Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Zvýšení uživatelského komfortu při práci s mimofunkčními charakteristikami

Plzeň 2012 Daniel Kuneš

Prohlášení

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

V Plzni dne 16. května 2012

Daniel Kuneš

Abstract

This master's thesis deals with the possibilities of improvement of user comfort during the work with extra-functional properties. The purpose of this work is to design and implement user-friendly interface for comparator tool used for comparing features of components by their extra-functional properties. As part of the presented solution is also web portal application which integrates said interface and allows for easy extensibility for future use in extra-functional properties project. Results of presented solution are demostrated and verified. This work also mentions the principles of component-based application design and subject of extra-functional properties.

Obsah

Seznam obrázků

Poděkování

Rád bych poděkoval Ing. Kamilovi Ježkovi za příkladné vedení mé práce, hodnotné rady a čas věnovaný konzultacím.

1 Úvod

S rozvojem informačních technologiích a především s jejich rozšiřováním do všech oblastí lidské činnosti, dochází ke zvyšování složitosti používaných aplikací. Toto působí zásadní problémy při jejich návrhu, testování a dlouhodobé údržbě. Přestože vývoj programovacích jazyků problém složitosti zmírňuje přesouváním jeho řešení na vyšší úroveň abstrakce, nestačí tento vývoj držet krok se stále rostoucími nároky. Jedním z dalších nástrojů, jak zvýšit udržitelnost a přehlednost aplikací, je jejich rozdělení na samostatné funkční části – komponenty, které nemají tak těsné vazby se svým okolím, další možnou výhodou je pak jejich znovupoužitelnost.

Se vzrůstajícím množstvím vývojářů využívajících tohoto postupu při návrhu aplikací, vzrůstá zároveň i dostupné množství těchto komponent. Z toho důvodu vznikla potřeba tyto komponenty hodnotit nejenom na základě jimi poskytovaných funkcí, ale i z hlediska jejich běhových parametrů (například paměťová náročnost, vytížení procesoru, úroveň zabezpečení přenosu dat apod.). Těmto běhovým parametrům se souhrnně říká mimofunkční charakteristiky. Touto problematikou se zaobírá 3. kapitola. Katedra informatiky a výpočetní techniky Západočeské univerzity se zabývá výzkumem oblasti mimofunkčních charakteristik v rámci projektu EFP.

Účelem této diplomové práce je zvýšit uživatelský komfort při práci s mimofunkčními charakteristikami v tomto projektu. Zabývá se návrhem a implementací uživatelského rozhraní aplikace pro porovnávání mimofunkčních charakteristik, jež bude integrováno v taktéž navrhovaném webovém portálu, jenž si jako budoucí cíl klade včlenění všech aplikací z EFP projektu. Takto navržený celek bude sloužit vědeckým pracovníkům při výzkumu v oblasti komponentového vývoje aplikací.

Na začátku této práce (kapitola 2) bude čtenář uveden do teorie vývoje softwaru založeného na komponentových technologiích. Dále bude seznámen s oblastí mimofunkčních charakteristik a projektem EFP (kapitola 3). Práce se dále zabývá výběrem vhodných technologií (kapitola 4) pro implementaci zmiňovaného webového portálu a uživatelského rozhraní aplikace pro porovnávání mimofunkčních charakteristik. Ke konci práce popisuje vzniklou implementaci a ověření funkčnosti (kapitola 6).

Práce přináší novou rozšiřitelnou webovou aplikaci, sloužící pro integraci nástrojů pro práci a analýzu mimofunkčních charakteristik. Zároveň přináší konkrétní nástroj pro vizualizaci výsledků aplikace *EFP Comparator*, který výrazně zpřehledňuje formu podávaných informací a prohlubuje jejich detailnost.

2 Komponentové technologie

Vývoj softwaru založeného na komponentových technologiích (dále jen CBD[1]) vzešel z myšlenky znovupoužitelnosti softwaru. Důvodem byly problémy zapříčiněné zvýšenými náklady na vývoj, provoz a údržbu systémů. Tyto systémy byly vyvíjeny úplně od začátku tedy bez žádných předchozích základů, ke kterým by bylo možné přidávat další funkcionalitu a zároveň udržovat tu stávající.

Naproti tomu software založený od základů na komponentových technologiích umožňuje snadno a rychle vyvíjet novou funkcionalitu skládáním komponent vytvořených třetími stranami, tudíž není nutné psát své vlastní moduly resp. komponenty, které zajišťují stejnou funkcionalitu. Jak uvádí Vojtěch Liška ve své diplomové práci [1], CBD přináší při vývoji softwaru výhody jako jsou nezávislá rozšíření, snížený čas dodání systému na trh a znovupoužitelnost komponent v dalších systémech.

Při tvorbě systémů založených na CBD se využívá návrhu strategie komponentově orientovaných systémů. Návrhem strategie se myslí něco jako návrhový vzor pro komponentově orientované systémy.

2.1 Komponenta

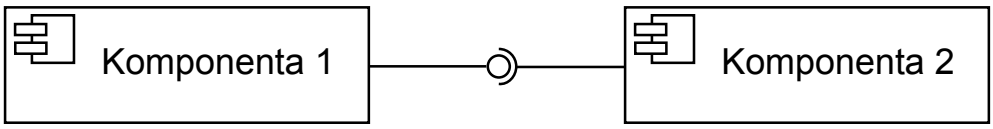
Jak je uvedeno výše, tak komponenty v CBD jsou funkční celky, které je možné skládat do větších celků. Princip komponent a jejich skládání se zdá jednoduchý a přesto nemá jednoznačnou definici(možná víc definic). Jednou z nabízených definic pro komponentu je definice Clemense Szyperského [3]. Tato definice říká:

A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Komponenty sdružují podobnou funkcionalitu, jejíž implementace by neměla být k dispozici (black-box model). Viditelné by mělo být pouze rozhraní, přes které komponenty komunikují s prostředím. Aby bylo možné komponenty přes rozhraní propojit musí každá komponenta obsahovat seznam funkčností, které poskytuje (imported/required) nebo vyžaduje (exported, provided). V případě, že komponenta od prostředí neobdrží požadovanou funkcionality nebude fungovat. Implementace komponenty nesmí být závislá na svém okolí

[1]Component Based Development

a měla by být verzována. Na obrázku 2.1 je uvedena ukázka grafického zobrazení spojení dvou komponent.



Obrázek 2.1: Ukázka grafického zobrazení komponent a jejich spojení

2.2 Komponentový model a framework

V předchozí části této práce byla věnována pozornost komponentovým technologiím a komponentám. Nicméně, aby komponenty mohly fungovat, musí být umístěny do komponentového modelu. Zdroj [3] uvádí, že model pro komponenty definuje striktně pravidla komunikace mezi nimi. Dále definuje, jak se komponenty skládají, povolené služby a také samotnou strukturu komponenty.

Tento model je tvořen množinou komponentových typů, jejich rozhraními a navíc specifikacemi povolených vzorů interakce mezi jednotlivými komponentovými typy. Tyto vlastnosti jsou popsány v práci [2].

Komponentových modelů existuje více, například OSGi, CoSi, EJB3, SOFA2, atd. V této práci se budeme podrobněji zabývat pouze modely OSGi a CoSi, které jsou využívány v projektu mimofunkčních charakteristik.

Komponentový model definuje pravidla pro komponenty a jejich kontext. Komponenty se tedy vytvářejí podle těchto pravidel a pro jejich běh je nutné mít odpovídající prostředí. Toto prostředí se nazývá komponentový framework, který je implementací komponentového modelu. V práci [2] je komponentový framework přirovnáván k operačnímu systému, který pracuje s komponentami jako operační systém s procesy.

2.2.1 OSGi platforma

OSGi[2] Alliance je nezávislá organizace, která vytváří otevřené specifikace pro oblast distribuovaných prostředí. OSGi platforma je těmito specifikacemi definována a skládá se z definice OSGi frameworku (dále jen OSGi) a množiny dalších služeb. Také poskytuje prostředí pro běh komponentových aplikací. Běhové prostředí pro OSGi je JVM, která

[2]Původně označována jako *Open Services Gateway initiative*

zahrnuje širokou škálu zařízení (modemy, mobilní telefony, spotřební elektroniku, osobní počítače, atd.).

OSGi se rozděluje na 4 základní vrstvy [4]:

- služby,
- životní cyklus,
- moduly,
- běhové prostředí.

Životní cyklus komponent je uveden v [4]. Volitelnou vrstvou je zabezpečení. OSGi má několik implementací, z nichž nejznámější jsou: Apache Felix[3], Eclipse Equinox[4] a Knopflerfish[5].

OSGi Bundle

Komponenta, která byla výše popsaná a definovaná, je v OSGi označována jako bundle. Jak je uvedeno v práci [5], tak se bundle sestává z vlastní implementace java tříd a dalších pomocných zdrojů, které společně poskytují danou funkcionalitu pro své klienty.

Bundly mohou poskytovat nebo vyžadovat javovské balíky (PACKAGE), což umožňuje přistupovat ke konkrétním třídám poskytovaného balíku jiným bundlem. Prostředí OSGi pracuje se službami, které má centrální registr těchto služeb. Nasazované bundly si do něj můžou zaregistrovat poskytující služby a zároveň od něj chtít ty, jež od prostředí vyžadují.

Implementace bundlu je jar soubor, který obsahuje zmíněné java třídy, pomocné zdroje jako jsou například ikony, soubor manifest a také může obsahovat java dokumentaci.

Manifest je textový soubor s příponou mf nacházející se v adresáři META-INF, který popisuje strukturu bundlu a je frameworkem vyžadován kvůli jeho instalaci. Při nasazení bundlu OSGi propojí všechny vyžadované funkcionality se správnými poskytovateli a ostatním nabídne poskytované funkcionality tohoto bundlu. Na následující ukázce je vidět struktura manifestu.

[3] felix.apache.org
[4] www.eclipse.org/equinox
[5] www.knopflerfish.org

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: cz.zcu.kiv.example.bundle
Bundle-Version: 1.0.0
Bundle-Name: Ukázkový bundle
Bundle-Activator: cz.zcu.kiv.example.bundle.Activator
Export-Package: cz.zcu.kiv.example.bundle.package;version="1.2.0"
Import-Package: cz.zcu.kiv.example.bundle2.package;version="1.4.0"
```

- Bundle-ManifestVersion – Verze OSGi hlaviček v manifestu.
- Bundle-SymbolicName – Jedinečný identifikátor, který by měl dodržovat konvenci pro pojmenovávání balíčků v javě.
- Bundle-Version – Verze bundlu ve formátu číslo. číslo. číslo
- Bundle-Name – Výstižné pojmenování bundlu.
- Bundle-Activator – Třída, která se stará o aktivaci a zastavení bundlu
- Export-Package – Poskytované balíky.
- Import-Package – Vyžadované balíky.

2.2.2 CoSi platforma

CoSi[6] [6] je komponentový framework založený na Javě, který je vyvíjen na Katedře informatiky a výpočetní techniky Západočeské univerzity. Tento framework je inspirován platformou OSGi z níž některé vlastnosti přebírá, vypouští nebo rozšiřuje. CoSi komponenty se zasazují do běhového prostředí, kterému CoSi říká kontejner.

Základní motivací je čistý black-box. Vše, co komponenta poskytuje je definováno mimo její implementaci, z čehož vyplývá, že ostatní komponenty v kontejneru nemohou přistupovat k ničemu jinému, než je definováno.

Specifikace CoSi se skládá z popisu:

- **komponent** – Definuje, co je komponenta, co obsahuje, jak psát komponenty a jak komponenty mezi sebou komunikují.

[6] Components Simplified

- **kontejneru** – Kontejner je běhové prostředí pro komponenty. Životní cyklus komponenty se odehrává uvnitř kontejneru. Skrze tento kontejner jsou dostupná všechna data o komponentách a je možné s jeho pomocí získat odkazy na služby.
- **základních služeb** – Kontejner poskytuje dvě základní služby pro komponenty. První z nich poskytuje jednoduché aplikační rozhraní (API) kontejneru a druhá reprezentuje komunikační službu založenou na událostech.

CoSi bundle

V terminologii CoSi jsou komponenty nazývány bundle stejně jako je tomu v případě OSGi. Každý bundle je jednoznačně identifikován jménem, verzí a svým poskytovatelem. Struktura CoSi bundlu je obdobná jako u OSGi bundlu včetně jeho distribuce v archivu jar.

CoSi bundly mohou poskytovat ostatním bundlům nějaké funkce nebo naopak mohou od jiných bundlů nějaké funkce vyžadovat. Tyto funkce jsou označovány jako vlastnosti (features), které mají své jméno a typ. Vlastnosti se stejným typem jsou rozlišeny jménem.

Model CoSi definuje čtyři základní druhy vlastností:

- **Služba** (Service) je implementace funkcionality, která je specifikovaná rozhraním v jazyce Java.
- **Typy** (Types) se vztahuje k třídě nebo rozhraní jazyka.
- **Události** (Events) umožňují zasílaní zpráv mezi komponentami zprostředkované kontejnerem.
- **Attributy** (Attributes) jsou hodnoty, které komponenta může zapisovat i číst a jsou přístupné přes registr atributů kontejneru.

3 Projekt mimofunkčních charakteristik

V této kapitole budou popsány mimofunkční charakteristiky, kterými se zabývá projekt vyvíjený na Katedře informatiky a výpočetní techniky Západočeské univerzity. Tento projekt bude blíže představen, z důvodu jeho následného rozšíření v rámci této práce o webový portál, do kterého budou postupně přidávány samostatné aplikace z téhož projektu.

3.1 Mimofunkční charakteristiky

Pojem mimofunkčních charakteristik pochází z termínu Extra-Functional Properties zkráceně EFP. Podle práce [7] jsou pro EFP často využívána synonyma "kvalitativní atributy nebo faktory". Definice mimofunkčních charakteristik z této práce říká, že se jedná o speciální druh informací sloužících k vyjádření poskytovaných a vyžadovaných vlastností na dané části softwaru.

3.1.1 Reprezentace EFP

EFP je možné reprezentovat několika jazyky například CQML [18], NoFun [19] atd. Ale v této práci bude využívána reprezentace EFP, která je vyvíjena na Katedře informatiky a výpočetní techniky Západočeské univerzity.

Jak bylo uvedeno v předchozí kapitole 2, tak komponenta může poskytovat nebo vyžadovat různou funkcionalitu, dále jen feature. Pro tyto featury může být specifikována jedna nebo více EFP, které ji blíže popisují. Mimofunkční charakteristikou může být například průměrná odezva, požadavek na paměť nebo podpora mezinárodních měn. Z čehož vyplývá, že každá EFP má své jméno, typ hodnoty, hodnotu, porovnávací funkce a meta informace.

Typ hodnoty udává jakého datového typu bude hodnota EFP. Typy mohou být real, integer, boolean, enum, set, ratio a string. Hodnota nemusí být uvedena, ale pokud je v EFP obsažena, tak musí odpovídat danému typu. Složené EFP navíc mají logickou formuli. Tato formule definuje vztah k jiným hodnotám charakteristik, ze kterých je složená

EFP sestavena. Pokud je logická formule vyhodnocena pravdivě, tak je přiřazená hodnota validní.

Porovnávací funkce neboli gamma, která porovnává dvě vlastnosti přiřazené k EFP a rozhodne, která z nich má lepší hodnotu. Meta informace jsou doplňující informace, které mají význam ve vztahu k EFP. V současnosti meta informace obsahují string s měrnou jednotkou a výčet jmen s možnými hodnotami dané vlastnosti, jež je možné použít při ukládání do lokálních registrů. Formální zápis EFP je uveden v dokumentu [8].

Mimofunkční charakteristiky jsou rozděleny na *SimpleEFP* a *DerivedEFP*, které jsou podrobně popsány v dokumentech [7, 9]. *DerivedEFP* se skládá z jednoduchých a jiných složených EFP.

EFP s hodnotami jsou ukládány do registrů. V případě projektu mimofunkčních charakteristik jsou používané dva druhy registrů (globální a lokální) [8].

Globální registr

Globální registr (GR) je úložiště, kde jsou uloženy seznamy definic jednoduchých a složených EFP. Obsahuje záznamy s unikátním identifikátorem, jménem úložiště a množinou EFP. Je třeba zdůraznit, že Globální Registr neobsahuje konkrétní hodnoty EFP, protože ty se můžou široce lišit v závislosti na prostředích. Z čehož vyplývá, že jsou kontextově nezávislé. Globální registr je validní pro všechna prostředí, ve kterých jsou EFP definované.

Lokální registr

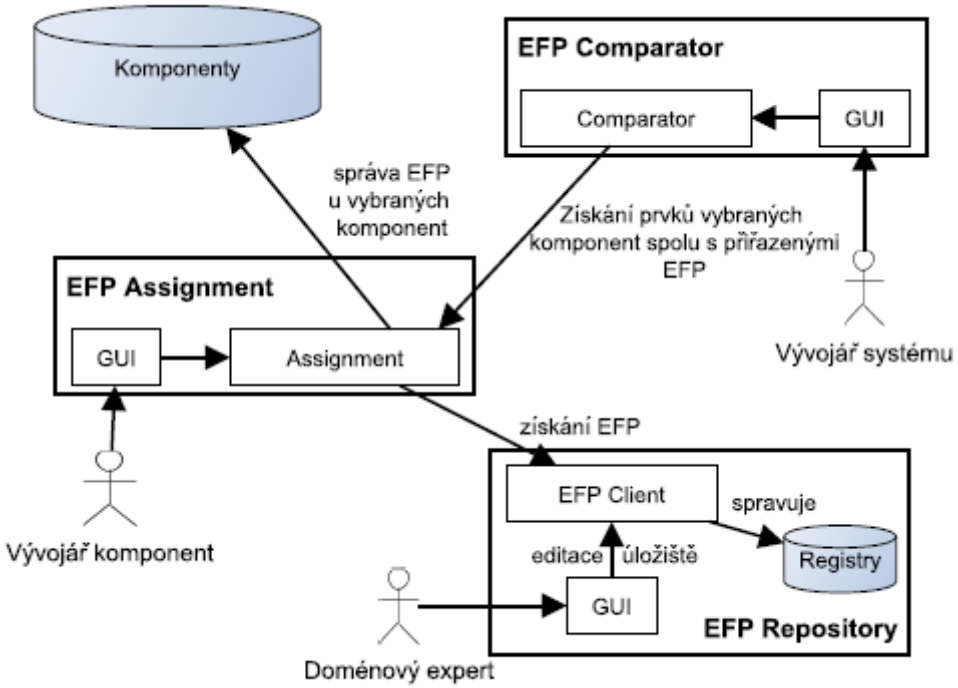
Lokální registr (LR) je úložiště hodnot, které jsou přiřazené ke konkrétní EFP. Každé prostředí definuje vlastní LR s konkrétními hodnotami vlastností definovaných v GR.

Každý lokální registr obsahuje unikátní identifikátor, odkaz na globální registr, se kterým je LR svázán, a jméno úložiště čitelné uživatelem. Dále může mít lokální registr přiřazený identifikátor nadřazeného LR, což umožňuje vytvářet stromové hierarchie LR. Hodnota z nadřazeného prvku je zděděna pouze v případě, že ji dané LR nepřepisuje.

3.2 Náhled na projekt mimofunkčních charakteristik

Projekt mimofunkčních charakteristik umožňuje deklarovat EFP, uchovávat jejich definice a hodnoty v repositáři, přiřazovat je konkrétním komponentám a vyhodnocovat

jejich kompatibilitu [8]. Struktura projektu je znázorněna na obrázku 3.1.



Obrázek 3.1: Schéma celého projektu (převzato z [10])

Nejprve doménový expert navrhne mimofunkční charakteristiky, jež budou použity u celé řady komponent a aplikací pomocí EFP Repositáře. Repositář uchovává navržené definice a je k němu přistupováno ostatními moduly, které chtějí získat, vytvořit nebo modifikovat vlastnosti uložených definic. Poté vývojář komponent prostřednictvím EFP Assignmentu přiřazuje konkrétní vlastnosti a hodnoty jeho komponentám.

Při sestavování komponentové aplikace vybere vývojář systému komponenty a ověří jejich vzájemnou kompatibilitu na základě přiřazených EFP. K ověření kompatibility slouží EFP Comparator. Z kompatibilních komponent následně sestaví výslednou aplikaci.

V následující části budou blíže popsány jednotlivé aplikace EFP Repository, EFP Assignment a EFP Comparator z EFP projektu.

3.2.1 EFP Repository

EFP repositář je jedna z aplikací EFP projektu, který má na starosti správu úložiště mimofunkčních charakteristik. Do úložiště je přistupováno prostřednictvím objektu EFP

Client. Tento klient může definovat a editovat globální registry, spravovat lokální registry a všechny EFP uložené v globálních registrech.

Struktura repositáře sestává ze čtyř částí *efpTypes*, *efpRegistryGUI*, *efpRegistryClient* a *efpRegistryServer*. První část *efpTypes* zahrnuje definice základních entit a to lokálního a globálního registru, mimofunkční charakteristiky a datové typy hodnot EFP. Také obsahuje třídy sloužící k vyhodnocování logických formulí, serializaci a přiřazování hodnot k EFP.

Část *efpRegistryGUI* obsahuje grafické uživatelské prostředí, které umožňuje interakci mezi úložištěm a uživatelem. Posledními částmi jsou *efpRegistryClient* a *efpRegistryServer*, které zajišťují komunikaci mezi klientem, který pro zobrazování dat využívá *efpRegistryGUI*, a serverem. Uživatel prostřednictvím grafického prostředí předá požadavek *efpRegistryClient* a ten jej následně přeposílá *efpRegistryServer*. EFP Repository je detailně popsáno v bakalářské práci [12].

3.2.2 EFP Assignment

EFP Assignment má za úkol umožnit uživateli prostřednictvím grafického uživatelského rozhraní přiřazovat mimofunkční charakteristiky nebo modifikovat již existující přiřazení.

Tato aplikace umožňuje uživateli vybrat typ komponent, se kterými chce pracovat. V tomto případě nabízí k výběru typ komponent OSGi nebo CoSi. Komponenty vybraného typu jsou načteny z úložiště, které může být realizováno například adresářem. Poté uživatel k těmto komponentám přiřazuje EFP. K těmto EFP se také přiřazuje jejich hodnota, která může být jedním z několika druhů:

- přímá hodnota,
- prázdná hodnota,
- hodnota z lokálního registru patřící k EFP,
- matematická formule.

Hodnoty přiřazené k EFP lze také pomocí EFP Assignmentu modifikovat. Upravené komponenty jsou následně uloženy do stejného adresáře, ve kterém se nacházely před jejich načtením. Podrobnější informace o struktuře této aplikace jsou v bakalářské práci [10].

3.2.3 EFP Comparator

Comparator byl rovněž jako předchozí aplikace vytvořen v rámci bakalářské práce [11]. V kapitole 2 bylo zmíněno, že komponenty jsou spojovány přes vlastnosti, které vyžadují a poskytují. Cílem Comparatoru je porovnávat komponenty na základě jejich vlastností (dále jen feature) a samozřejmě také EFP přiřazených k těmto featurám.

Výstupem této aplikace je pouze textový výpis s informacemi o kompatibilitě či nekompatibilitě porovnávaných komponent.U nekompatibilních komponent uvádí proč tyto komponenty nemohou být vzájemně propojeny. EFP Comparator bude v této části popsán podrobněji, neboť jedním z cílů této práce je pro něj vytvořit grafické uživatelské rozhraní.

Princip

Porovnávání komponent EFP Comparatorem prochází několika základními fázemi. Nejprve musí dojít k načtení dat, která budou komparátorem porovnávána. Při načítání dat se využívá modul *efpAssignment*, jež přes své rozhraní poskytuje vstupní data nezávislá na vybraném frameworku.

Načtená data jsou následně uložena do struktury realizující graf, který je procházen do hloubky. Při tomto průchodu dochází zároveň k vyhodnocování a ukládání výsledků do seznamu s položkami typu **EfpEvalResult**. Poslední fází je vypsání těchto výsledků uživateli do konzole. Výstupy z EFP Comparatoru jsou vidět na následující ukázce.

```
Components: ..\osgi\InventoryDatabase-1.0-SNAPSHOT.jar and
            ..\osgi\InventoryData-1.0-SNAPSHOT.jar

   Feature: PACKAGE.cz.zcu.kiv.osgi.example.
            inventory.inventorydatabase.jdbc,0.0.0

     EFPs express_mode_supported Evaluation: OK
        Value on InventoryDatabase-1.0-SNAPSHOT.jar: true,
        Value on InventoryData-1.0-SNAPSHOT.jar: true
```

Struktura jednoho záznamu výsledku obsahuje následující informace:

- jména dvou porovnávaných komponent,
- objekt Feature,

- objekt EFP,
- výsledek porovnání
- hodnoty k přiřazeným EFP,
- strana.

Každý výsledek nese informaci o dvou porovnaných komponentách. Jestliže jsou komponenty spojeny přes danou feature, tak jedna z komponent tuto featuru poskytuje a druhá ji vyžaduje. Tato informace je uložena v objektu *Feature*. Tento objekt obsahuje jméno featury, identifikátor (celé symbolické jméno), typ (udává, zda se jedná o PACKAGE, EVENT, TYPE atd. viz kapitola 2), verzi a stranu.

Tato strana se vztahuje vždy k prvnímu jménu komponenty uložené v záznamu výsledku a říká, zda tuto featuru první komponenta vyžaduje nebo poskytuje. Z toho vyplývá, že vztah featury s druhou komponentou je vždy obráceně, než je uvedeno v položce strana.

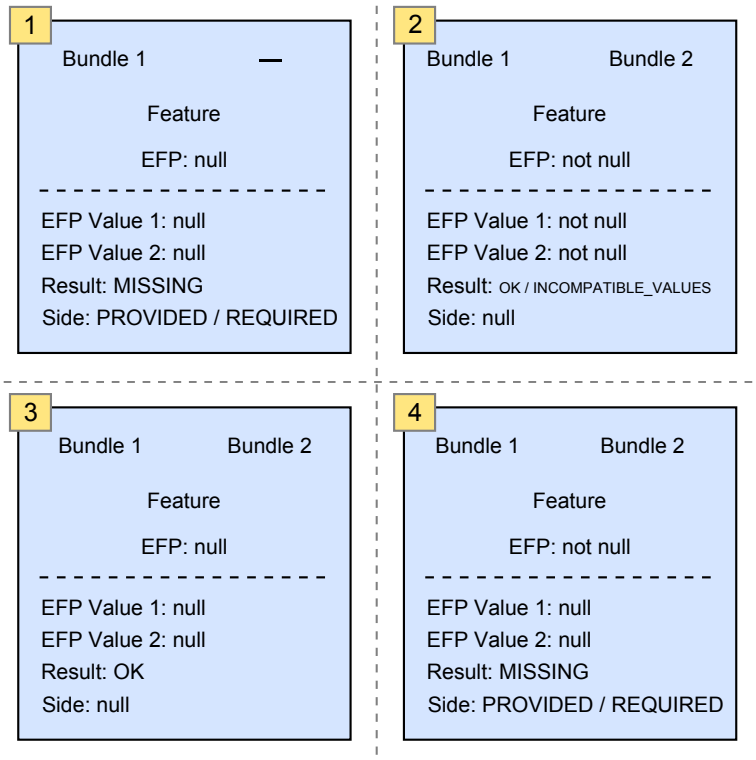
Objekt *EFP* reprezentuje mimofunkční charakteristiku, které je navázána na danou featuru. Zahrnuje své jméno a typ (jednoduchá nebo složená viz výše). V záznamu výsledku je také výsledek porovnání nabývající třech hodnot:

- **OK** – porovnání přes featuru nebo EFP proběhlo v pořádku,
- **INCOMPATIBLE_VALUES** – hodnoty EFP nejsou kompatibilní,
- **MISSING** – featura nebo EFP chybí u jedné z porovnávaných komponent.

Dále hodnoty obou EFP a strana, která doplňuje stav MISSING o to, na které straně featura či EFP chybí.

Z uvedené bakalářské práce [11] vyplývá, že záznamy výsledků **EfpEvalResult** mohou nabývat čtyř stavů s různě vyplněnými informacemi. Tyto stavy jsou důležité pro tuto práci, neboť je potřeba dané výsledky zpracovat a přehledně zobrazit. Jednotlivé stavy jsou zachyceny na obrázku 3.2.

První stav říká, že porovnání komponent neproběhlo v pořádku, protože neexistuje druhá komponenta, která by měla stejnou featuru. Druhý stav ukazuje výsledek porovnání v případě, že na featuru na obou stranách bylo navázáno EFP. Třetí stav reprezentuje pouze spojení komponent na základě featury, protože na žádné z nich není navázáno EFP. Poslední stav zachycuje, že EFP je navázáno na featuru pouze u jedné z komponent.



Obrázek 3.2: Stavy jednotlivých výsledků

Použití EFP Comparatoru

EFP Comparator ke svému běhu vyžaduje dva povinné parametry a jeden volitelný. Prvním parametrem je cesta k adresáři, kde jsou uložené komponenty, které mají být porovnány. Dalším parametrem je ID lokálního registru. Při spuštění bez posledního volitelného parametru se standardně porovnávají komponenty vytvořené pro framework CoSi. Volitelný parametr je řetězec „osgi“, který řekne Comparatoru, že má porovnávat komponenty z OSGi frameworku.

4 Výběr webových technologií

V předchozí kapitole byl představen projekt řešící mimofunkční charakteristiky. Tento projekt je tvořen několika moduly a samostatnými aplikacemi, které tyto moduly ke své činnosti využívají. Konkrétně se jedná o aplikace *EFP Repository*, *EFP Assignment* a *EFP Comparator*, jejichž účel a vzájemná spolupráce je popsána a zobrazena na obrázku uvedeném v podkapitole č. 3.2.

Tyto aplikace vznikaly v rámci bakalářských prací, které jsou uvedeny u jejich jednotlivých popisů v podkapitole 3.2. Pro aplikace *EFP Repository* a *EFP Assignment* bylo v rámci těchto prací vytvořeno grafické uživatelské rozhraní, které bylo pro tyto aplikace nezbytné. Pro poslední aplikaci *EFP Comparator* grafické uživatelské prostředí vytvořeno nebylo a jeho ovládání spočívá v zadání všech potřebných parametrů při spouštění aplikace. Výsledky *EFP Comparatoru* jsou prezentovány ve formě výpisu do konzole.

Cílem této práce bylo zvýšit uživatelský komfort pro práci s mimofunkčními charakteristikami. Z tohoto důvodu bylo zvoleno vytvoření grafického uživatelského rozhraní pro aplikaci *EFP Comparator* kvůli jeho nedostatečně komfortnímu ovládání a také prezentaci jeho výsledků porovnávání, které nemají tak dobrou vypovídací hodnotu, jakou by mohly mít při jejich lepším podání.

Nejprve bylo potřeba navrhnout grafické uživatelské rozhraní a formu prezentace výsledků, které budou mít lepší vypovídající hodnotu než je tomu u současného stavu *EFP Comparatoru*. Návrhem zmíněného se zabývá kapitola 5.

Při navrhování grafického uživatelského rozhraní byly brány v potaz dvě možnosti implementace daného rozhraní. Jednou z nich bylo rozšířit *EFP Comparator* o rozhraní, které by bylo součástí desktopové aplikace *EFP Comparatoru* a druhým možným řešením bylo vytvořit webovou aplikaci, která by rozšířila *EFP Comparator* o dané grafické rozhraní.

Z těchto dvou možností byla vybrána druhá možnost a to navrhnout *EFP Comparator* jako webovou aplikaci. Rozhodujícím kritériem zvolené možnosti byla snadná přístupnost aplikace *EFP Comparatoru* odkudkoli a kdykoli. Výhodou tohoto návrhu by byla jediná běžící verze aplikace. To znamená, že není nutné vytvářet žádný distribuční kanál pro šíření novějších verzí *EFP Comparatoru*. Toto řešení přináší také výhodu odstranění nutnosti stahovat a instalovat klientskou aplikaci.

Na základě vybraného řešení, vytvořit *EFP Comparator* jako webovou aplikaci, vznikla myšlenka s možností vytvoření webového *EFP Portálu*, kam by mohly být in-

tegrovány všechny dosavadní aplikace z EFP projektu. Případně by mohly být přidány další aplikace, které by rozšiřovaly EFP projekt.

Výhody plynoucí z vytvoření EFP Portálu jsou zřejmé – všechny aplikace z EFP projektu na jednom místě přístupném odkudkoli. S vytvořením *EFP Portálu* plyne realizace jeho návrhu, který je uveden ve stejné kapitole, jako je tomu v případě návrhu grafického rozhraní *EFP Comparatoru*, tedy v kapitole 5.

Tato kapitola dále popisuje výběr vhodných webových technologií, které povedou k realizaci *EFP Portálu* a grafického rozhraní *EFP Comparatoru*. Vhodnou webovou technologii je potřeba určit, jak na straně serveru, tak i na straně klienta. Důvod mít část aplikační logiky i na straně klienta vyplývá z principu technického řešení *EFP Comparatoru*, který je uveden v kapitole 6. Výběr technologií je tedy rozdělen do dvou hlavních částí:

- výběr technologií na straně serveru,
- výběr technologií na straně klienta.

4.1 Serverová část

V této části budou popsána kritéria výběru a samotný výběr vyhovující technologie, která byla použita na straně serveru. Následně bude vybraná technologie podrobněji popsána.

4.1.1 Kritéria výběru

Jedním z hlavních kritérií, které je potřeba brát v potaz je, že EFP projekt je celý vyvíjený v jazyce Java a je sestavován pomocí buildovacího nástroje Maven. Aplikace integrované do *EFP Portálu* potřebují interagovat s moduly z EFP projektu a také je požadavkem, aby *EFP Portál* byl integrován do hierarchie EFP projektu. Z toho plyne potřeba využít pouze takové technologie, které jsou založeny na platformě Java konkrétně J2EE[1].

Následujícím kritériem je zajištění snadné rozšiřitelnosti webové aplikace *EFP Portálu* o další integrované aplikace. Z tohoto důvodu je také nezbytné zajistit lepší orientaci v kódu celého *EFP Portálu* pro budoucí potřeby začleňování projektů. Jednou z možností, jak toho docílit, je při výběru technologie zohlednit podporu třívrstvé architektury.

[1] Java 2 Enterpise Edition – nadstavba nad Java 2 Standard Edition pro rozsáhlé vícevrstvé aplikace

4.1.2 Výběr technologie

Na *back-endovou* technologii se nejvíce hodí použití *Java Servlets* nebo *Spring MVC* podle kritéria výběru, které se týká platformy Java. Pro snadnou rozšiřitelnost, udržovatelnost a zajištění přehlednosti pro budoucí projekty integrované do *EFP Portálu* bylo vhodné zvolit použití *Spring MVC*. Výběr tohoto frameworku dále podpořil i fakt, že je tento framework již v projektu EFP použit, tudíž bylo logickým krokem ho vybrat, aby v projektu EFP nebylo zbytečné využití více různých technologií.

4.1.3 Spring MVC

Spring je modulární J2EE opensource aplikační framework [13], který se také označuje jako odlehčený (lightweight) kontejner. Převážně je používán k vytváření webových aplikací, ale je možné jej využít i pro jiné typy. Hlavní myšlenkou *Springu* je usnadnit vývoj Java aplikací. A to hned v několika základních bodech:

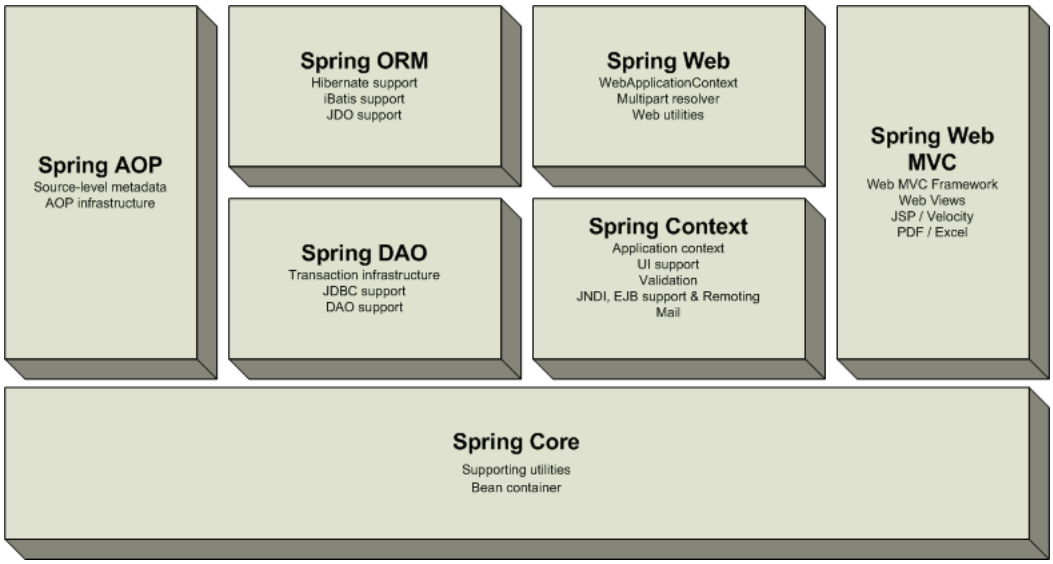
- odstraňování těsných vazeb mezi jednotlivými POJO[2] objekty za pomoci návrhového vzoru *Dependency Injection*,
- správa a konfigurační management business komponent,
- odstranění závislosti na roztroušených konfiguracích a pracného dohledávání jejich významu.
- podpora pro přístup k datům, buď formou přímého JDBC[3] či ORM[4] technologií nebo nástrojů jako Hibernate,
- možnost volby implementace business vrstvy pro aplikační architekturu.

Struktura frameworku Spring sestává ze sedmi základních modulů [14], které jsou uvedeny na obrázku 4.1.

Core modul Tento modul je základem celého frameworku a zajišťuje řízení práce celého kontejneru pomocí *Dependency Injection*. O životní cyklus jednotlivých POJO se stará implementace BeanFactory.

Context modul Context modul zobecňuje práci s BeanFactory. Hraje roli prostředníka mezi klientským kódem a BeanFactory.

[2] *Plain Old Java Objects* – Old neznamená v tomto případě starý, ale pouze označuje klasickou Java třídu.
[3] Java Database Connectivity
[4] Object-Relation Mapping



Obrázek 4.1: Základní moduly frameworku Spring (převzato z http://static.springsource.org)

DAO modul Modul DAO[5] poskytuje abstraktní vrstvu pro práci s JDBC, která odstraňuje potřebu opakujícího se kódu. Také umožňuje deklarativní transakce pro všechny POJO.

ORM modul Poskytuje integrační vrstvy pro ORM (například Hibernate). Při použití ORM můžeme využívat všechny vlastnosti poskytované Springem.

AOP modul AOP modul přidává Springu podporu aspektově orientovaného programování. Dále umožňuje oddělovat části kódu prolínající se celou aplikací do takzvaných aspektů. Tyto aspekty lze následně aplikovat na jakýkoli POJO. AOP se používá v celém Springu a je jedna z jeho nejsilnějších vlastností.

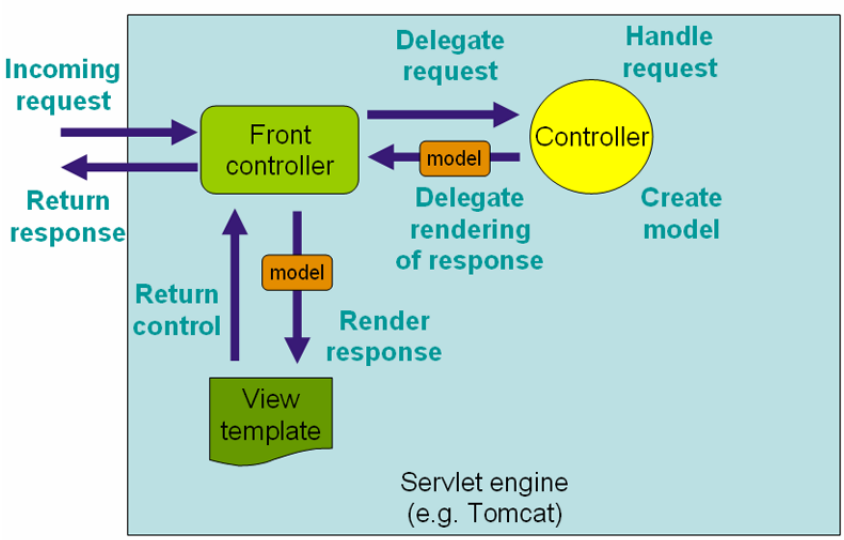
Web modul Přidává podporu základních webově orientovaných vlastností, jako jsou správa uploadovaných souborů, internacionalizace nebo práce s cookies. Dále také poskytuje podporu integrace s webovými frameworky.

Web MVC Poskytuje architekturu MVC pro implementaci webových aplikací. Opět umožňuje využít všech vlastností poskytovaných Springem jako například validaci.

Spring MVC je částí jádra Spring frameworku, který je ustálený a schopný reagovat na akce požadavek-odpověď ve stylu webového frameworku s širokými možnostmi a nastaveními, jež se zaměřuje na webovou vrstvu, jak aplikační, tak i uživatelské rozhraní. Spring MVC zahrnuje většinu stejných základních konceptů jako ostatní MVC frameworky.

[5]Data Access Object

Princip reakce na požadavek je znázorněn na obrázku 4.2. Příchozí požadavky vstupují do frameworku přes *Front Controller*. Tento *Controller* je Java Servlet, kterého Spring MVC nazývá *DispatcherServlet*. Nevykonává žádnou bussiness logiku, ale pouze slouží jako prostředník mezi příchozími požadavky a Java třídami POJO nazývané *Controllery*, kde se vykonává skutečná práce. Po vykonání této práce *Controller* vytvoří model s odpovědí, který předá *DispatcherServletu*. Model je dále delegován na vrstvu *View template*. Tato vrstva vytvoří odpověď pro klienta pomocí JSP stránek. Odpověď může být také ve formátu *JSON*, který je popsán níže. Odpověď je *DispatcherServletem* předána klientovi.



Obrázek 4.2: Schéma komunikace Spring MVC (převzato z http://www.springsource.org)

V následujícím textu budou zmíněny nejzajímavější části či rozšíření frameworku Spring MVC, které byly využity při implementaci *EFP Portalu*.

Tiles2

Tiles2 [15] je framework pro tvorbu šablon, které slouží ke zjednodušení vývoje uživatelského rozhraní webových aplikací. Není přímo součástí frameworku Spring a musí být explicitně připojen.

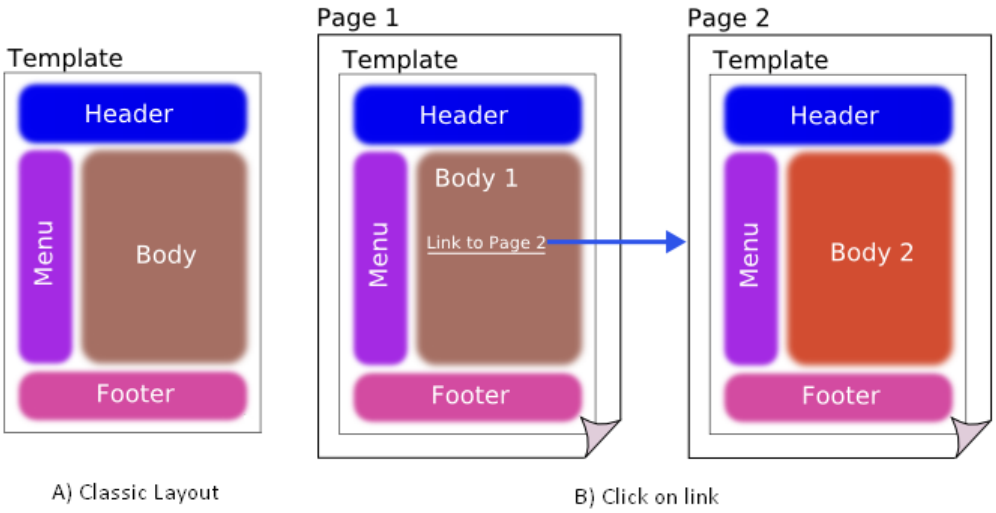
Tento framework umožňuje ve stránce definovat části, které budou použity při sestavování celé stránky za běhu. Tyto části neboli *tiles* mohou být použity jako jednoduché *includes* pro jiné stránky. Tento princip slouží k odstranění duplicitních prvků stránek (menu, hlavička, patička atd.).

Všechny webové stránky mají podobnou strukturu a jednotlivé prvky v nich obsažené sdílejí stejné rozložení. Některé z prvků mohou mít však rozdílný obsah, ale jejich umístění

je vždy stejné.

Composite View je vzor, který formalizuje toto typické použití a umožňuje vytvářet stránky s podobnými strukturami, ve kterých se jednotlivé prvky mění podle situace. Tiles2 je implementací tohoto vzoru.

Na obrázku 4.3 část A je vidět struktura, která se nazývá „Classic Layout“. Šablona (template) organizuje stránky podle tohoto layoutu. Každý prvek stránky je umístěn na příslušné místo, které je definováno „Classic Layoutem“ tak, že hlavička stránky je umístěna v horní části *Header* atd. Využití tohoto layoutu je na obrázku v části B.



Obrázek 4.3: Princip Tiles2 (převzato z http://tiles.apache.org)

Princip změny jednotlivých prvků je zobrazen na témže obrázku 4.3 v části B. Po kliknutí na odkaz na stránce Page 1 v části *Body 1* dojde k přechodu na stránku *Page 2*, kde je změněn pouze obsah části *Body*.

Anotace

Ve Springu se veškerá nastavení provádí v konfiguračních souborech xml. Od verze Spring 2.0 byly zavedeny anotace, které umožňují některá nastavení vytvářet přímo v kódu. S příchodem Springu verze 2.5 a vyšší byla podpora anotací rozšířena.

Díky zavedení anotací je nyní možné provádět konfiguraci, jak v kódu, tak i v konfiguračních souborech a je na uváženou, která z těchto možností bude použita. Vhodné použití je anotování třídy *Controlleru*, které mnohonásobně zlepšuje přehlednost a zjednodušuje jeho tvorbu. Dalšími příklady použití může být konfigurace mapování url na akce

(metody) *Controlleru* nebo v případě získávání parametrů požadavku pro danou akci.

Přestože Spring podporuje anotace, tak musí být explicitně zavedeno jejich použití v konfiguraci, jinak je nelze standardně využívat. Tato konfigurace a použití anotací je popsáno v kapitole 6.

Validace

Narozdíl od použití *Tiles2* zmíněného výše je validace Springem přímo podporována a slouží k ověřování vstupních dat od klienta. Je možné ověřovat neočekávané hodnoty, rozsahy platných hodnot, datové typy, prázdné vstupy, atd. Implementace a veškeré konfigurace spojené s validací jsou uvedeny v kapitole 6.

4.2 Klientská část

Část *EFP Comparatoru* bude realizována na straně klienta pomocí skriptovacího jazyka JavaScript. Pro usnadnění práce se využívají JavaScriptové frameworky, které byly v této práci použity. Těchto frameworků je velké množství a pro tuto práci bylo potřeba vybrat ten nejvhodnější. Tato podkapitola se tedy zaměřuje na jeho výběr a popis ostatních použitých technologií na straně klienta včetně důvodu jejich použití.

4.2.1 Kritéria výběru

Práce [16] se zabývá analýzou webových technologií. Mezi těmito technologiemi je i vybraná množina týkající se JavaScriptových frameworků, která je vybrána z velkého množství podle šíře poskytovaných možností jejich využití. Konkrétně se zabývá těmito frameworky:

- ExtJS4,
- jQuery,
- Dojo Toolkit,
- SmartClient,
- Vaadin,
- OpenLacslo.

Práce sleduje celou škálu kritérií těchto frameworků, ale pro tuto práci jsou důležité pouze některé z nich. Důležitými sledovanými kritérii je kvalita zpracování dokumentace, podpora pro vývojáře (jak začít, příklady a jejich interaktivní ukázky), podpora prohlížečů a množství pluginů.

4.2.2 Výběr technologie

Zmíněná práce [16] uvádí velké množství kritérií hodnocení, které jsou zobrazeny v tabulce. Tato tabulka zde není uvedena z důvodu příliš velkého rozsahu. Při výběru byla zohledněna pouze ta kritéria, která jsou uvedena výše. Pro tato kritéria se všechny uváděné JavaScriptové frameworky ve své podstatě příliš neliší. Jediným podstatnějším rozdílem u frameworku jQuery je široká podpora prohlížečů včetně jejich starších verzí a zároveň je pro něj dostupné velké množství pluginů. Na základě těchto faktů bylo vybráno jQuery. V této práci byly dále použity některé pluginy, které tento framework rozšiřují.

4.2.3 jQuery

JQuery je aktuálně nejpoužívanější JavaScriptový framework [17], který se rozděluje na čtyři základní části jQuery, jQueryUI, jQueryMobile a QUnit. JQuery klade důraz na jednoduchost, čitelnost a rychlost, dále zajišťuje základní funkcionalitu. Zjednodušuje práci s procházením DOMu HTML dokumentu, zpracováním událostí, animacemi, efekty a „AJAXem“, čímž zajišťuje rychlost vývoje.

Pro jQuery je k dispozici velké množství pluginů, které doplňují jeho funkcionalitu. Tento framework se neustále vyvíjí, což je mimo jiné vidět i při vývoji této práce, protože na začátku práce byl k dispozici s verzí 1.6.4 a aktuální verze při psaní této části je 1.7.1. V současné době je podporován firmou Microsoft, z čehož vyplývá ještě větší rozvoj tohoto frameworku a podpory pro vývojáře. JQuery je distribuován ve dvou verzích. První z nich je určená pro vývoj, jejíž velikost činí 247KB a druhá verze je určená pro produkci s kompaktní velikostí 32KB.

4.2.4 HTML5

Na klientské straně byla použita platforma HTML5, protože s sebou přináší mnoho výhod, které umožňují snazší vývoj. HTML5 nabízí nové sémantické značky, které usnadňují orientaci v již existujícím kódu, a proto dochází ke snížení náročnosti budoucích rozšíření či modifikací.

Stěžejním důvodem pro využití HTML5 byla jeho schopnost umožnit nahrávání velkého množství souborů na server zároveň. Toto je důležité z hlediska uživatelské přívětivosti aplikace *EFP Comparatoru*, neboť by uživatel byl nucen všechny soubory nahrávat jednotlivě. Tuto novou funkcionalitu by jinak bylo možno suplovat s použitím JavaScriptu. Jistým problémem ale je závislost této funkcionality na podpoře HTML5 internetovým prohlížečem.

Dalším rozhodujícím prvkem pro využití dané platformy je rozšíření standardu o atribut *data-*, který slouží pro uložení metadat. Tato vlastnost byla využita k uložení informací o umístění jednotlivých prvků v rámci hierarchie zobrazených výsledků. Bližší popis je uveden v následujících kapitolách.

V této platformě je vylepšená podpora zpracování formulářů, kde je možno určit pomocí jediného parametru, že bude pole formuláře povinné. V předchozích verzích HTML bylo toto nutné řešit samostatně pomocí JavaScriptu.

4.2.5 CSS3 a LESS

Tato práce využívá CSS3 generované pomocí LESS preprocesoru. Výhoda této kombinace je možnost parametrizovat CSS při zachování velmi podobného formátu zdrojového kódu obyčejného CSS, aniž by byla ovlivněna kompatibilita.

LESS rozšiřuje základní CSS o dynamické chování jako například použití proměnných, operací a funkcí. Především umožňuje vkládat CSS do elementů pomocí šablon (*mixins*). LESS výrazně zpřehledňuje kód CSS a usnadňuje rozšiřitelnost pro budoucí práce na *EFP Portálu*.

Záměrem využití CSS3 bylo zefektivnění vývoje grafické prezentace *EFP Portálu* a zlepšení uživatelské přívětivosti. Nové vlastnosti CSS3 umožnily úplně se oprostit od grafických editorů. Design *EFP Portálu* je tedy kompletně tvořen CSS prvky, jehož ukázka je zobrazena dále v této práci. Nejdůležitější prvky, které výrazně zjednodušily tvorbu grafického prostředí *EFP Portálu* jsou nové typy selektorů.

4.2.6 AJAX a JSON

Technologie AJAX umožňuje načítat části webových stránek bez nutnosti jejich znovu načtení nebo pouze data ze serveru, což zlepšuje odezvu aplikace na akci uživatele. Jako formát načítaných dat je vhodné používat JSON, který svou strukturou vychází z JavaScriptu a je nativně podporován jak Springem tak jQuery.

5 Návrh webové aplikace EFP Portálu

Tato kapitola se zabývá návrhem celého *EFP Portálu*, přičemž bere v potaz požadavek na možné další integrace aplikací rozšiřující projekt EFP, u kterých lze očekávat výrazné zlepšení přístupnosti, pokud by k nim byl v budoucnu umožněn přístup přes webové rozhraní.

Cílem této práce je zvýšit uživatelský komfort pro práci s mimofunkčními charakteristikami. Po zvážení vícero možností dosažení tohoto cíle bylo navrženo grafické uživatelské rozhraní *EFP Comparatoru* a jeho následná integrace do *EFP Portálu*. Tímto krokem byla otevřena možnost pro další budoucí rozšíření dostupnosti součástí EFP projektu.

Dále se kapitola zabývá návrhem zmíněného rozhraní *EFP Comparatoru*. Ke zvýšení komfortu je potřeba navrhnout takové uživatelské rozhraní, aby bylo jeho použití snadné a intuitivní. Největší požadavek je však kladen na prezentaci výsledků porovnání, které v současném stavu *EFP Comparatoru* nemají tak dobrou vypovídací hodnotu, jakou mohly mít.

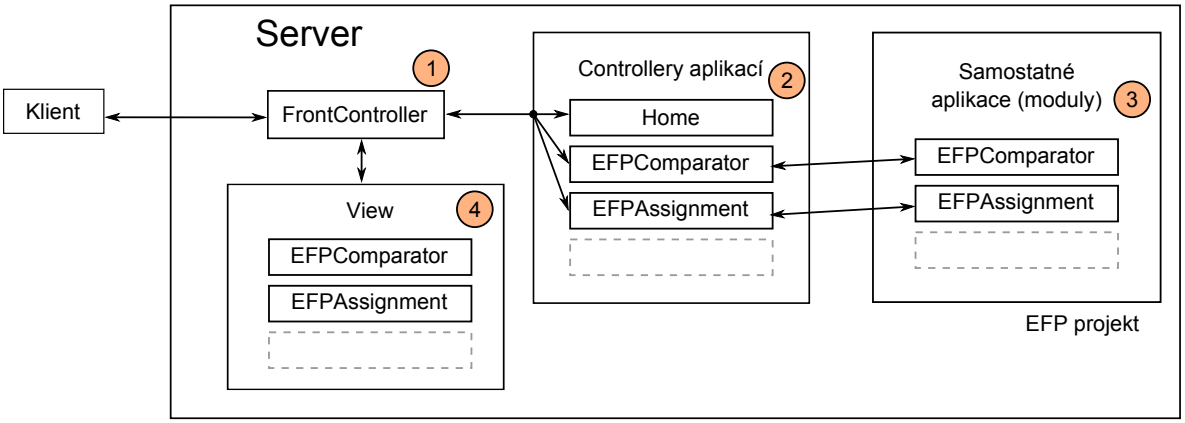
5.1 EFP Portál

V této části budou blíže představeny záměry a předpoklady, se kterými se přistupovalo k návrhu architektury a designu portálu. Též je popsáno jakým způsobem došlo k naplnění těchto cílů.

5.1.1 Architektura

Při návrhu architektury *EFP Portálu* byl již od prvních kroků brán zřetel na snadnou budoucí integraci aplikací z EFP projektu. Z těchto předpokladů odvozený výchozí návrh je tedy realizován takovým způsobem, že vývojář, jež se bude v budoucnu zabývat integrací libovolné EFP aplikace do zmiňovaného portálu již musí zasahovat jen minimálně do implementace samotného *EFP Portálu* a může se plně soustředit na návrh uživatelského rozhraní integrované aplikace.

Na obrázku 5.1 je zobrazen princip architektury *EFP Portálu*. Každá integrovaná EFP aplikace potřebuje mít v portálu svůj controller (2) a veškeré potřebné stránky týkající se view (4), přesto však stále zůstává modulem EFP projektu (3). Při vstupu požadavku na server jej převezme *FrontController* (1), který předá požadavek controlleru konkrétní aplikace (2) podle URL v něm obsažené. Tento controller předá řízení modulu dané aplikace (3), která provede svou činnost popřípadě vrátí výsledek zpracování požadavku zpět controlleru. Ten následně přes *FrontController* předá View (4) požadavek na vygenerování stránky. Vygenerovaná stránka je předána zpět *FrontControlleru* (1), který vrátí odpověď klientovi. Na tomtéž obrázku jsou vyznačeny čárkovanou čarou místa, kam je nutné přidat potřebnou funkcionalitu, aby bylo docíleno plně funkčního začlenění. Integrace aplikací do *EFP Portálu* je popsána dále v této práci.



Obrázek 5.1: Princip architektury EFP Portálu

5.1.2 Uživatelské rozhraní

Design portálu je navržen s ohledem na nasazení v akademickém prostředí a jeho výhradní určení pro seriozní činnost, tzn., že rozmístění prvků je určeno podle běžných zvyklostí pro, co nejjednodušší orientaci uživatele. Portál je proporčně přizpůsoben aplikacím, aby jim poskytnutý prostor byl co největší. Dále grafický návrh, zejména sladění barev a jejich vzájemný kontrast, je řešen takovým způsobem, aby uživatele zbytečně nerozptyloval a nepůsobil únavným dojmem i po delší době práce s integrovanými aplikacemi. Výše popsané je možné nahlédnout na obrázku 5.2.

Portál je tedy koncipován jako flexibilní kontejner sestávající z hlavní obrazovky, která obsahuje menu všech dostupných aplikací. Jednou výjimkou tohoto pravidla je první položka, která byla věnována bližším informacím o projektu. Vložení aplikace do tohoto menu je řešené na straně portálu, tudíž je z hlediska vývojáře dané aplikace triviální. Pokud uživatel vybere položku z menu aplikací je uživatelské rozhraní této aplikace zobrazeno. Samotná implementace integrované aplikace a portál jsou na sobě nezávislé entity.

Obrázek 5.2: Návrh EFP Portálu

5.2 EFP Comparator

Tato část bude věnována architektuře logiky prezentační vrstvy. Dále se bude zabývat návrhem grafického uživatelského rozhraní *EFP Comparatoru* současně s návrhem formy pro prezentaci výsledků.

5.2.1 Architektura

Při návrhu architektury logiky prezentační vrstvy bylo nutné zajistit dostatečně jednoduché a rychlé rozlišování uživatelů odpovídající požadavkům kladeným na aplikaci. Dále vyřešit vzájemnou komunikaci všech prvků tvořící aplikaci a také komunikaci klient-server včetně datových struktur potřebných pro předávání dat.

Princip ukládání nahraných komponent

U zvolené webové aplikace je potřeba řešit nahrávání komponent na server, protože k aplikaci může přistupovat více uživatelů zároveň. Z toho plyne potřeba komponenty jednotlivých uživatelů odlišovat.

Jedním z možných řešení by bylo vytváření jednotlivých uživatelských účtů. Nevýhoda tohoto řešení je potřeba registrace a přihlašování k aplikaci, která to nutně nevyžaduje. Vybraný druhý způsob řešení je uživatele odlišovat na základě přiřazeného unikátního *JSessionID*, které je uživateli přiřazeno při prvním vstupu na stránku s aplikací. Toto ID

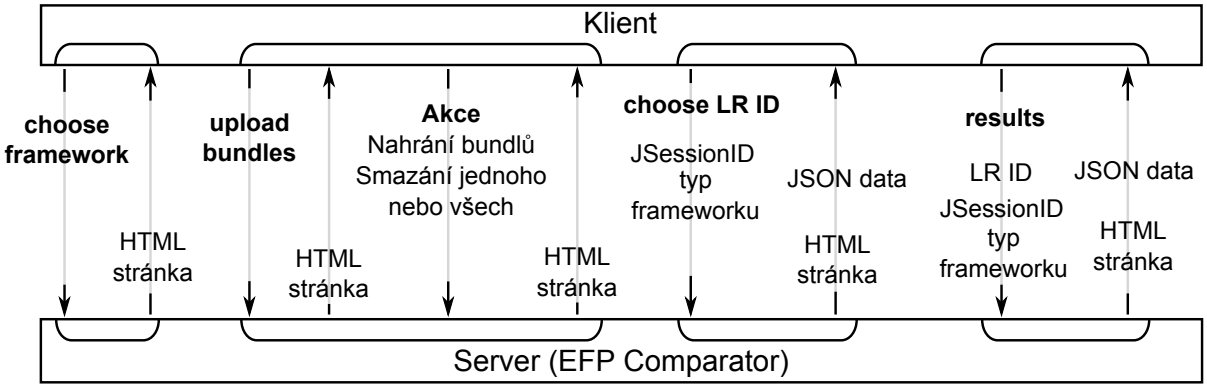
je uživateli uloženo na straně klienta do cookie.

Uživatel při jakémkoli vstupu na stránku je tedy identifikován na základě přiřazeného jedinečného ID. Komponenty, které budou nahrány na server, budou ukládány do adresáře nesoucí název podle přiděleného *JSessionID*. Z tohoto principu vyplývá nutnost odstranění komponent, které jsou nahrané na server a uživatel je při ukončení aplikace nesmazal. *JSessionID* je každému uživateli přiřazeno na určitou dobu měřenou vždy od posledního požadavku daného uživatele. Tato doba je nastavena 2 hodiny. Řešením odstraňování komponent je smazaní adresáře s komponentami po uplynutí nastavené doby. Konkrétní implementace zmíněného je v kapitole 6. Jestliže tentýž uživatel, kterému již vypršelo přiřazené *JSessionID*, přijde opět k aplikaci, pak je mu přiděleno nové *JSessionID*.

Komunikace server-klient

Princip komunikace klienta se serverem je na ukázce 5.3. Jestliže klient vybere z menu *EFP Portálu* aplikaci *EFP Comparator* je na server odeslán požadavek na zobrazení kroku *choose framework*. Klient zvolí požadovaný framework a vyžádá si následující stránku průvodce. Při přechodu na další krok je vybraný framework uložen do *cookie*.

Po zobrazení kroku *upload bundles* je klientovi umožněno spravovat komponenty v přiděleném unikátním adresáři, viz výše. Klient má k dispozici akce nahrání všech komponent, smazání jedné komponenty a nebo smazání všech komponent. Tyto akce jsou na obrázku ilustrovány jedním požadavkem. V rámci těchto požadavků je současně posílána na server identifikace adresáře *JSessionID*. Odpovědí serveru na každou z těchto akcí je zobrazení stejné stránky, přičemž je na ní zobrazen aktuální obsah komponent v poskytovaném adresáři.



Obrázek 5.3: Princip komunikace EFP Comparator-klient

Při přechodu na další krok *choose LR ID* se na server odesílá požadavek včetně *JSessionID*, aby server věděl, ze kterých komponent v daném adresáři budou vybrána ID

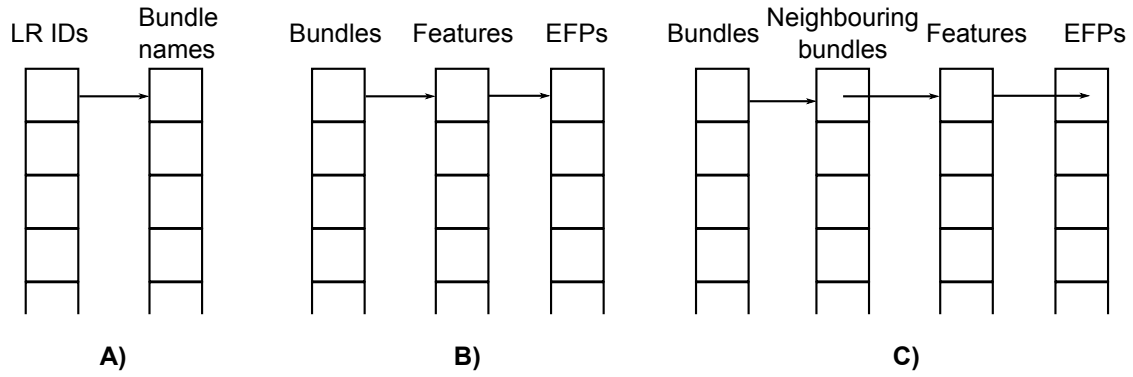
lokálních registrů. Dále také server potřebuje informaci o vybraném typu frameworku, z důvodu načítání komponent pro zjištění ID lokálních registrů. Server odpovídá požadovanou stránkou včetně dat (viz struktury dat) ve formátu *JSON* potřebných pro vyznačování spojení mezi komponentami a ID. Důvod přenosu těchto dat je blíže popsán v návrhu uživatelského rozhraní.

Jakmile si klient vybere ID a chce přejít na poslední krok *results*, tak ho klient uloží do *cookie*. Server v požadavku očekává vybrané ID, na jehož základě proběhne porovnání komponent. Také je potřeba jako v předchozím případě *JSessionID* a vybraný typ frameworku, aby bylo možné identifikovat komponenty konkrétního klienta a správně je načíst. Po porovnání komponent server odešle odpověď se stránkou včetně dat ve formátu *JSON* potřebných pro zobrazení výsledků. Struktura dat je popsána dále.

Princip odesílání dat ve formátu *JSON* v jedné odpovědi je popsán v kapitole 6. Data získaná během průchodu průvodcem ukládána do *cookies* včetně *JSessionID* jsou odesílány na server s každým požadavkem, ale server je potřebuje znát pouze ve zmíněných požadavcích. Veškerá komunikace probíhající v rámci *EFP Comparatoru* je zajištěna pomocí technologie AJAX.

Struktury dat

Pro zajištění navrhovaných funkcionalit dále rozebíraného uživatelského rozhraní bylo nutné přijít s vlastními strukturami zasílaných dat, jež přesně odpovídají požadavkům, které jsou kladeny tímto rozhraním. Nyní bude následovat popis těchto struktur v pořadí v jakém jsou požadovány při průchodu průvodcem, jejichž schéma je na obrázku 5.4.



Obrázek 5.4: Schéma datových struktur

Struktura (A), jež je potřebná pro vyznačování komponent svázaných stejným ID lokálního registru, je odesílána na klienta ve formátu *JSON* k tomuto řešení bylo nutné přistoupit, kvůli požadavkům na dynamické zobrazování dat na straně klienta. Tato struktura je reprezentována mapou ID a na nich navázaných množin se jmény komponent.

Zvolené datové typy, jež tvoří tuto strukturu umožňují efektivní výběr jmen komponent pro zvolené ID.

Struktura (B) je určená k zobrazení levé části obrazovky výsledků, jejíž popis je uveden dále. Je z důvodu efektivity kompozicí třech elementárních datových struktur typu mapa. První mapa obsahuje všechny porovnávané komponenty a každá z těchto komponent se odkazuje na mapu všech jejích vlastností (poskytovaných/vyžadovaných). Jednotlivé vlastnosti obsahují mapu všech svých mimo-funkčních charakteristik.

Struktura (C) je poslední a zároveň nejdůležitější, jejíž smysl je umožnit zobrazení dat u klienta filtrovaných na základě relevance k prvku zvolenému na levé straně výsledků, takto vybraná data budou zobrazena na pravé straně obrazovky výsledků. Upřesnění principu filtrace dat je uvedeno dále v části uživatelské rozhraní. Tato struktura je odesílána v odpovědi serveru ve formátu *JSON*.

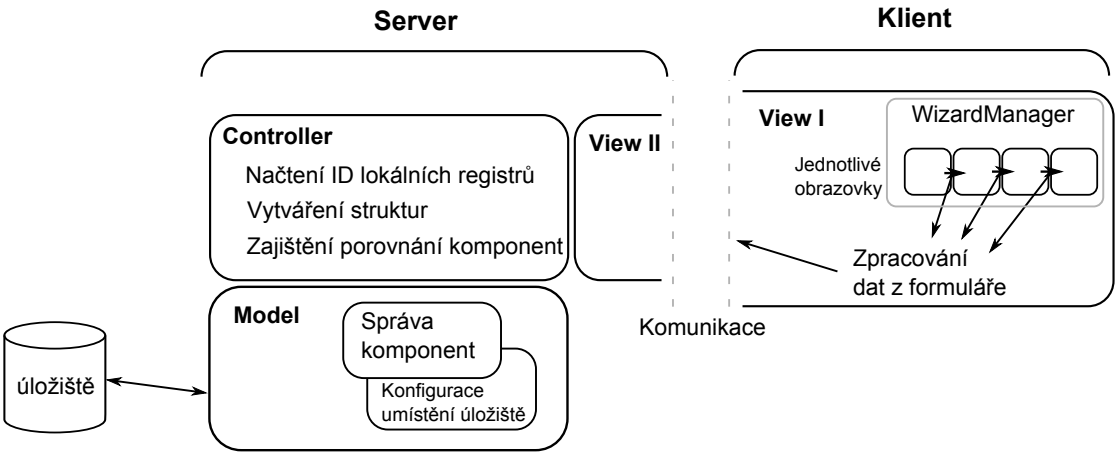
Náhled na strukturu EFP Comparatoru

Struktura *EFP Comparatoru* je postavena na klasickém MVC modelu. Tento model je aplikován takovým způsobem, že vrstvy *Model-Controller* jsou umístěny na serveru a vrstva *View* je rozdělena na dvě části, kde jedna z částí je na straně klienta kvůli potřebě dynamických změn vzniklé na základě snahy o zlepšení uživatelské přívětivosti. Druhá část je *View* určená k vytváření stránek na serveru, které budou odesílány klientovi.

Vrstva *Controlleru* obsluhuje veškeré požadavky aplikace. Mezi tyto požadavky patří získání ID lokálních registrů z uživatelem nahraných komponent. Takto získanými daty je následně naplněna vlastní datová struktura (A) na obrázku 5.4 popsaná výše. Dále zajistí porovnání komponent a vytvoření struktur (B,C) potřebných pro podání dostatečně vypovídajících výsledků. *Controller* komunikuje s modelem v případě vznesení požadavku na manipulaci s nahrávanými komponentami.

Model spravuje komponenty nahrané v úložišti, kde toto úložiště je realizováno adresářem na pevném disku, v němž má každý uživatel vyhrazený unikátní prostor. Umístění adresáře s úložištěm je možné definovat za použití specifického parametru v konfiguračním souboru. Tento MVC model je zobrazen na obrázku 5.5.

Komunikace na obrázku mezi *View* na straně klienta a serverovou částí je popsána výše. Princip činnosti klientské strany *View* je také vidět na zmíněném obrázku. Jednotlivé stavy kroků na straně klienta spravuje *WizardManager*. Vždy při přechodu na následující krok v průvodci *WizardManager* volá akce potřebné pro přechod mezi těmito kroky (zpracování formulářů z předchozích stavů). Poté dojde k odeslání příslušného požadavku o danou stránku na server.



Obrázek 5.5: Schéma MVC

5.2.2 Uživatelské rozhraní

Jak bylo zmíněno v kapitole 4, tak *EFP Comparator* nemá grafické uživatelské rozhraní a jeho ovládání je řešeno pouze zadáváním parametrů při spouštění. Výsledky jsou taktéž prezentovány nikterak přívětivou formou. Z toho plyne, že pro uživatele je porozumění výsledkům porovnání zbytečně komplikované.

Při spouštění *EFP Comparatoru* je očekáváno několik parametrů. Prvním parametrem je umístění adresáře obsahujícího komponenty určené k porovnání, následně je vyžadováno ID lokálního registru, jehož popis je uveden v podkapitole 3.1. Poslední parametr je volitelný, určuje typ komponent, pokud není zadán, tak *EFP Comparator* automaticky předpokládá komponenty typu *CoSi*.

Větší množství zadávaných parametrů bylo motivací ke zvolení postupného zadávání parametrů. Pro tuto možnost se jako nejlepší volba jeví průvodce (wizard), který umožňuje libovolné procházení zadávaných parametrů oběma směry. Největší výhodou průvodce je, že uživatele postupně provede zadáním všech parametrů. Každý krok má jednoznačný význam, což výrazně zlepšuje orientaci a přehlednost v nastavení parametrů oproti variantě, kde by se všechny parametry nastavovaly současně v jednom formuláři. Orientace v průvodci je dále posílena nasazením navigace, jež uživateli sděluje jeho momentální pozici v průvodci.

V rámci vytváření uživatelského rozhraní byla odstraněna i jedna z největších nevýhod při zadávaní parametrů a to požadavek na znalost ID lokálního registru každé porovnávané množiny komponent. Nyní budou postupně představeny jednotlivé kroky průvodce, jehož návrh byl realizován v nástroji *Lumzy*.

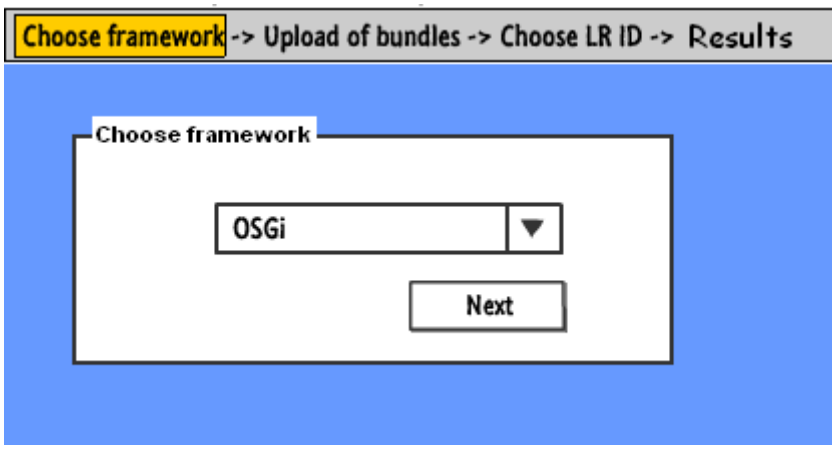
Lumzy

Webový nástroj *Lumzy*[1] poskytuje jednoduché prostředí pro návrh raných verzí webových stránek. Návrh je pouze hrubým náčrtem finálního vzhledu stránky a jeho účelem je otestování možného rozmístění uživatelských prvků a nástin grafického řešení.

V *Lumzy* je k dispozici většina prvků běžně se vyskytujících na webových stránkách, což jeho uživateli umožňuje sestavit základní návrh ve velmi krátkém čase. Mnoho z poskytovaných prvků simuluje jednoduchou interakci s uživatelem, takže je možné vyzkoušet účelnost rozvržení jednotlivých stránek.

Výběr frameworku

První krok průvodce je formulář, který dává uživateli na výběr mezi dvěma typy komponent (OSGi, CoSi), které budou nahrány na server a později porovnány. Formulář je zobrazen na obrázku 5.6, kde lze mimo jiné také vidět zmíněnou navigaci zobrazující pozici aktuálního kroku v průvodci.



Obrázek 5.6: Návrh kroku výběr frameworku

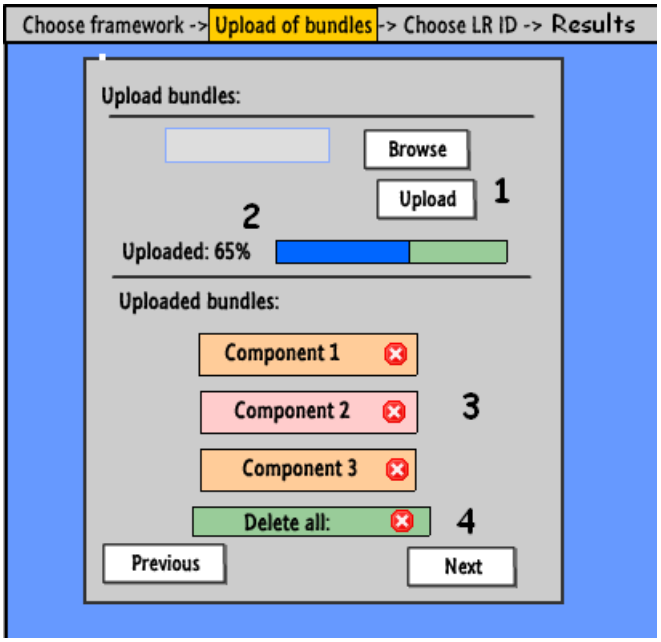
Tento krok byl navržen s ohledem na stávající řešení, které vylepšuje tím, že dává uživateli na výběr jednu z možností. V případě ponechání přednastavené hodnoty uživatel vidí, který typ komponent je vybrán a bude porovnáván. Kdežto u stávajícího řešení bez vybrání typu se automaticky vybere OSGi, viz výše a pokud uživatel aplikaci nezná, tak neví, jaký typ byl vlastně vybrán.

[1] http://lumzy.com/

Nahrání komponent

Na této obrazovce byl návrh vylepšen oproti původnímu řešení, které umožňovalo zadat pouze jeden adresář, ze kterého se porovnávaly všechny obsažené komponenty. V tomto návrhu byly možnosti uživatele podstatně rozšířeny. Může tak podle potřeby nahrávat na server libovolné komponenty z daného adresáře včetně nahrání všech. Toto řešení umožňuje i nahrávání libovolných komponent z jiný adresářů, takže výsledně vybrané komponenty ani nemusí tvořit celou aplikaci, ale pouze její část, aniž by uživatel musel z daného adresáře některé odstranit, pokud by chtěl na server nahrát jen určitou podmnožinu komponent.

Formulář pro nahrávání komponent (1) je vidět na obrázku 5.7. Pro lepší orientaci během nahrávání velkého množství komponent byl navržen prvek *progressbar* (2), který uživateli podává informaci o tom, kolik procent z vybraných komponent je již na server nahráno.



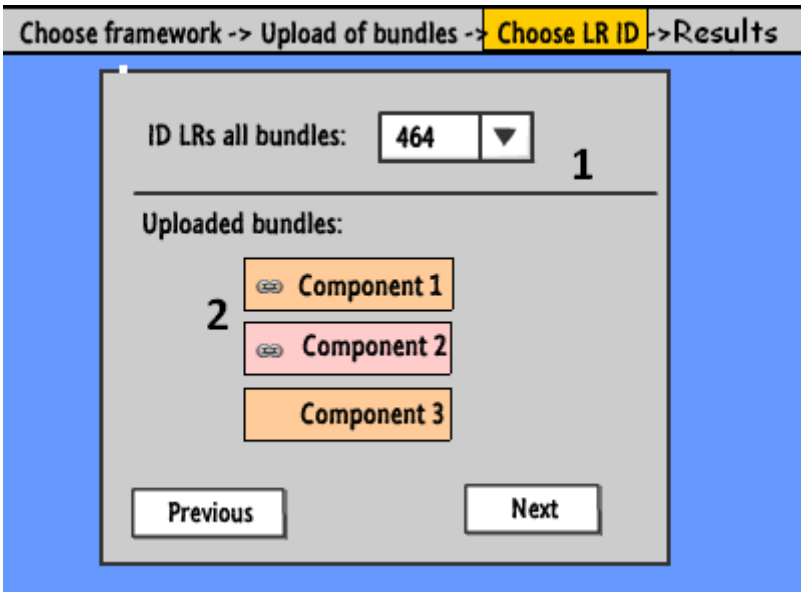
Obrázek 5.7: Návrh kroku nahrání komponent

Po nahrání komponent jsou všechny zobrazeny pod nahrávacím formulářem (3) včetně těch, které na serveru byly nahrány již předtím. Z toho plyne, že uživatel po nahrání jedné množiny komponent, může nahrát další. K tomuto návrhu také patří možnost nahrané komponenty na server smazat buď jednotlivě a nebo všechny najednou (4). V případě nahrání souborů neodpovídajících očekávanému formátu je zobrazena informace o nepodporovaném formátu. Uživateli je znemožněno pokračovat na další krok, dokud nenahraje komponenty.

Výběr lokálního registru

Jak již bylo zmíněno, tento krok je navržen tak, aby uživatel nemusel mít znalost ID lokálního registru jako v původním řešení *EFP Comparatoru*. V tomto kroku jsou uživateli nabídnuta všechna ID lokálních registrů, které jsou obsaženy v nahraných komponentách z předchozí obrazovky, což je vidět na obrázku 5.8 v bodě (1).

V tomto případě by sice uživatel mohl vybrat jedno z nabízených ID lokálních registrů, ale zase by přišel o znalost, se kterým ID jsou spojeny které komponenty. Návrh byl tedy ještě rozšířen o vyznačování vazeb daného ID s komponentami, které jsou s ním spřaženy. Komponenty jejichž ID odpovídá vybranému jsou označeny ikonou řetězu (2), jak je vidět na obrázku.



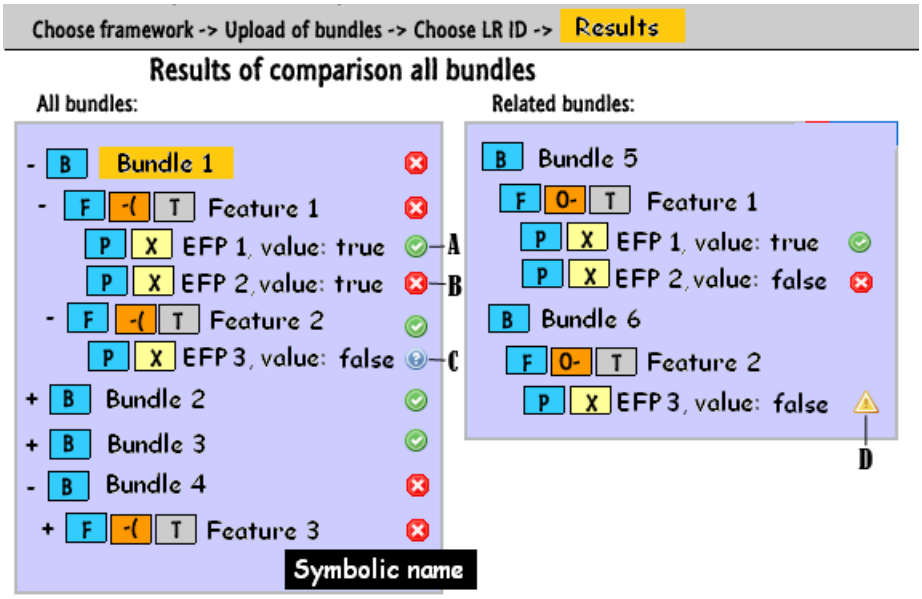
Obrázek 5.8: Návrh kroku výběr lokálního registru

Zobrazení výsledků

Poslední obrazovkou v průvodci je zobrazení výsledků porovnání. *EFP Comparator* poskytuje na základě porovnání poskytnutých komponent velmi široké spektrum informací. Jedním z nejdůležitějších úkolů této práce bylo zprostředkování těchto informací uživateli takovým způsobem, aby z nich byl schopen získat co nejvíce relevantních informací. Proto bylo návrhu, jakým způsobem budou získané informace uživateli poskytnuty, věnováno mnoho úsilí. V zájmu dosažení tohoto cíle uspokojivým způsobem bylo nutné zvolit příznivý poměr mezi rozsahem a přehledností podávaných dat.

Jako ideální kandidát pro reprezentaci výsledků porovnání byl tedy zvolen způsob

několikaúrovňového stromu. V něm jsou na nejvyšší úrovni zobrazeny uživatelem vybrané komponenty, na nižší úrovni jsou vlastnosti a úplně nejníže jsou zobrazeny mimo-funkční charakteristiky. V rámci usnadnění orientace ve výsledcích je použito několik grafických symbolů, jež umožňují předat uživateli informace v kompaktní, ale stále přehledné formě.



Obrázek 5.9: Návrh kroku výsledky (po zvolení komponenty)

Na obrázku 5.9 je návrh možné reprezentace výsledků vytvořený v nástroji *Lumzy*. Použité symboly jsou pouze ilustrační a nejedná se o finální verze. V následujícím výčtu bude přiblížen význam každého z užitých symbolů. Obrazovka výsledků je rozdělena do dvou částí, kde levá část vždy zobrazuje kompletní přehled všech komponent, které mají pouze vyžadované vlastnosti. V pravé části jsou případně zobrazeny komponenty závislé na zvoleném prvku v levé části a vlastnosti u nich jsou vždy poskytované. Výsledná data jsou prezentována formou víceúrovňového stromu, kde levá strana zpravidla obsahuje velké množství prvků a proto je vždy sbalena. Pravá strana se ve výchozím stavu naopak zobrazuje rozbalena.

Symbol *B* označuje komponentu.

Symbol *F* označuje vlastnost komponenty.

Symbol *P* označuje EFP příslušející dané vlastnosti.

Symbol *„-(“* zobrazuje, že daná vlastnost je komponentou vyžadována.

Symbol *„O-“* zobrazuje, že daná vlastnost je komponentou poskytována.

Symbol *T* označuje typ vlastnosti. V případě OSGi je to *PACKAGE* a u CoSi se může jednat buď o *EVENT*, *INTERFACE* a *TYPE*.

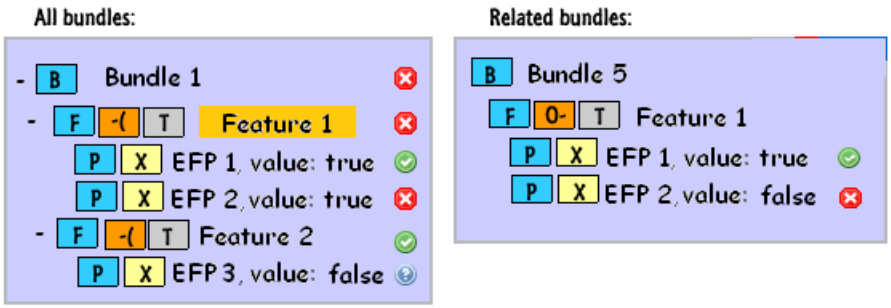
Symbol *X* udává typ EFP *SIMPLE* nebo *DERIVED*, viz podkapitola 3.2.

Další poskytovaná informace je hodnota dané EFP, která je zobrazena vždy s názvem EFP. U vlastnosti je zobrazena její verze a také symbolické jméno v popisku, který je zobrazen po najetí kurzoru myši na jméno vlastnosti. Samotný výsledek porovnání je zobrazen u EFP a symbolizován ikonami, které jsou odlišné pro každý ze čtyř možných stavů. Výsledek porovnání může být buď pozitivní (A) nebo negativní (B). Negativní stav může mít dvě možné příčiny. Pokud nedošlo k porovnání daného EFP (C), znamená to, že EFP na protější straně chybí (D).

Výsledky porovnání všech EFP jsou předány až na nejvyšší úroveň v rámci každé jednotlivé komponenty. To znamená, jestliže na úrovni libovolného z podřízených EFP dojde k chybě jsou jako chybné označeny všechny nadřazené celky. Například pokud u jediného EFP nastane jakýkoliv z chybových stavů, je vlastnost a následně i komponenta, obsahující tuto EFP, označena jako chybná a to bez ohledu na to, že výsledky porovnání zbylých EFP mohou být v pořádku.

Po kliknutí na komponentu v levé části obrazovky, dojde k podbarvení zvolené komponenty (viz obrázek 5.9) se v pravé zobrazí všechny komponenty, které jsou se zvolenou komponentou svázány přes společné vlastnosti. U komponent v pravé části jsou zobrazeny pouze ty vlastnosti, jež jsou společné se zvolenou komponentou v části levé. Pro každou zobrazenou vlastnost v pravé části jsou zároveň zobrazeny všechny její EFP. Dále je možné kliknout na vlastnost či EFP, což způsobí filtrování zobrazených výsledků v levé části obrazovky.

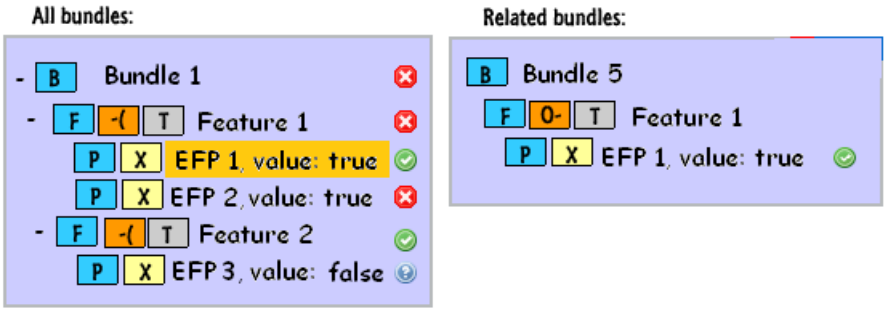
Při výběru vlastnosti v levém stromě se oproti předchozímu případu zobrazí komponenta spojená přes zvolenou vlastnost. U této komponenty je zobrazena pouze společná vlastnost včetně všech jejích EFP. Tento případ zachycuje obrázek 5.10.



Obrázek 5.10: Návrh kroku výsledky (po zvolení vlastnosti)

Poslední filtrem, jež je možné aplikovat na výsledky, je aplikován po kliknutí na EFP v levé části. V tomto případě bude v pravé části zobrazena komponenta a její vlastnost,

přes kterou je spojena s komponentou obsahující vybrané EFP. Jestliže vlastnost v pravé části obsahuje více EFP, je pro přehlednost zobrazeno pouze zvolené EFP.



Obrázek 5.11: Návrh kroku výsledky (po zvolení EFP)

6 Implementace EFP Portálu

Tato kapitola se zabývá všemi celky, jež tvoří strukturu *EFP Portálu* navrženou v kapitole 5 návrh. Nejdříve je popsána konfigurace částí frameworku *Spring MVC*, jež byly v této práci využity včetně jejich implementace. Popis implementace je dále rozdělen na klientskou a serverovou část, kde v klientské části jsou popsány principy průchodu uživatele průvodcem aplikací a v serverové části je popsána problematika rozpoznání uživatele a zpracování jím nahraných komponent. Dále se serverová část zabývá strukturami navrženými pro uchování zpracovávaných dat takovým způsobem, jež umožní uživateli získat pro něj relevantní informace.

6.1 Struktura projektu

Zdrojové kódy projektu tvořící jádro aplikace jsou rozděleny do logických celků, které se striktně drží vzoru MVC. Celá aplikace se rozděluje na tři hlavní adresáře *WEB-INF*, *resources* a *sources*.

V adresáři *WEB-INF* se nachází konfigurační soubory *web.xml*, *applicationContext.xml*, *dispatcher-servlet.xml* pro celý *EFP Portál*, layout s rozvržením celé stránky *EFP Portálu*, k jeho dalšímu obsahu patří také adresář *views* obsahující JSP soubory, ze kterých je sestavena stránka *EFP Portálu*. Navíc tento adresář dále obsahuje zvláštní složku pro každou integrovanou aplikaci.

Všechny zdroje typu obrázků, JavaScriptů či CSS souborů jsou uloženy v adresáři *resources*. Pro každý typ těchto zdrojů je určena samostatná složka, kde má každá z integrovaných aplikací vlastní adresář se soubory daného typu. Soubory nacházející se v této složce volně jsou společné pro všechny aplikace.

Zdrojové soubory samotného *EFP Comparatoru* jsou umístěny v adresáři sources, ve kterém se nachází jednotlivé balíčky, jež jsou organizovány na základě jejich funkcionality.

6.2 Spring MVC

Aplikace je založena na frameworku *Spring MVC*, jehož obecný popis je uveden v kapitole 4. V této části bude uvedena implementace použitých vlastností ze *Spring MVC*.

6.2.1 Lokalizace a internacionalizace

Lokalizace v této práci je vyřešena jednoduchým a efektivním způsobem. V konfiguračním souboru *applicationContext.xml* jsou definovány beany *messageSource*, *localeChangeInterceptor*, *localeResolver* a *handlerMapping*.

V beaně *messageSource* se určí základ jména (*basename*) pro properties soubory, ve kterých bude uložen překlad do jednotlivých jazyků a také jejich kódování. V *EFP Portálu* je jako *basename* nazváno *messages*. Properites soubory jsou umístěny v adresáři *resources*.

Standardně je nastavena taková lokalizace, která je uvedena jako parametr beany *localeResolver*, která má v tomto případě hodnotu *en*. Lokalizace je uložena do cookies pro dané *JSessionID*, které je přiděleno při vstupu na *EFP Portál*.

Změna lokalizace se provede právě tehdy, když přijde na server požadavek s parametrem *lang*, jehož hodnota je automaticky uložena do cookies. Tento parametr je definován v beaně *localeChangeInterceptor*. Poslední nezmíněnou beanou je *handlerMapping*, která předává všechny požadavky beaně *localeChangeInterceptor*.

Properties soubor s danou lokalizací je Springem vybrán složením názvu *basename* nastaveném v beaně *messageSource* s lokalizací nastavené v cookies. Spring v tomto případě bude hledat soubor *messages_en.properties*. Nacházející se v adresáři resources.

Po výše uvedených konfiguracích je potřeba představit, jakým způsobem je možné přistupovat k jednotlivým záznamům uložených v properties souborech. Aby bylo možné přistoupit k těmto záznamům v JSP stránkách, je nutné v těchto stránkách uvést Springem definovaný taglib, který je zobrazen na následující ukázce.

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<spring:message code="bundles.required"/>
```

Na místě, kde má být zobrazena hodnota konkrétního záznamu, se musí využít tag *spring:message*, jehož parametr *code* požaduje klíč tohoto záznamu v properties souboru.

6.2.2 Konfigurace anotací

Anotace použité ve Springu byly obecně popsány v kapitole 4, kde bylo také uvedeno, že musí být použití anotací explicitně zavedeno v konfiguraci, jinak je nelze standardně využívat. A to i přesto, že Spring nativně anotace podporuje.

Aby bylo možné použít anotace v této práci, musely být nastaveny v konfiguračním souboru *dispatcher-servlet.xml* parametry uvedené v následující ukázce:

```
<mvc:annotation-driven />
<context:component-scan base-package="cz.zcu.kiv.efps.efpportal" />
```

První z uvedených umožní ve Springu použit anotace a druhý řádek z konfigurace znamená, že Spring bude hledat anotované části pouze od balíčku uvedeném v parametru *base-package* včetně jeho a v dalších pod balíčcích.

6.2.3 Konfigurace nahrávání komponent

V této aplikaci byla potřeba nahrávání souborů na server jedním ze stěžejních úkolů. Aby toto Spring podporoval bylo nutné nastavit v konfiguračním souboru *dispatcher-servlet.xml* třídu *multipartResolver*, která nahrávání souborů zajistí. V tomto nastavení je také možné pomocí parametru *maxUploadSize* určit maximální velikost nahrávaného souboru. Nastavení je vidět na následující ukázce:

```
<bean id="multipartResolver"
     class="org.springframework.web.multipart.commons.Commons-
     MultipartResolver">
     <property name="maxUploadSize" value="10000000"/>
</bean>
```

V této práci je díky HTML5 umožněno nahrávat více souborů najednou. Při nahrávání více souborů najednou je však tato velikost vztažena k celkovému součtu velikostí jednotlivých nahrávaných souborů.

6.2.4 Tiles2

Tiles2 jsou v této práci použity pro skládání obsahu jednotlivých JSP souborů, který by se jinak v každém JSP souboru musel opakovat. Více o principu skládání je popsáno v části 4.1.3, kde bylo také zmíněno, že *Tiles2* nejsou součástí frameworku *Spring MVC* a musí být explicitně přidán.

Jak jíž bylo řečeno, tak celý EFP projekt je sestavován pomocí nástroje Maven. Přidání *Tiles2* do Springu bylo tedy zajištěno Mavenem konfigurací souboru *pom.xml*, který je umístěn přímo v adresáři *efpPortal*.

Konfigurace

Po přidání Tiles2 do Springu bylo nutné ještě provést konfiguraci. V *applicationContext.xml* muselo být definováno *Tiles2*, aby ho Spring bral v potaz. V tomto XML souboru je také nastavení cest, kde má Spring hledat schémata pro sestavení jednotlivých stránek. Na následující ukázce je vidět, že standardní schéma *tiles.xml* pro rozvržení *EFP Portálu* se nachází ve složce *layout*, kde se mimo jiné nachází soubor *layout.jsp*. Jednotlivá schémata *views.xml* samostatných stránek jsou ve složkách konkrétních EFP aplikací. Definice uvedené v těchto schématech by všechny mohly být umístěny ve schématu *tiles.xml*. Z důvodu přehlednosti jsou pro každou aplikaci jednotlivá schémata ve zvláštních souborech *views.xml*.

```
<property name="definitions">
  <list>
    <value>/WEB-INF/layout/tiles.xml</value>
    <value>/WEB-INF/views/**/views.xml</value>
  </list>
</property>
```

Pro opakující se části kódu byly vytvořeny samostatné JSP soubory *header.jsp*, *menu.jsp* a *footer.jsp*. V souboru *layout.jsp* jsou určená místa, která jsou zobrazena na následující ukázce.

```
<tiles:insertAttribute name="title" />
<tiles:insertAttribute name="header" />
<tiles:insertAttribute name="menu" />
<tiles:insertAttribute name="body" />
<tiles:insertAttribute name="footer" />
```

Na tato místa se budou vytvořené JSP soubory vkládat podle definice ze souboru *tiles.xml*, která je vidět na následující ukázce. Určená místa jsou pojmenována parametrem *name*.

```
<definition name="default" template="/WEB-INF/layout/layout.jsp">
  <put-attribute name="header" value="/WEB-INF/views/header.jsp"/>
  <put-attribute name="menu" value="/WEB-INF/views/menu.jsp"/>
  <put-attribute name="footer" value="/WEB-INF/views/footer.jsp"/>
</definition>
```

Definice v *tiles.xml* pojmenovaná *default* bere jako šablonu zmíněný soubor *layout.jsp*. Na pojmenovaná místa určená v tomto souboru se budou vkládat soubory definovány v parametru *value*, jež jsou pojmenovány stejně.

V *tiles.xml* není nadefinován atribut se jménem *title* a *body*, protože tyto části jsou pro každou stránku odlišné. Každá akce v controlleru vrací *name*, podle kterého se určí odpovídající definice. Jestliže je vrácený *name* shodný s jednou z definic, tak je vybrána právě tato definice pro složení konkrétního JSP souboru.

V této práci jsou použity dva druhy definic. Jedna z definic je stejná jako v ukázce s *tiles.xml*, jež má určen parametr *name* a k němu odpovídající JSP soubor v tomto případě *layout.jsp*. Druhá možnost je definice, jež má parametr *name* a k němu parametr *extends*, ve kterém je jméno jiné definice, kterou rozšiřuje. Což je vidět na poslední následující ukázce demonstrující příklad s doplněním určených míst pro *body* a *title*.

```
<definition extends="default" name="efpcomparator/index">
  <put-attribute name="body"
      value="/WEB-INF/views/efpcomparator/index.jsp" />
  <put-attribute name="title"
      value="EFP Comparator - Choose framework" />
</definition>
```

Tento princip byl vhodný pro zobrazování stránek *EFP Portálu* dostupných z jeho menu. Avšak nevyhovoval v případě *EFP Comparatoru*, protože klient se dotazuje pomocí technologie AJAX a tak očekává pouze část stránky resp. jednotlivé kroky průvodce. To znamená, že nechce od serveru poslat celou novou stránku včetně hlavičky, menu a patičky, protože by jinak docházelo k problému se zanořováním stránky do stránky. Toto se vyřešilo použitím prvního zmíněného způsobu použití definic, kde se pro jednotlivé kroky *EFP Comparatoru* vytvořily definice, které nerozšiřovaly původní layout.

Problém s opětovným odesláním celé stránky *EFP Portálu* na požadavek *EFP Comparatoru* pomocí AJAXu byl vyřešen, takže se odesílají pouze jednotlivé části, aniž by docházelo k zanořování stránek.

Na prvotní zobrazení *EFP Comparatoru* bude Springem vybrána akce stepFramework() z *EFPComparatorControlleru*, která reaguje na metodu GET a url "/"nebo "". Tato akce dává Springu name "efpcomparator/index", který bude použit pro vyhledání konkrétní definice ve *views.xml* ve složce *efpcomparator*. Definice z *Tiles2* se jménem *efpcomparator/index* rozšiřuje standardní definici layoutu a tudíž se poskládá na serveru celá HTML stránka včetně hlavičky, menu atd., která se odešle klientovi, což je v pořádku.

Problém však nastává, jestliže byl u klienta zobrazen krok s nahráváním komponent a přišel požadavek na krok předchozí. Požadavek byl odeslán na server AJAXem metodou POST. Jak již ale bylo řečeno, tak první krok reaguje na metodu GET, tudíž by nebyla stránka zobrazena a došlo by k chybovému hlášení, že stránka nebyla nalezena. Tento problém by se dal jednoduše vyřešit tak, že by se do příslušné akce stepFramework() přidala reakce i na metodu POST. To by však způsobilo známý problém se zanořováním stránek do sebe, protože se při prvotním zobrazení odesílá celá stránka.

Pro návrat na první krok průvodce musela být implementována nová akce reagující na metodu POST, jež Springu odesílá *name efpcomparator/back* pro získání definice z *views.xml*, u které je definován parametr *template* pouze pro příslušný JSP soubor generující část stránky s vybráním frameworku a nikoli stránky celé.

U ostatních přechodů na následující/předchozí kroky tento problém nenastává z principu, protože požadavky na jednotlivé kroky jsou odesílány vždy metodu POST nikoli kombinovaně.

6.3 Implementace EFP Comparatoru z pohledu serveru

Celá aplikace používá logování za pomoci knihovny *log4j*, která musela být přidána do konfiguračního souboru *pom.xml* Mavenu. Samotná konfigurace této knihovny se nachází

ve stejném adresáři jako properties soubory tedy v adresáři *resources*.

V aplikaci se nacházejí tři kontroléry umístěny v balíčku *cz.kiv.efps.efpportal.controllers*. První kontrolér pouze zajišťuje zobrazení stránky s informacemi o celém *EFP Portálu*, jeho aplikacích a odkazy na zdroje s materiály ohledně EFP projektu. Ostatní kontroléry jsou již pro integrované aplikace. *EFPComparatorController* je jediným kontrolérem pro aplikaci *EFP Comparator*, kde jsou jednotlivými akcemi obslouženy všechny požadavky *EFP Comparatoru*.

V této práci jsou použity anotace, které byly nakonfigurovány výše. Třídy s kontroléry jsou POJO, které jsou anotovány anotací *@Controller*. Akce jsou metody, které mají anotaci mapování například *@RequestMapping(value = {"/", ""}, method = RequestMethod.GET)*. Anotace s mapováním může být uvedena i u třídy, která je kontrolérem. Výsledné URL se pak skládá z mapování kontroléru a akce. V případě, že kontrolér je namapován na URL */efpcomparator* a akce na */results*, tak bude výsledná URL */efpcomparator/results*.

Správa komponent na serveru

Třída *StorageManager* zajišťuje veškeré akce týkajících se nahraných komponent. Slouží k jednotlivému či hromadném nahrávání a mazání komponent uživatelem v jemu přiděleném adresáři pojmenovaném podle *JSessionID*. Dále umožňuje tento adresář smazat.

V návrhu *EFP Comparatoru* byl popsán princip přidělování adresářů na základě *JSessionID* včetně doby trvání přidělené *session*. Tato doba v minutách se nastaví v konfiguračním souboru *web.xml*. Smazání přiděleného adresáře včetně nahraných komponent zajišťuje třída *SessionHandler*, ve které jsou metody *sessionCreated* a *sessionDestroyed*, jež jsou volány Springem při vytvoření nového *JSessionID* nebo při jeho odstranění serverem po vypršení jeho platnosti. Volání těchto metod Springem je zajištěno registrováním třídy obsahující zmíněné metody jako listeneru v konfiguračním souboru *web.xml*.

Třída *StorageManager* používá k určení umístění úložiště třídu *StorageConfiguration*, do níž je Springem „injektována“ cesta k úložišti, která je definována v konfiguračním souboru *applicationContext.xml*. Pro správu souborů je používána externí knihovna *apache commons-io*, která musela být přidána do *pom.xml*.

Princip vytváření struktur

Po příchodu požadavku od klienta na obrazovku s výběrem lokálních registrů je potřeba datová struktura, jež umožní na straně klienta dynamicky vyznačovat komponenty

vzájemně sdílející zvolené ID.

K dosažení popsané funkce je na straně serveru nutné mít k dispozici *JSessionID*, které je uloženo v cookie uživatele společně s typem frameworku. Tyto parametry jsou společně odeslány s požadavkem. Na základě *JSessionID* jsou načteny příslušné komponenty z úložiště a poté zvolen modul z *efpAssignmentu*, odpovídající typu frameworku zvolenému uživatelem, pro získání zmíněných ID z nahraných komponent. O vytvoření datové struktury popsané v návrhu se stará třída *BundleLRService*.

Datová struktura je odesílána ve formátu JSON spolu s vygenerovanou HTML stránkou. Princip odeslání struktury ve formátu JSON v jednom požadavku spočívá v uložení celé této struktury do JavaScriptové proměnné umístěné v odpovídajícím HTML tagu (¡script¿) odesílané HTML stránky.

Po vyžádání poslední stránky s výsledky uživatelem je na server zaslán požadavek obsahující ID lokálních registrů, *JSessionID* a typ frameworku, což jsou data, jež byla po celou dobu průběžně ukládána do cookie na straně klienta a jsou to data, jež EFP Comparator vyžaduje pro porovnání komponent. V případě *JSessionID* se ovšem jedná pouze o identifikaci adresáře, ve kterém se nacházejí komponenty, jež budou porovnávány. Výsledkem porovnání je datová struktura popsána v části 3.2.3.

Struktura, jež je výstupem *EFP Comparatoru* není ve tvaru, jež by umožnil zobrazení těchto dat takovým způsobem, který byl navržen a popsán v rámci kapitoly 5. Tento návrh počítá s převedením těchto výsledků do dvou struktur. Obě struktury jsou odesílány serverem současně spolu s vygenerovanou stránkou. Struktura určená pro levou část výsledků je ještě před odesláním odpovědi převedena do HTML struktury. Druhá struktura je opět odesílána ve formátu JSON stejně jako tomu bylo v předchozím případě.

6.4 Implementace EFP Comparatoru z pohledu klienta

Stejně jako tomu bylo u serverové strany, tak bylo i na straně klienta použito logování tentokrát však za použití objektu *console*.

V adresáři *js*, ve kterém jsou všechny JavaScriptové soubory, je složka vyhrazená *EFP Comparatoru*, kde se nachází všechny zdrojové soubory. Pro správu jednotlivých obrazovek náležících průvodci aplikací slouží *WizardManager.js*. *Wizard.js* obsahuje všechny akce odpovídající dotazům na obrazovku průvodce, jež může klient vyžadovat. Ostatní soubory tvoří dvojici určenou pro každou z obrazovek průvodce. Soubory jejichž název končí slovem *Events* obsahují registrace událostí pro ovládací prvky. Tyto události volají akce uložené v souborech končících slovem *Actions*.

Po zobrazení každé obrazovky jsou vždy zaregistrovány události na tlačítka *previous* a *next* funkcí *setHandlers()*. Po stisknutí jednoho z tlačítek jsou vyvolány funkce *stepNext()* nebo *stepPrev()* z *WizardManagera*. Tyto funkce po přechodu na danou obrazovku volají odpovídající akce ze souboru *Wizard.js*.

Přijmutím odpovědi od serveru je získanými daty přepsána část DOMu, v níž se nachází měněný obsah stránky. Tímto způsobem je zjednodušena komunikace mezi klientem a serverem, protože je třeba načítat pouze měněný obsah a ne celou stránku (AJAX). Při implementaci AJAXové komunikace došlo k problému, když byl odeslán požadavek na server a poté byly registrovány události na prvky vyskytující se na stránce, jež měla přijít v odpovědi. Problém však byl v tom, že se události na prvky registrovaly ještě předtím, než přišla odpověď od serveru a tudíž musely být veškeré registrace událostí pro daný krok vykonány v „callbackové“ funkci *success()*, protože teprve až v ní je jasné, že požadavek a odpověď na něj proběhly v pořádku.

Pro zvýšení uživatelské přívětivosti byl implementován mezi jednotlivými přechody aplikace *loader*, který symbolizuje, že aplikace vykonává nějakou činnost. K tomuto byl použit *plugin spin*[1] do *jQuery*.

Výběr frameworku

Po vstupu uživatele na úvodní obrazovku průvodce aplikací *EFP Comparator* proběhne počáteční inicializace *WizardManagera* a registrace události na tlačítko *next* pomocí funkce *setHandlers()*. Po vybrání frameworku a stisknutí tlačítka *next* je *WizardManagerem* zavolána funkce *beforeUploadBundleAction()*, která provede uložení vybraného typu frameworku do cookie. Poté je zavolána funkce *showUploadBundles()*, jež zajistí odeslání požadavku na server a po příjmu odpovědi zaregistruje události na ovládací prvky (tlačítka *next* a *previous*) pomocí funkce *setHandlers()*. Tato funkce je volána vždy při přechodu na jinou obrazovku. Poslední činnost této funkce je registrace událostí, jež obstarávají správu týkající se nahrávaných komponent.

Nahrávání komponent

Pro nahrávání komponent na server byla využita vlastnost HTML5, která umožňuje nahrávat více souborů najednou. Konkrétně se jedná o atribut *multiple*, jehož použití je na následující ukázce.

[1] http://fgnass.github.com/spin.js/

```
<input id="bundles" type="file" multiple="multiple"
required="required" name="bundles" />

<progress id="progress" max="100" value="0" />
```

Na stejné ukázce je také vidět využití nového atributu z HTML5. Při odeslání formuláře musí element s daným atributem obsahovat data. Součástí ukázky je také použití nového elementu *progress*, jež je použit pro informování uživatele o postupu nahrávání dat na server.

Nahrání komponent na server probíhá pomocí technologie AJAX. Jako součást obsluhy události stisku tlačítka *upload* je volána funkce *sendBundlesToServerAction()*, která zajistí odeslání souborů. Při nahrávání souborů byl použit plugin *jquery.form*[2].

V předchozím kroku se zaregistrovaly události na obrázky křížku stojících vedle názvů nahraných komponent. Po kliku na tento obrázek je vyvolána akce *deleteBundleAction()*, která zajistí smazání odpovídající komponenty, nebo akce *deleteAllBundlesAction()*, jež je vyvolána stiskem obrázku křížku zobrazeného vedle popisku *Delete all*. Tato akce smaže všechny nahrané komponenty.

Při přechodu na následující krok je zavolána funkce *showChooseLR()*, jež zajistí získání dat ze serveru potřebných pro tento krok. V odpovědi je HTML stránka obrazovky, jejíž součástí je element *script*, který obsahuje kód pro zpracování dat přijatých ve formátu *JSON*. Tento kód se vykoná po vložení přijatého HTML do DOMu aktuálně zobrazené stránky. Dále se zaregistruje událost, která obstarává označení komponent podle zvoleného ID lokálního registru ikonou řetězu.

Výběr lokálního registru

V této obrazovce je možné vybrat lokální registr, pro nějž má proběhnout porovnání nahraných komponent. Při výběru jiného ID se vyvolá událost *change*, na kterou je registrována funkce *emphasizeBundlesAction()* zajišťující aktualizaci ikon řetězu, které značí komponenty, jež jsou svázány zvoleným ID.

Při stisku tlačítka *next* se volá funkce *showResults()*, která žádá server o výsledek porovnání. Data pro pravý strom ve formátu *JSON* jsou přijata stejným způsobem jako v předchozím případě.

Data pro vytvoření levého stromu jsou již součástí HTML stránky. Po jejím načtení se

[2]http://jquery.malsup.com/form/

vytvoří ze struktury HTML seznamu (elementy ¡ul¿, ¡li¿) ovládací prvek *Tree View* pomocí pluginu *jquery.treeview*. Dále se zaregistruje *tooltip* podávající informace o symbolickém jménu po najetí kurzoru myši na vlastnost. Tento *tooltip* zajišťuje plugin pro *jQuery qTip2*[3]. Poslední činností funkce *showResults()* je registrace události kliknutí na libovolný prvek stromu.

Výsledky

Událost kliknutí na libovolný prvek stromu zajistí zobrazení dat vyfiltrovaných na základě zvoleného prvku. Tato data budou zobrazena také ve formě *Tree View*. Způsob, na jehož základě jsou data filtrována, je podrobně popsán v kapitole 5. Ve stejné kapitole jsou zmíněny i ostatní prostředky, jež byly uživateli poskytnuty ve snaze co nejvíce mu usnadnit získání relevantních výsledků.

6.5 Použití jazyku LESS pro generování CSS

Při implementaci byl využit LESS pro usnadnění práce se styly, který byl obecně popsán v části 4.2.5. Z tohoto důvodu, zde nebude popisován obecně. Překlad LESS šablon je možné provést třemi způsoby:

- dynamicky JavaScriptem na straně klienta po načtení stránky,
- dynamicky na straně serveru pomocí některé dostupné knihovny,
- a nebo staticky prostřednictvím externího překladače.

V práci je použit statický překlad souborů **.less* externím nástrojem Simpless[4]. Výhodou statického překladu je, že klient při vstupu na stránku nemusí stahovat JavaScripový soubor, který poté přeloží **.less* soubory na CSS. Výsledkem statického překladu jsou standardní CSS soubory, tudíž ve výsledku není žádná změna oproti nepoužití jazyka LESS. Nástroji Simpless stačí tyto **.less* soubory předat a on je při každé jejich změně opět přeloží.

[3] http://craigsworks.com/projects/qtip2/
[4] Nástroj Simpless je možné stáhnout na http://wearekiss.com/simpless

6.5.1 Struktura v EFP Portálu

Struktura LESS šablon je navržena hierarchicky. Na nejvyšší úrovni se nachází soubor *resources/css/main.less*, který dále importuje ostatní šablony v hierarchii. Každá integrovaná aplikace pak může obsahovat jeden nebo více LESS šablon. V případě, že je potřeba více souborů šablon pro danou aplikaci, pak je vhodné tyto šablony importovat pomocí jediného LESS souboru. Tímto vzniká snadno udržitelná hierarchie šablon.

Main.less dále importuje soubory na stejné úrovni a to *layout.less*, ve kterém jsou všechny styly pro celý *EFP Portál*. Soubor *common.less* obsahují pomocné funkce a šablony (*mixins*). Soubor *elements.less*[5] je rozšíření, které poskytuje již předdefinovanou sadu funkcí *mixins* usnadňující práci při stylování. V *elements.less* se vyskytují *mixins* převážně pro nové styly CSS3 jako například průhlednost (opacity), kulaté rohy (rounded-corners) atd.

6.6 Ověření funkčnosti

V kapitole 5 bylo dosaženo poznatku, že data poskytovaná *EFP Comparatorem* nejsou vzhledem ke své struktuře vhodná k dostatečně vypovídající prezentaci výsledků porovnání. Na základě tohoto poznatku bylo rozhodnuto, že je nutné vytvořit vlastní struktury, do kterých bude tato nevyhovující kompozice dat transformována a to takovým způsobem, že umožní prezentaci těchto dat, jež uživateli poskytne možnost se v nich snadno orientovat a co nejjednodušeji z nich získat pro něj podstatné informace.

Z náročnosti těchto transformací vyplývá riziko vzniku zkreslení poskytovaných dat. Na základě tohoto nebezpečí vznikla nutnost ověřit konzistenci převáděných dat pomocí vzájemného srovnání jejich hodnot před a po transformaci.

Srovnáním jednotlivých výsledků se zjistilo, že data nebyla transformací mezi jednotlivými strukturami žádným způsobem narušena a výsledky podávané *EFP Comparatorem* jsou v naprosté shodě s jejich vizualizací pojednávanou aplikací.

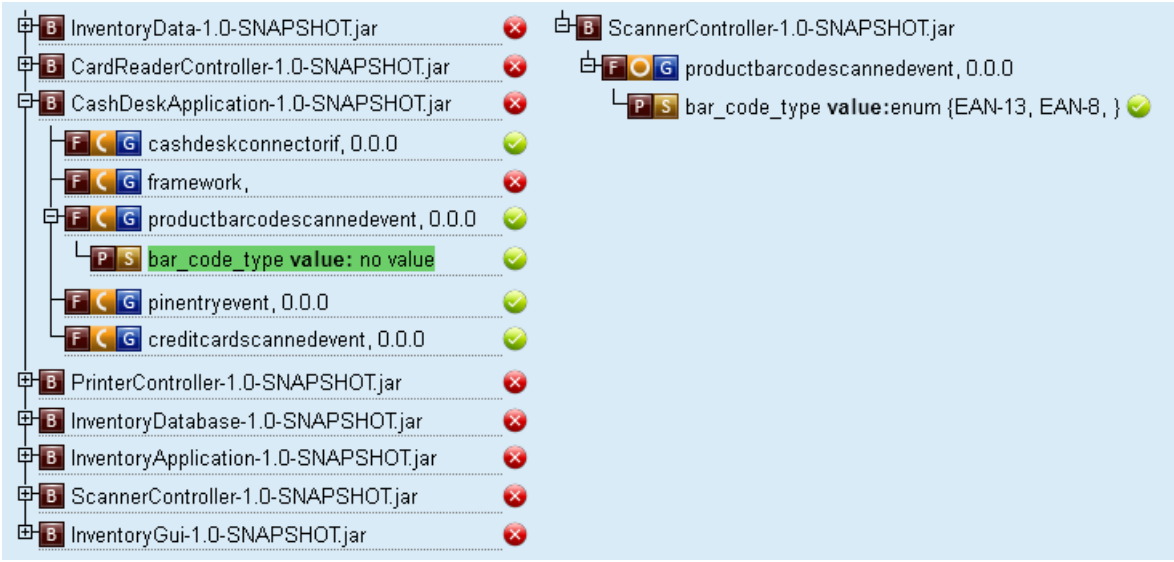
Porovnáním následujících ukázek, kde na první ukázce jsou zobrazena výstupní data *EFP Comparatoru* a ukázka 6.1 zobrazuje jejich vizualizaci provedenou pomocí ověřované aplikace, je možné vidět, že prezentované výsledky se skutečně navzájem nijak neliší. Komponenty modelu OSGi, z nichž byly vytvořeny prezentované ukázky výsledků, tvoří dohromady aplikaci *CashDesk-Inventory* nacházející se na přiloženém DVD spolu s ostatními výsledky. Uvedené výsledky odpovídají zvolenému ID lokálního registru 463.

[5]Toto rozšíření je k dispozici na webových stránkách http://lesselements.com/.

```
Components: ..\osgi\ScannerController-1.0-SNAPSHOT.jar and
            ..\osgi\CashDeskApplication-1.0-SNAPSHOT.jar

   Feature: PACKAGE.cz.zcu.kiv.osgi.example.
            cashdesk.scannercontroller.productbar-
            codescannedevent,0.0.0

     EFPs bar_code_type Evaluation Evaluation: OK
        Value on ScannerController-1.0-SNAPSHOT.jar:enum{EAN-13, EAN-8},
        Value on CashDeskApplication-1.0-SNAPSHOT.jar:null
```



Obrázek 6.1: Ukázka výsledků porovnání

Veškerá funkcionalita byla testována pouze v prohlížečích Firefox verze 8.0, Chrome verze 18.0 a Opera 11.62, kde jsou mírné obtíže se zobrazením stylů v obrazovce výsledků.

7 Závěr

Cílem této diplomové práce bylo přinést zlepšení komfortu práce s mimofunkčními charakteristikami v projektu EFP Katedry informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni. Pro dosažení tohoto cíle bylo nutné se s tímto projektem blíže seznámit. Poznatky získané tímto krokem byly zpracovány v kapitole 3.

Na základě analýzy stávajícího řešení uživatelského rozhraní aplikace *EFP Comparator* byla dle získaných poznatků navržena zlepšení, která byla modelována v nástroji *Lumzy* a schvalována zadavatelem projektu práce. Tento návrh byl ještě rozšířen o realizaci webové aplikace *EFP Portál* v souladu se smyslem této práce. Do tohoto portálu byla integrována aplikace *EFP Comparator*. Tímto byl položen základ pro snadnou integraci a jednotný přístup k dalším aplikacím EFP projektu.

Pro implementaci byl po pečlivé analýze vhodných kandidátů technologií zvolen framework *Spring MVC* pro serverovou část. Aplikační logika na straně klienta byla implementována s využitím jazyků JavaScript, HTML5 a framework jQuery.

Výsledkem této práce je průvodce procesem přípravy a vyhodnocení dat aplikace *EFP Comparator*. Silnou stránkou tohoto řešení je jeho intuitivnost a schopnost úspěšně provést i méně zkušeného uživatele celým postupem, jež je nutný pro získání výsledků porovnání. Reprezentace výsledných dat, která je uživateli poskytnuta ve formě prvků *Tree View*, umožňuje tato data filtrovat takovým způsobem, aby z nich získal pro něj relevantní informace.

Literatura

[1] LIŠKA, Vojtěch. *Porovnávání rozhraní komponent v dynamickém frameworku.* Plzeň, 2010. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[2] BACHMANN, Felix et al. *Volume II: Technical concepts of component-based software engineering.* 2000. Carnegie Mellon University, Software Engineering Institute.

[3] SZYPERSKI, Clemens et al. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[4] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.* Duben 2007, dostupná na http://www.osgi.org/.

[5] BRADA, Přemysl; LIŠKA, Vojtěch; WAJTR Břetislav. *Modelování existujících OSGi komponent.* Plzeň, 2006. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[6] BRADA, Přemysl; VALENTA, Lukáš. *The CoSi Component Model - Specification of CoSi version 2.* Technická zpráva. Plzeň, 2008. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[7] JEŽEK, Kamil; BRADA, Přemysl; ŠTĚPÁN, Petr. *Towards Context Independent Extra-functional Properties Descriptor for Components.* In Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010) , Electronic Notes in Theoretical Computer Science (ENTCS), ročník 264, Říjen 2010, ISSN 1571-0661, str. 55-71.

[8] JEŽEK, Kamil; BRADA, Přemysl. *Correct matching of components with extra-functional properties - A Framework Applicable to a Variety of Component Models.* Plzeň, 2011. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[9] JEŽEK, Kamil. *A Complex Meta-model for Extra-functional Properties Concerning Common Data Types Their Comparing and Binding.* In 2nd World Congress on Software Engineering (WCSE 2010), Volume 2, pages: 71-74, ISBN: 978-0-7695-4303-1, 2010.

[10] ŠVÁB, Jan. *Obecna reprezentace mimofunkcnch charakteristik na komponentach.* Bakalářská práce. Plzeň, 2011. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[11] VLČEK, Lukáš. *Vyhodnocování mimofunkčních charakteristik na komponentách.* Bakalářská práce. Plzeň, 2011. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[12] ŠTULC, Martin. *Uživatelské rozhraní pro úložiště mimofunkčních charakteristik.* Bakalářská práce. Plzeň, 2011. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[13] JOHNSON, Rod et al. *Professional Java Development with the Spring Framework.* Wrox, Červen 2005. ISBN 978-0-764-57483-2.

[14] SPRINGSOURCE. *Documentation.* Dostupné na: static.springsource.org/spring/docs/1.2.x/reference/introduction.html.

[15] CLARENCE, Ho; HARROP, Rob. *Pro Spring 3.* Apress, Duben 2012. ISBN13: 978-1-4302-4107-2.

[16] KUNEŠ, Daniel. *Webové frameworky.* Nepublikováno. Plzeň, 2011. Oborový projekt. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[17] W3TECHS. *Usage of JavaScript libraries for websites.* Dostupné na: http://w3techs.com/technologies/overview/javascript_library/all.

[18] ALVARO, A.;ALMEIDA, E. S. a MEIRA, S. L. *A Software Component Quality Model: A Preliminary Evaluation.* EUROMICRO '06: Procee- dings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications.

[19] FRANCH, X. *Systematic Formulation of Non-Functional Characteristics of Software.* Proceedings of International Conference on Requirements Engineering (ICRE).

Příloha A - uživatelská dokumentace

EFP Portál

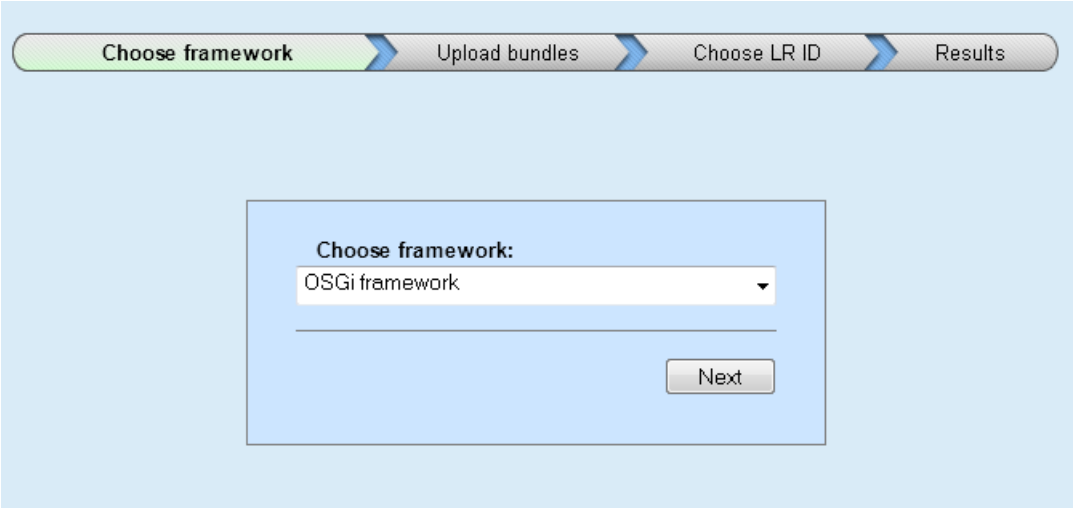
Při vstupu na EFP portál je jako první zobrazena úvodní stránka obsahující základní informace o portálu, novinky týkající se nabízených aplikací a důležité odkazy. Z menu v horní části obrazovky je možné vybrat buď *homepage* portálu nebo některou z integrovaných aplikací. V současné době je k dispozici pouze EFP Comparator.

EFP Comparator

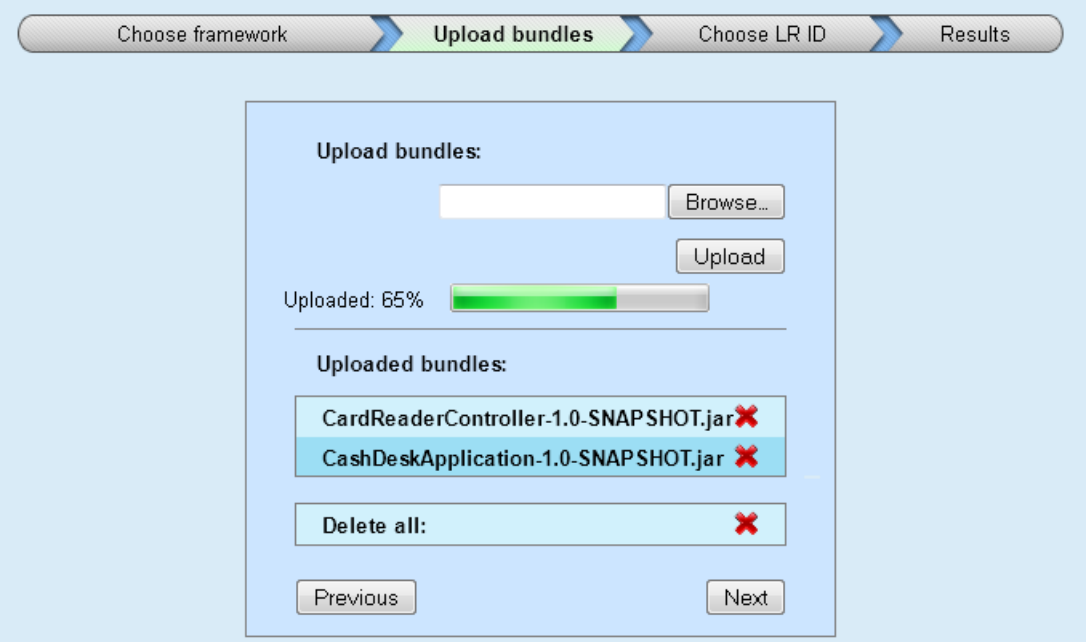
Po tom co si uživatel zvolil z menu EFP portálu možnost EFP Comparator je zobrazena první stránka průvodce aplikací (viz obrázek 7.1), kde je zobrazen název zvolené aplikace, ale hlavně navigace se všemi kroky průvodce s vyznačeným aktuálním krokem. Mezi kroky se lze samozřejmě libovolně přesouvat oběma směry pomocí tlačítek *Previous* a *Next*. V této první obrazovce je uživateli nabídnuta možnost výběru ze dvou typů frameworků a to OSGi nebo CoSi. Poté co si uživatel zvolí jeden z frameworků, může pokračovat dále stisknutím tlačítka *Next*.

Tímto se uživatel dostane na další obrazovku 7.2, jež požaduje nahrání komponent, které se budou porovnávat. Kliknutím na tlačítko *Browse* je uživateli nabídnuta volba souborů (file chooser), ve kterém může vybrat adresář, kde se nacházejí komponenty, jež chce nahrát.

Komponenta se nahraje tak, že uživatel označí jednu nebo více komponent, stiskne tlačítko *Otevřít* a poté *Upload*. Procentuální postup nahrávání komponent je zobrazen číselně i graficky u položky *Uploaded*. Takto nahrané komponenty se zobrazí v okně pod *Uploaded Bundles*. Komponenty je možné mazat jednotlivě stisknutím červeného křížku vedle názvu komponenty nebo smazat všechny nahrané komponenty najednou stisknutím křížku u položky *Delete all*. V případě, že je uživatel spokojen se svým výběrem může pokračovat na další krok stisknutím *Next*.

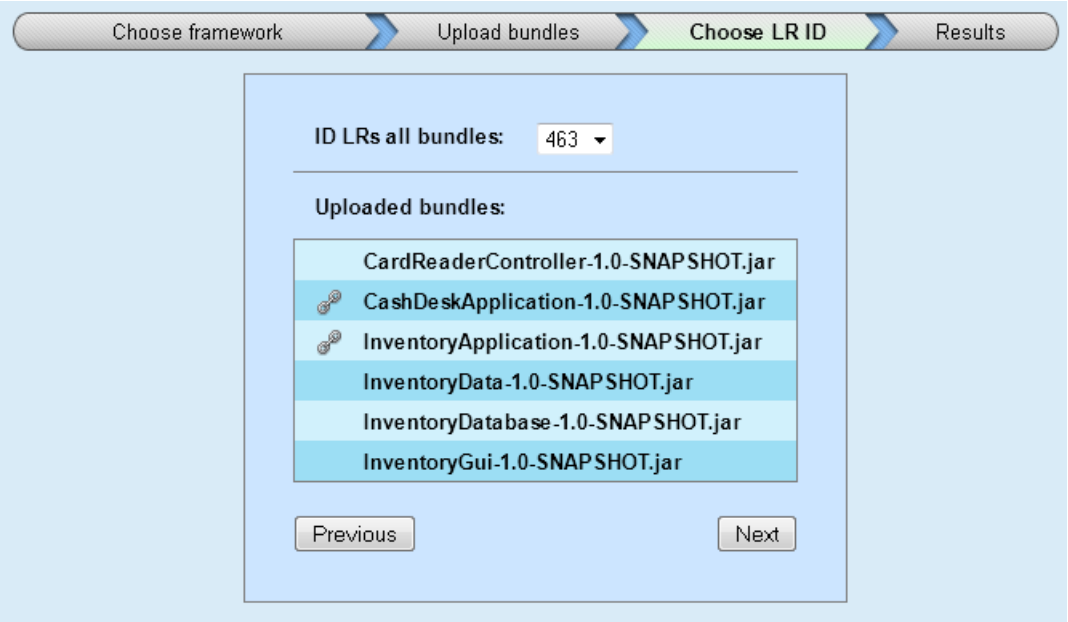


Obrázek 7.1: Krok průvodce Choose framework



Obrázek 7.2: Krok průvodce Upload bundles

Na následující obrazovce *Choose LR ID* (viz obrázek 7.3), je uživateli umožněno na základě ID lokálních registrů získaných z nahraných komponent označovat komponenty, jež sdílejí stejné ID. Položka s názvem *ID LRs all bundles* obsahuje seznam všech ID, z nichž je možné volit. Při výběru ID z tohoto seznamu jsou označeny ikonkou řetězu umístěnou vedle názvu komponenty ty z nich, jejichž ID je shodné. Poté co uživatel skončí s výběrem může opět pokračovat na další krok průvodce stisknutím tlačítka *Next*.



Obrázek 7.3: Krok průvodce Choose LR ID

Posledním krokem průvodce jsou samotné výsledky porovnání nahraných komponent. Obrazovka výsledků je pomyslnou čarou rozdělena na levou a pravou část, kde v levé části je zobrazen víceúrovňový strom všech komponent (viz obrázek 7.5). Na nejvyšší úrovni stromu jsou zobrazeny jednotlivé komponenty, poté co uživatel komponentu rozbalí kliknutím na plus nalevo od názvu komponenty zobrazí se na nižší úrovni všechny vlastnosti příslušející dané komponentě. Podobným postupem je možné se dostat k EFP patřící k dané vlastnosti.

Všechny položky stromu jsou pro zvýšení čitelnosti dat opatřeny grafickými symboly, jež je jednoznačným způsobem identifikují, symbolicky je sdělena také informace o výsledku porovnání. Podávané informace jsou ještě rozšířeny pomocí *Tooltipu*, jež je zobrazen po najetí kurzorem myši nad vlastnost dané komponenty, v tomto *Tooltipu* je zobrazeno symbolické jméno vlastnosti. Následuje výčet symbolů, které jsou vidět na obrázku 7.4 včetně ozřejmění jejich významu.

(A) označuje komponentu.

(B) označuje vlastnost komponenty.

Obrázek 7.4: Použité symboly

(C) označuje EFP příslušející dané vlastnosti.

(D) udává jednoduchý typ EFP.

(E) udává složený typ EFP.

(F) zobrazuje, že daná vlastnost je komponentou poskytována.

(G) zobrazuje, že daná vlastnost je komponentou vyžadována.

(H) udává typ komponenty *EVENT*.

(I) udává typ komponenty *INTERFACE*.

(J) udává typ komponenty *TYPE*.

(K) udává typ komponenty *PACKAGE*.

(L) označuje výsledek porovnání nekompatibilních hodnot.

(M) označuje chybějící EFP.

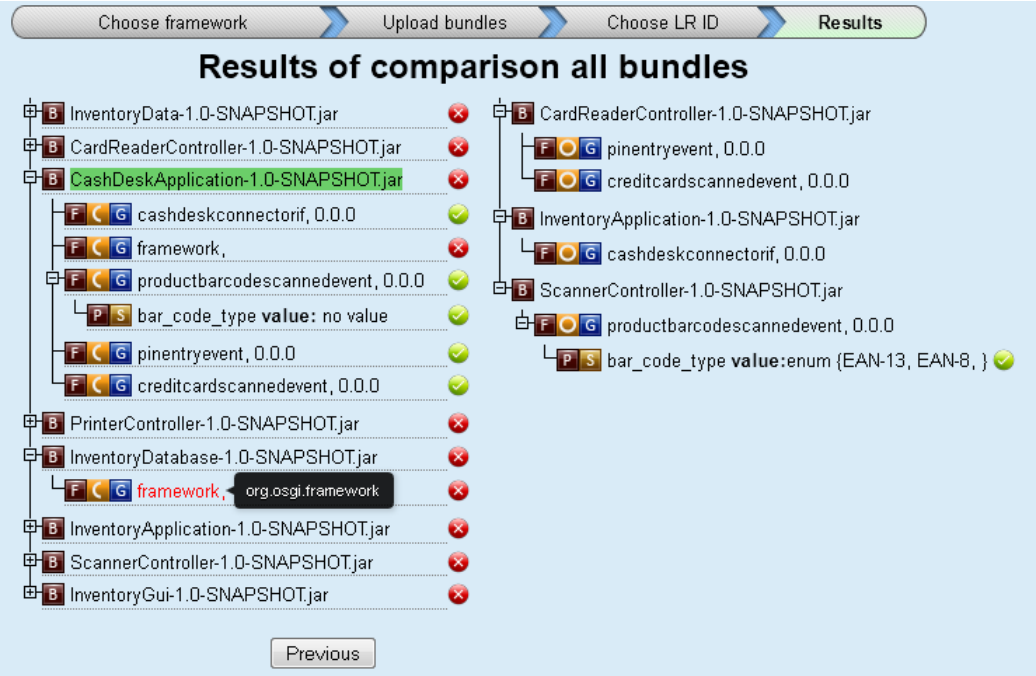
(N) označuje výsledek porovnání kompatibilních hodnot.

(O) označuje EFP, jež nemohlo být porovnáno s odpovídajícím protějškem, neboť se tento v dané vlastnosti nenachází.

Zmíněná pravá část obrazovky výsledků je určena k zobrazení informací filtrovaných na základě vztahů mezi prvky vybranými uživatelem v levém stromě. Možností výběru dat je vícero. Pokaždé, když je v levé části zvolen nějaký prvek, je označen podbarvením, což přispívá ke zlepšení orientace ve výsledcích. První z možností je označení samotné komponenty (viz obrázek 7.5), v tom případě je v pravé části obrazovky zobrazena komponenta či komponenty sdílející stejné vlastnosti. Pod těmito vlastnostmi jsou zobrazeny všechny jim přiřazené EFP.

Pravidlem je, že v levé části jsou komponenty tuto vlastnost vyžadující, zatímco v pravé jsou zobrazeny komponenty, jež danou vlastnost poskytují. Druhá z nabízených možností je výběr vlastnosti. Pokud je uživatelem zvolena vlastnost, je opět v pravé části

zobrazena komponenta tuto vlastnost poskytující, kde je pro zlepšení přehlednosti filtrovaných výsledků zobrazena pouze s vlastností vybranou na levé straně. Tato vlastnost je zobrazena včetně všech EFP k ní přiřazených. Poslední z možností filtrování výsledků porovnání je vybrání EFP. V tom případě je v pravé části zobrazena komponenta, jež obsahuje vlastnost se zvoleným EFP. Pod komponentou je zobrazena pouze vlastnost a EFP odpovídající výběru v levém stromě.



Obrázek 7.5: Krok průvodce Results

Nasazení aplikace

Pro nasazení aplikace je nutné mít nainstalované běhové prostředí Javy verze 1.6. Dále nainstalovaný a nakonfigurovaný nástroj Maven 2, který je možné stáhnout na adrese: http://maven.apache.org/. Na přiloženém DVD je k dispozici přeložená verze celého projektu. Zdrojové soubory všech částí projektu EFP včetně vytvořeného portálu. V adresáři *results* jsou uloženy výsledky porovnání testovacích komponent (umístěných v adresáři *test_data*) v textové podobě získané z konzolového výstupu původního rozhraní.

Nejprve je nutné se na přiloženém DVD přesunout do adresáře *project_efp_sources\efp-Portal*. *EFP Portál* bude nasazen a spuštěn zadáním příkazu `mvn tomcat:run` do příkazové řádky. Po tomto kroku bude webová stránka dostupná z URL *http://localhost:8080/efp-Portal*. Testovací soubory jsou umístěny v adresáři `test_data`. Veškerá funkcionalita byla testována pouze v prohlížečích Firefox verze 8.0, Chrome verze 18.0 a Opera 11.62, kde jsou

mírné obtíže se zobrazením stylů v obrazovce výsledků. Výchozí nastavení cesty k úložišti uživatelem nahrávaných komponent je d:\efp_portal\storage\.