.xml* addonu.
8. Nyní je ve forge dostupný nový příkaz *code-generator-generate*. Před spuštěním generátoru je nutné si někam zkopírovat testovací projekt, který je taktéž na cd s diplomovou prací. Zkopírujte jej např. na umístění C:/Ramses/ramses-akademie-project.
9. Spuštění generátoru:

10. Ve složce *C:*/Ramses/ramses-akademie-project/ramses-akademie-war/ spusťte příkazovou řádku. (Např. z Total Commanderu zavoláním cmd). POZOR: v názvu složek nesmí být mezera.
11. Do složky ramses-akademie-war si umístěte vsupní soubor generátoru, vstup*.xml*
12. V příkazové řádce zadejte příkaz *forge,* kterým pustíte textovou konzoli Forge 2.
13. V konzoli Forge 2 zavolejte příkaz:
    code-generator-generate --inputType XML --inputParametersFile vstup.xml

Pozn.: *code-generator-generate* je příkaz, který pouští generátor, *inputType* je parametr, který říká, že vstup bude ve formátu *XML*. V současné době je implementován pouze vstup v *XML*. Attribut *inputParametersFile* uvádí cestu ke vstupnímu souboru. Zde se jedná o relativní cestu, takže soubor *vstup_generatoru.xml* musí být umístěn ve složce *war*, ve které jsme spouštěli Forge. Na názvu vstupního souboru nezáleží, jen takový soubor musíme do složky war nakopírovat. Rozlišení generovaných objektů pak závisí na atributu *entityName* uvnitř tohoto souboru.

Na následujícím obrázku 55 je vidět úspěšné spuštění Forge a generátoru. Generované soubory naleznete ve složce v modulu *war*, ve kterém byl příkaz puštěn.

```
C:\Windows\System32\cmd.exe - forge
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. Všechna práva vyhrazena.

c:\Ramses\ramses-akademie-project\ramses-akademie-project\ramses-akademie-war>forge
Using Forge at c:\devTools\tools\forge-distribution-2.20.2.Final

JBoss Forge, version [ 2.20.2.Final ] - JBoss, by Red Hat, Inc. [ http://forge.jboss.org ]

[ramses-akademie-war]$ code-generator-generate --inputType XML --inputParametersFile vstup.xml
[ramses-akademie-war]$
```

Obrázek 55 Spuštění Forge

PŘÍLOHA B – Export z EA, XSL soubor a výstup XSTL transformace

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <?xml-stylesheet type="text/xsl" href="EA_JavaCodeGeneration.xsl"?>
3  <page
4      code="TST004F"
5      title="Běh skupiny testů">
6
7      <dataBlock
8          label="Vnořené skupiny testů"
9          entity="CcagPodrizeneTestPripady"
0          condition="kde ID_SKUPINY_TESTU_NAD = :Skupina.ID_SKUPINY_TESTU"
1          validation=""
2          order="podle KOD vzestupně">
3          <notes>Skupiny testů, které jsou podřízené skupině vybrané v bloku Skupina.</notes>
4
5          <attributes>
6              <attribute label="Poř." field="poradoveCislo" ... >
7                  <notes>Pořadové číslo vnořené skupiny testů.</notes>
8                  <tooltip>Pořadové číslo testovacího případu ve skupině testů</tooltip>
9              </attribute>
0              <attribute label="Kód"field="kod" type="String" ... >
1                  <notes>Kód podřízené skupiny testů ... </notes>
2                  <tooltip>Kód podřízené skupiny testů</tooltip>
3              </attribute>
4          </attributes>
5          <!-- element list generovan dle atributu SCN: Dotaz?
6              u datoveho bloku -->
7          <list header="Výpis vnořených skupin testů">
8              <attributeToShow field="poradoveCislo" />
9              <attributeToShow field="kod" />
0          </list>
1
2          <detail>
3              <attributeToShow field="kod" />
4          </detail>
5
6          <!-- element edit generovan pokud alespon jednu z vlastnosti
7              (u datoveho bloku ) create, update, delete nastavenou na true -->
8          <edit
9              create="false (SCN: Vkládání?)"
0              update="true (SCN: Změna?)"
1              delete="true (SCN: Mazání?)">
2              <!-- atribut generovan pokud ma alespon jednu z vlastnosti
3                  (u atributu) create, update nastavenou na true -->
4              <attributeToShow
5                  field="poradoveCislo"
6                  label="Pořadí" />
7              <attributeToShow field="kod" />
8          </edit>
9      </dataBlock>
0  </page>
```

Obrázek 56 EA export.xml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3     <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
4
5     <xsl:template match="/">
6       <xsl:element name="generator-unit">
7         <xsl:apply-templates select="page/dataBlock[1]" />
8         <xsl:element name="detail"><xsl:apply-templates select="page/dataBlock[1]/attributes" mode="detail" />
9         </xsl:element>
10        <xsl:element name="edit"><xsl:apply-templates select="page/dataBlock[1]/attributes/attribute[@create='true' or update='true']" mode="edit" />
11        </xsl:element>
12        <xsl:element name="list"><xsl:apply-templates select="page/dataBlock[1]/attributes/attribute[@read='true']" mode="list" />
13        </xsl:element>
14      </xsl:element>
15    </xsl:template>
16
17    <!-- entity name -->
18    <xsl:template match="page/dataBlock[1]">
19      <xsl:element name="name" >
20        <xsl:value-of select="@entity"/>
21      </xsl:element>
22    </xsl:template>
23
24    <!-- blok detail - spracovano for-eachem -->
25    <xsl:template match="page/dataBlock[1]/attributes" mode="detail">
26      <xsl:for-each select="attribute[@read='true']">
27        <xsl:element name="view" >
28          <xsl:attribute name="id">
29            <xsl:value-of select="@field" />
30          </xsl:attribute>
31          <xsl:attribute name="label">
32            <xsl:value-of select="@label" />
33          </xsl:attribute>
34          <xsl:attribute name="field">
35            <xsl:value-of select="@field" />
36          </xsl:attribute>
37        </xsl:element>
38      </xsl:for-each>
39    </xsl:template>
40
41    <!-- blok edit -->
42    <xsl:template match="page/dataBlock[1]/attributes/attribute[@create='true' or update='true']" mode="edit">
43      <xsl:element name="edit" >
44        <xsl:attribute name="id">
45          <xsl:value-of select="@field" />
46        </xsl:attribute>
47        <xsl:attribute name="label">
48          <xsl:value-of select="@label" />
49        </xsl:attribute>
50        <xsl:attribute name="field">
51          <xsl:value-of select="@field" />
52        </xsl:attribute>
53      </xsl:element>
54    </xsl:template>
55
56    <!-- blok list -->
57    <xsl:template match="page/dataBlock[1]/attributes/attribute[@read='true']" mode="list">
58      <xsl:element name="column" >
59        <xsl:attribute name="id">
60          <xsl:value-of select="@field" />
61        </xsl:attribute>
62        <xsl:attribute name="header">
63          <xsl:value-of select="@label" />
64        </xsl:attribute>
65        <xsl:attribute name="field">
66          <xsl:value-of select="@field" />
67        </xsl:attribute>
68      </xsl:element>
69    </xsl:template>
70  </xsl:transform>
71
```

Obrázek 57 Příklad XSLsouboru pro transformaci

```
<?xml version="1.0" encoding="UTF-8"?>
<generator-unit>
<name>CcagMilnik</name>
<detail>
<view id="kod" label="Kód" field="kod"/>
<view id="nazev" label="" field="nazev"/>
<view id="system" label="Systém" field="system"/>
</detail>
<edit>
<edit id="kod" label="Kód" field="kod"/>
<edit id="nazev" label="" field="nazev"/>
<edit id="system" label="Systém" field="system"/>
</edit>
<list>
<column id="kod" header="Kód" field="kod"/>
<column id="nazev" header="" field="nazev"/>
<column id="system" header="Systém" field="system"/>
</list>
</generator-unit>
```

Obrázek 58 Výsledek XSLT transformace targe.xml

PŘÍLOHA C – Příklad vstupního XML generátoru

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<generator-unit>
  <entityName>Funkce</entityName>
  <pageTitle>Funkce pracovníka</pageTitle>
  <pageTemplate>/templates/template.xhtml</pageTemplate>
  <dateTimePattern>dd. MM. yyyy</dateTimePattern>
  <detail>
    <displayItem id="kodFunkce" label="Kód" field="kodFunkce"/>
    <displayItem id="nazevFunkce" label="Název" field="nazevFunkce"/>
    <displayItem id="organizace" label="Organizace" field="organizace"/>
    <displayItem id="platnostOd" label="Platnost od" field="platnostOd" type="date" format="dd.MM.yyyy"/>
    <displayItem id="platnostDo" label="Platnost do" field="platnostDo" type="date" format="dd.MM.yyyy"/>
    <displayItem id="popisFunkce" label="Popis" field="popisFunkce"/>
    <displayItem id="znalosti" label="Znalosti" field="znalosti"/>
    <displayItem id="odpovednosti" label="Odpovědnosti" field="odpovednosti"/>
    <displayItem id="pouzitiPovoleno" label="Použití povoleno" field="pouzitiPovoleno"/>
    <button id="prirazeniDovednostiButton" label="Přiřazení dovedností">
      <description>Otevře okno pro výběr dovedností, která mají být funkci přiřazeny</description>
    </button>
  </detail>

  <edit>
    <textItem id="kodFunkce" label="Kód" field="kodFunkce" link="detail" readonly="true"/>
    <textItem id="nazevFunkce" label="Název" field="nazevFunkce"/>
    <textItem id="organizace" label="Organizace" field="organizace"/>
    <dateItem id="platnostOd" label="Platnost od" field="platnostOd" type="date" format="dd.MM.yyyy"/>
    <dateItem id="platnostDo" label="Platnost do" field="platnostDo" type="date" format="dd.MM.yyyy"/>
    <textArea id="popisFunkce" label="Popis" maxlength="4000" rows="5" field="popisFunkce"/>
    <textArea id="znalosti" label="Znalosti" maxlength="4000" rows="5" field="znalosti"/>
    <textArea id="odpovednosti" label="Odpovědnosti" maxlength="4000" rows="5" field="odpovednosti"/>
    <selectOneMenu id="pouzitiPovoleno" label="Použití povoleno" field="pouzitiPovoleno">
      <option value="T">Ano</option>
      <option value="F">Ne</option>
    </selectOneMenu>
  </edit>

  <list selectionMode="multiple" header="Customers">
   <filter>
      <textItem id="filterKodFunkce" label="Kód funkce" field="kodFunkce" link="detail"/>
      <textItem id="filterNazevFunkce" label="Název funkce" field="nazevFunkce"/>
      <textItem id="filterPopisFunkce" label="Popis funkce" field="popisFunkce"/>
      <dateItem id="filterPlatnostOd" label="Platnost od" field="platnostOd" format="dd.MM.yyyy"/>
      <dateItem id="filterPlatnostDo" label="Platnost do" field="platnostDo" format="dd.MM.yyyy"/>
    </filter>
    <dataTable>
      <commandLink id="kodFunkce" label="Kód" field="kodFunkce" link="detail"/>
      <displayItem id="nazevFunkce" label="Název" field="nazevFunkce"/>
      <displayItem id="popisFunkce" label="Popis" field="popisFunkce"/>
      <displayItem id="platnostOd" label="Platnost od" field="platnostOd" type="date" format="dd.MM.yyyy"/>
      <displayItem id="platnostDo" label="Platnost do" field="platnostDo" type="date" format="dd.MM.yyyy"/>
      <displayItem id="organizace" label="Organizace" field="organizace"/>
    </dataTable>
    <button id="deleteSelected" label="Smazat">
      <description>Odstraní vybrané záznamy</description>
    </button>
  </list>
</generator-unit>
```

Obrázek 59 Vstupní XML generátoru