Sem vložte zadání Vaší práce.

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

Diplomová práce

Systém pro párování nabídek a poptávek postavený na čistém objektovém paradigma

Bc. Michal Balda

Vedoucí práce: Ing. Robert Pergl, Ph.D.

29. dubna 2015

Poděkování

Nejdříve chci poděkovat vedoucímu práce, Ing. Robertu Perglovi, Ph.D., který ve mně vzbudil zájem o Smalltalk a platformu Pharo. Dále chci poděkovat své rodině za podporu během celého studia.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na základě níž se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 29. dubna 2015 .....

Odkaz na tuto práci

Balda, Michal. *Systém pro párování nabídek a poptávek postavený na čistém objektovém paradigma.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato práce se zabývá analýzou, návrhem, implementací a otestováním portálu, který zjednoduší proces hledání a přihlašování pracovníků na krátkodobé práce. K vývoji používá čistě objektovou platformu Pharo Smalltalk a webový framework Seaside, důraz je kladen zejména na kvalitu objektového návrhu. Verifikace návrhu je provedena pomocí sady testů. Nezanedbatelnou roli u webových aplikací hraje také bezpečnost a výkonnost, i tyto aspekty jsou prověřeny. Čistě objektový přístup nepředstavuje jediný způsob vývoje, použitá platforma je v závěru porovnána s procesním přístupem platformy Liferay.

Klíčová slova Smalltalk, Pharo, Seaside, objektově-orientované paradigma, párování nabídek a poptávek

Abstract

The main task of this thesis is to analyse, design, implement and test a portal that should simplify the process of seeking workers for short-term jobs and managing their applications. Pharo, a clean object-oriented platform, and Seaside, its main web framework, are used to accomplish this task. A high-quality design is a must, therefore a multiple-aspect verification follows the design and implementation phases. Being a web application, the portal needs to be as secure as possible and adequately responsive. Those two aspects are included in the tests as well. Finally, since the object-oriented paradigm is not the only one possible, the platform is compared to Liferay, a more widely used process-oriented platform.

Keywords Smalltalk, Pharo, Seaside, object-oriented paradigm, offer and inquiry matching

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Krátkodobé práce (brigády) jsou bezesporu rozšířeným způsobem výdělku mezi mladými studenty. Hledání vhodné práce ale patří mezi nelehké úkoly. Představy pracovníků se různí, stejně jako požadavky zaměstnavatelů. Ve velkých městech, kde nabídek bývá nepřeberné množství, není v lidských silách všechny procházet. Proto vznikly portály, které nabídky sdružují na jednom místě a do určité míry pomáhají studentům množství nabídek filtrovat a vybrat tu vhodnou. Tím zcela jistě ušetří hledajícím spoustu času. Ne však zaměstnavatelům, jimž na bedrech zůstane stále stejný počet činností. Musí nabídku zveřejnit, udržovat si přehled o obsazenosti pozic, každému zájemci odpovědět, vyžádat od něj potřebné informace, pozvat na případný pohovor, vybrat vhodné kandidáty, domluvit s nimi detaily nástupu, řešit, když některý z nich nakonec odmítne nebo domluvenou spolupráci zruší, a mnoho dalšího. V těchto činnostech obvykle portály příliš nepomohou.

Zjednodušení některých uvedených činností by pro zaměstnavatele bylo jistě přínosné a mohlo by dokonce vyústit ve větší ochotu nabízet krátkodobé práce. Proto bylo rozhodnuto o tvorbě webového portálu, který se bude snažit proces hledání pracovníků a nabídek práce co nejvíce usnadnit a zautomatizovat pro obě zúčastněné strany. Díky tomu se proces výrazně zrychlí a bude možné shánět pracovníky i na tak zvané „last minute“ nabídky, které je potřeba obsadit co nejrychleji, v řádu dnů nebo dokonce hodin.

Kapitola 1

Cíl práce

Prvním cílem práce je navrhnout, implementovat a otestovat aplikaci sloužící k hledání pracovníků na krátkodobé práce (brigády) s využitím platformy Pharo Smalltalk a webového frameworku Seaside. Platforma se vyznačuje čistým objektovým návrhem, hlavním požadavkem je proto maximální využití čistého objektového paradigmatu. Druhým cílem je výslednou aplikaci porovnat s *Portálem spolupráce s průmyslem*, provozovaným na Fakultě informačních technologií a vytvořeným pomocí procesního přístupu na platformě Liferay.

Práce začíná analýzou v kapitole 2. Ta představuje základní vizi aplikace, detailně specifikuje požadavky a blíže popisuje platformu Pharo Smalltalk a framework Seaside.

Z provedené analýzy vychází kompletní návrh aplikace, popsaný v kapitole 3. Navrženy jsou všechny části aplikace tak, aby respektovaly specifikované požadavky.

Následuje kapitola 4 o implementaci, tedy převedení návrhu do funkční podoby.

Výstupy z předchozích kapitol jsou podrobeny různorodým testům, jež mají za úkol ověřit správnost a čistotu návrhu i funkčnost implementace. O výsledcích informuje kapitola 5.

O procesu a podmínkách nasazení aplikace do reálného provozu pojednává kapitola 6.

Kapitola 7 porovnává z různých hledisek čistě objektový a procesní přístup za pomoci výsledné aplikace a již existujícího Portálu spolupráce s průmyslem.

Kapitola 2

Analýza aplikace

2.1 Vize aplikace

Základní úlohou aplikace bude zjednodušit a zrychlit proces hledání pracovníků na krátkodobé práce (brigády). Díky komplexním možnostem párování bude pracovníkům předkládat pouze ty nabídky, které jsou pro ně relevantní. Zadavatelům naopak nabídne vhodně seřazené kandidáty spolu s jednoduše použitelným a částečně automatizovaným procesem schvalování a potvrzování kandidátů.

2.2 Specifikace požadavků

V této části jsou podrobně vypsány požadavky na aplikaci. Většina z nich byla stanovena při konzultacích s externím zadavatelem projektu, společností Daylight s.r.o. Požadavky se dají rozdělit do kategorií podle jejich vztahu k aplikaci. První kategorií jsou funkční požadavky, stanovující hranice schopností celé aplikace z uživatelského pohledu. Druhou kategorií jsou nefunkční požadavky, týkající se aplikace jako celku a popisující na příklad požadovanou výkonnost, škálovatelnost nebo bezpečnost. Třetí kategorií jsou požadavky na uživatelské rozhraní, jelikož se bude jednat primárně o webovou aplikaci a všichni uživatelé budou s aplikací komunikovat právě skrze něj. Poslední kategorií jsou ostatní požadavky, které mohou specifikovat mimo jiné zákony a jiné legislativní požadavky, které aplikace musí dodržovat.

2.2.1 Funkční požadavky

2.2.1.1 Uživatelé

Kromě role hosta (anonymního, nepřihlášeného uživatele) aplikace počítá se třemi rolemi registrovaných uživatelů: pracovníkem (brigádníkem), zadavatelem práce (společností) a administrátorem. Všichni se přihlašují pomocí zvo-

leného přihlašovacího jména (e-mailu v případě pracovníků a administrátorů) a hesla, pracovníci se navíc mohou přihlašovat svým účtem na sociálních sítích a jiných externích službách, jako jsou Facebook nebo Google. V případě přihlašování kombinací e-mailu a hesla by měl uživatel mít možnost obnovit si zapomenuté heslo za předpokladu, že neztratil přístup ke svému e-mailu.

Pracovník po registraci projde procesem aktivace profilu a nastavení filtru. V několika krocích vyplní informace o sobě, týkající se oblastí jako vzdělání, předchozích pracovních zkušeností, schopností a dovedností, postavy a osobních údajů. Přesný seznam včetně jednotlivých možností bude dodán zadavatelem. Dále vyplní preference ohledně hledané práce, jako je lokalita nebo druh pracoviště. Nakonec si může upravit svou dostupnost v kalendáři, což znamená zadat dny a hodiny, kdy není dostupný. Všechny zmíněné informace může později kdykoliv upravit.

Zadavatel práce může svůj profil vyplnit nebo upravit kdykoliv a má prostor pro představení, případně nahrání loga a dalších fotek, aby zapůsobil na potenciální zájemce. Dobře vyplněný profil bude jedním z vedlejších kritérií při hodnocení a řazení společností a jejich nabídek.

2.2.1.2 Nabídky práce

Zadavatel do systému vkládá nové nabídky. Při jejich vytváření vyplní počet hledaných pracovníků podle pohlaví, případně bez rozlišení. Dále zadá druh pracoviště (pozici), lokalitu, termín a cenu (hodinovou sazbu i celkový výdělek). Poté může specifikovat požadavky na uchazeče, kde má k dispozici stejné volby jako pracovník při vyplňování svého profilu. Nakonec může k nabídce dopsat další doplňující informace nebo přiložit fotky.

Pokud vkládá více stejných nebo podobných nabídek, na příklad na různé termíny, s různým počtem pracovníků nebo různým výdělkem, nemusí vyplňovat všechny požadavky znovu, ale použije libovolnou předchozí nabídku jako šablonu s tím, že pouze upraví odlišné údaje.

Zadané požadavky se využijí k párování nabídek a pracovníků. Pracovníkovi se zobrazí pouze nabídky, jejichž požadavkům podle svého profilu vyhovuje. Filtr lokality a druhu pracoviště však může dočasně pozměnit a nechat si tak zobrazit nabídky z jiných oblastí. Když ho některá nabídka zaujme, může zaslat běžnou nebo závaznou žádost, případně společnost kontaktovat kvůli doplnění informací. Po zaslání žádosti se společnosti zobrazí v seznamu kandidátů, z nichž může pozice obsazovat. Pokud společnost přijme uchazeče na jednu z pozic, má pracovník v případě běžné žádosti ještě určitý čas na zpětné potvrzení nebo odmítnutí. Pokud zaslal závaznou žádost, čas na rozmyšlenou již nedostane a spolupráce je definitivně potvrzena. Společnost může na pozice obsadit i náhradníky, kteří budou automaticky dosazeni na pozici v případě, že předchozí uchazeč nakonec práci odmítne.

Pro zrychlení procesu bude existovat ještě tzv. neomezená volba, kdy společnost vybere a předschválí více uchazečů. Všichni z nich dostanou možnost

nabídku zpětně potvrdit a ten, který ji potvrdí jako první, pozici získá. Pokud společnost hledá více pracovníků, pak práci získá prvních tolik zájemců, kolik zbývá volných míst. Tato funkce bude využitelná zejména pro nabídky práce, které je potřeba na poslední chvíli rychle zaplnit.

2.2.1.3 Administrační rozhraní

Administrátor by měl mít k dispozici několik speciálních sekcí, kde může:

- procházet, vyhledávat a upravovat pracovníky a firmy,
- procházet, vyhledávat a upravovat nabídky,
- přidávat a odebírat možné hodnoty u voleb v profilu pracovníka,
- měnit váhy a koeficienty jednotlivých složek řazení pracovníků a nabídek,
- prohlížet statistiky pracovníků, společností a nabídek.

2.2.2 Nefunkční požadavky

Aplikace bude vyvinuta na platformě Pharo Smalltalk [37] za pomoci webového frameworku Seaside [6]. Požadavky na jiné knihovny nebyly stanoveny. Nepředpokládá se propojení s žádnou další aplikací, pouze s webovou službou Administrativního registru ekonomických subjektů (ARES) Ministerstva financí České republiky [26] pro zjišťování údajů o společnostech na základě jejich identifikačního čísla osoby.

Aplikace jakožto veřejně dostupná webová aplikace musí být imunní vůči známým formám útoků běžných pro tento druh aplikací, aby bylo možné a bezpečné ji takto veřejně provozovat. Kritická je i možnost rychle reagovat na nově objevené zranitelnosti.

Požadavky na výkonnost a škálovatelnost nebyly předem stanoveny v konkrétních číslech, proto bude nutné u výsledné aplikace provést testy výkonu a na jejich základě rozhodnout o dalším postupu, na příklad nutnosti optimalizace některých částí za cenu složitějších algoritmů nebo denormalizace datových struktur.

2.2.3 Požadavky na uživatelské rozhraní

Aplikace bude pro koncové uživatele i administrátory dostupná pouze skrz webové rozhraní. To musí být plně funkční ve všech běžně používaných verzích klientů, jako Microsoft Internet Explorer [25], Mozilla Firefox [30], Google Chrome [14], Opera [32], Apple Safari [2] a dalších, a to jak na stolních počítačích a laptopech, tak na mobilních zařízeních, jako jsou mobilní telefony a tablety. Zároveň by rozhraní mělo splňovat obecná pravidla použitelnosti a očekávání uživatelů v dnešní době.

Grafický návrh rozhraní bude dodán zadavatelem. Úkolem tedy bude pouze převést návrh do funkčního rozhraní odpovídajícího výše zmíněným požadavkům.

2.2.4 Ostatní požadavky

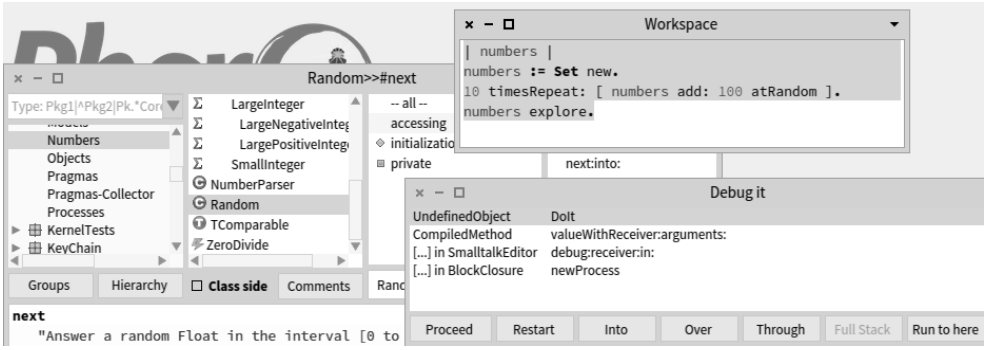
Jelikož bude aplikace zpracovávat i citlivé osobní údaje uživatelů, na příklad bydliště nebo rodné číslo, je nutné stanovit podmínky využívání, kde bude přesně vymezen rozsah získávaných dat a způsob jejich uložení a použití. Každý uživatel pak bude muset tyto podmínky přečíst a souhlasit s nimi předtím, než dané údaje sdělí, jinak mu nebude umožněn vstup do aplikace. Také je nutné, aby aplikace způsobem nakládání s daty splňovala platnou legislativu České republiky. Konformita bude zkontrolována spolu se zadavatelem práce před nasazením aplikace do provozu, proces kontroly patří mimo rozsah této práce.

2.3 Popis platformy Pharo Smalltalk

Pharo je dynamický jazyk a zároveň prostředí vycházející z původní implementace systému Smalltalk-80 [4]. Vzniklo v roce 2008 jako odnož systému Squeak [4]. Pharo se snaží celé prostředí zmodernizovat a vyčistit, i když tím někdy poruší zpětnou kompatibilitu [4]. Jazyk Smalltalk je čistě objektový, dynamicky typovaný a využívá automatickou správu paměti, zároveň je velice jednoduchý, využívá jen několik málo konceptů a minimalistickou syntaxi. Objekty představují vše, co se v systému vyskytuje, celý systém je tak plně reflektivní a transparentní. Všechny komponenty lze za chodu zkoumat a měnit, což přispívá k celkovému porozumění systému a následně i lepší kvalitě kódu.

Mezi komponenty implementované uvnitř systému patří kompilátor, základní chování objektů i tříd, veškeré datové typy, celé grafické prostředí, vývojářské nástroje, procesy včetně plánovače procesoru a mnoho dalších. Dokonce i virtuální stroj pohánějící celý svět je z větší části napsán v podmnožině jazyka Smalltalk, otestován uvnitř systému a poté automaticky převeden do jazyka C pro zvýšení výkonu [28]. Obrázek prostředí Pharo zachycuje obrázek 2.1.

Celá distribuce sestává ze dvou částí: obrazu pro virtuální stroj a virtuálního stroje samotného. Obraz je přenositelný mezi platformami a obsahuje veškeré objekty v systému. Virtuální stroj se snaží být co nejtenčí, a tak zajišťuje pouze běh uzavřeného objektového světa v obrazu a jeho komunikaci s okolním světem a externími knihovnami, případně reimplementuje část funkčnosti kvůli výkonu.



Obrázek 2.1: Ukázka prostředí platformy Pharo Smalltalk, zdroj: hlavní webová stránka platformy [37] [převzato 2015-04-02]

2.4 Popis frameworku Seaside

Seaside je framework pro tvorbu webových aplikací v jazyce Smalltalk. Seaside běží na několika implementacích, Pharo je označeno za referenční implementaci [4], vývoj probíhá právě na této platformě a poté je Seaside portován do jiných implementací [9]. Framework vznikl v roce 2002 [12] a na rozdíl od jiných webových frameworků se od začátku snaží „jít proti proudu“ odmítáním některých běžně rozšířených konvencí v oblasti webových aplikací [9].

Tou první je použití unikátních a permanentních adres pro identifikaci zdrojů. Seaside je však od základu stavově orientovaný, a proto využívá adresu k označení stavu namísto zdroje [9]. Silná stavovost přináší mnoho výhod, ale i řadu nevýhod. Hlavní výhodou je abstrakce nad bezestavovým protokolem HTTP. Sám framework automaticky řeší serializaci a aktualizaci stavu aplikace v cyklu HTTP požadavku a odpovědi, aplikace je od tohoto problému odstíněna. Automatická je i synchronizace stavu s klientem, pokud dojde k manipulaci stavu na straně klienta, na příklad použitím funkce historie. Další výhodou je, že sezení a všechny související objekty zůstávají vytvořené a uložené v paměti po celou dobu interakce, není nutné je serializovat a obnovovat při každém novém požadavku jako u některých jiných technologií, mimo jiné PHP. Díky tomu dosahuje dobrého výkonu, mnohem vyššího než zmíněné PHP. Nevýhodou je samozřejmě nemožnost odkazování na konkrétní stránku nebo vytvoření záložky, což v důsledku vylučuje indexaci vyhledávači. V aplikaci, jejíž většina je přístupná pouze po přihlášení, to příliš nevadí, nicméně lze tento problém vyřešit částečným omezením stavovosti. Další nevýhodou je nutnost podporovat cookies pro uchování identifikátoru sezení vždy, i když by na některých statických stránkách nebyl potřeba. I to se stejně jako odkazovatelné adresy vyřeší omezením stavovosti na těchto stránkách.

Druhou konvencí, proti které Seaside bojuje, je použití šablon pro oddělení prezentační a pohledové vrstvy [9], jako je tomu u frameworků využívajících vzor Model–View–Controller (MVC) nebo Model–View–Presenter (MVP). Se-

aside argumentuje tím, že šablony, obsahující zejména HTML kód stránek, v dnešní době slouží pouze k definici základní struktury podle pravidel sémantiky a nenese téměř žádnou přidanou informaci [5]. Většina práce s určením vzhledu pak zůstane doménou kaskádových stylů (CSS), které jsou od šablon odděleny. Pohled je tedy určen zejména použitými kaskádovými styly, nikoliv strukturou HTML, tudíž Seaside chápe šablony spíše jako součást prezentační vrstvy místo pohledové. Obvyklým argumentem pro oddělení šablon do samostatné vrstvy je čistota kódu, protože by nemělo dojít k míchání dvou různých technologií: HTML kódu a prezentačního kódu v jiném jazyce, zde ve Smalltalku. Seaside tento bod řeší vlastním rozhraním pro stavbu stránky korespondující se strukturou HTML, díky čemuž prezentační vrstva neobsahuje ani řádek HTML kódu.

Seaside nabízí i mnoho funkcionality ke zjednodušení vývoje: podporu budování REST rozhraní, integraci s JavaScriptovými knihovnami jQuery a Scriptaculous, rozhraní pro asynchronní požadavky (AJAX). Také obsahuje vývojové nástroje přímo ve stránkách, jako lištu s užitečnými informacemi a funkcemi, prohlížeč objektů a celého systému nebo jednoduchý debugger. Je tak možné části aplikace vyvíjet přímo ve webovém klientu, bez původního vývojového prostředí. Nenabízí však takový komfort a všechny funkce, tudíž se hodí pouze pro menší změny, případně vývoj ve vzdáleném systému, běžícím na serveru bez grafického prostředí.

Kapitola 3

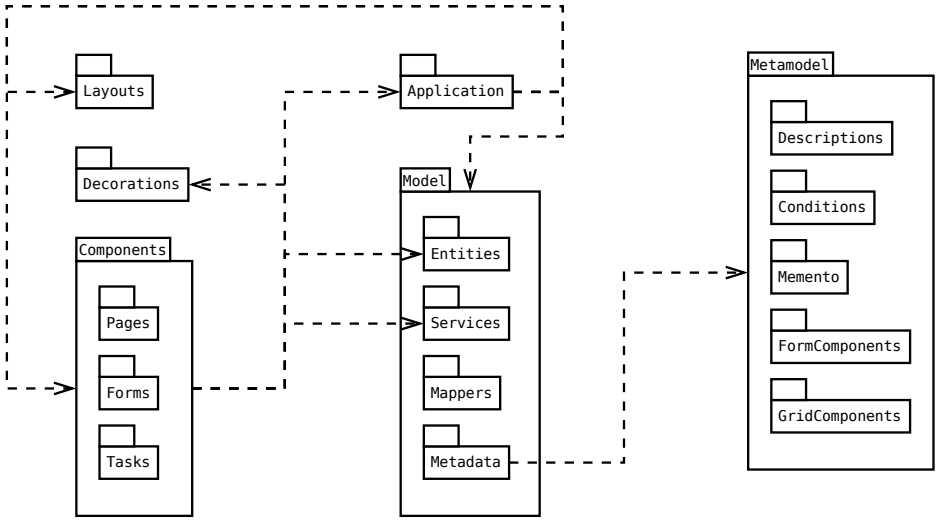
Návrh aplikace

3.1 Architektura aplikace

Základní architektura aplikace je z větší části již předurčena stanovenými požadavky: bude se jednat o webovou aplikaci vytvořenou v čistě objektovém frameworku Seaside na platformě Pharo Smalltalk. Webové aplikace se obecně skládají ze dvou hlavních částí, klientské a serverové, přičemž klientská část je díky frameworku z velké části generována částí serverovou. Účelem klientské části je prezentovat data uživatelům, reagovat na jejich vstupy a odesílat požadavky ke zpracování serverové části. Serverová část se stará o udržování dat, provádění veškerých operací nad nimi a předávání výsledků zpět klientské části.

Serverová část je složena z více samostatných celků, většina z nich je navržena pro platformu Pharo Smalltalk a tvoří jádro aplikace žijící pohromadě v jednom obrazu virtuálního stroje. Diagram balíčků jádra aplikace a jejich závislostí zachycuje obrázek 3.1. Závislosti uvnitř větších balíčků pro přehlednost nejsou zobrazeny a budou vysvětleny níže, v odpovídajících sekcích.

Ve zbytku kapitoly je návrh rozveden do detailů. Na vhodných místech je připojen UML diagram tříd pro grafické znázornění struktury. Atributy označené v těchto diagramech jako veřejné (pomocí znaku „+“) jsou ve skutečnosti definovány jako privátní, ale třída obsahuje metody pro přístup k atributu a případně jeho změnu. Tento bod je důležité zmínit, jelikož jazyk Smalltalk je založený na zapouzdření a veškerý stav objektu je tudíž privátní. Mimo to obsahuje platforma Pharo koncept takzvaných *traits*, což jsou menší jednotky pro definici a znovupoužití sady chování než třídy [41]. Ty jsou v diagramech modelovány jako abstraktní třídy se stereotypem „trait“, kompozice do tříd je vyjádřena notací dědění.



Obrázek 3.1: Diagram balíčků jádra aplikace

3.2 Klientská část

Jelikož u klientské části je velice důležitá kompatibilita napříč spektrem používaných webových klientů (obvykle prohlížečů), jsou pro její budování využity standardizované anebo široce podporované technologie. Strukturu stránky tak určuje kód v jazyce HTML podle doporučení organizace W3C pro verzi HTML5 z roku 2014 [46], nicméně téměř všechen kód je zpětně kompatibilní s předchozími verzemi HTML. Správnost kódu, tedy validitu podle zmíněného doporučení, zajišťuje sám framework Seaside za předpokladu, že vývojář dodrží několik pravidel, mimo jiné pro vnořování jednotlivých elementů. HTML by také mělo být použito pouze pro určení struktury a obsahu stránky, nikoliv pro definici vzhledu.

Vzhled stránek poté zůstane doménou kaskádových stylů (CSS) ve verzích 2.1 [45] a 3 (specifikace ještě nejsou dokončeny). Prvky ze starší verze jsou podporovány naprostou většinou dnešních webových klientů a mohou být použity téměř bez ohledu na zpětnou kompatibilitu. Naopak novější, třetí verzi kaskádových stylů stále plně nepodporují někteří široce používaní klienti. Při uplatnění nových funkcí CSS je nutné brát ohled na použitelnost v těchto klientech, na příklad pomocí metody postupného vylepšování [42]. Při něm je základní verze stránek, plně funkční ve všech klientech, obohacována o nové prvky pro klienty, kteří je podporují. Uživateli je tak předložena nejlepší možná funkcionalita v závislosti na možnostech a omezeních jeho klienta. Uživatelé s limitovanými klienty ale stále můžou stránky bez problémů využívat, byť je čeká částečně ochuzená zkušenost.

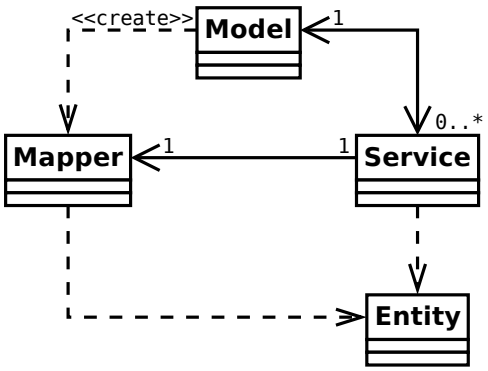
V neposlední řadě je vhodné využít skriptovací jazyk JavaScript. Obohacení stránek vhodnými skripty zajistí interaktivitu a zkrátí dobu odezvy některých úkonů. Není totiž nutné při každé akci odesílat požadavek a přijímat novou stránku ze serverové části. Vyřízení jednoho druhu událostí zajistí sám webový klient bez jakékoliv další komunikace a vyřízení druhého druhu bude vyžadovat jen jednoduchou interakci mezi klientem a serverem, bez nutnosti stahovat a překreslovat celou stránku. Podobně jako u kaskádových stylů je i u interaktivity brán ohled na uživatele, kteří nemohou tyto funkce využívat, díky metodě postupného obohacování. Stále se totiž jedná o nezanedbatelnou část internetové populace (přibližně jedno procento uživatelů [15] [47]). Nepatří do ní však jen uživatelé s klienty zcela nepodporujícími JavaScript, ale i uživatelé se záměrně vypnutým skriptováním z bezpečnostních nebo jiných důvodů.

Vytvoření uživatelského rozhraní a vzhledu stránek patří částečně mimo rozsah této práce, návrh vzhledu byl vytvořen externím grafikem již před zahájením prací. Jediným úkolem proto bylo převést zadaný návrh nejprve na statickou stránku využívající značkovací jazyk HTML a kaskádové styly, následně převést tuto reprezentaci do komponent frameworku Seaside.

3.3 Model

Modelová vrstva zastřešuje jednak data aplikace, jednak její logiku. Je soběstačná a nijak nezávisí na prezentační vrstvě. To představuje nutnou podmínku k fungování aplikace s více různými prezentačními vrstvami, kupříkladu webovým rozhraním, zjednodušeným webovým rozhraním pro mobilní zařízení nebo samostatnou mobilní aplikací. Přestože se s více prezentačními vrstvami s ohledem na specifikaci požadavků nepočítá, nezávislá modelová vrstva značí důležitý krok na cestě k čistému návrhu. Prezentační vrstva totiž přímo závisí na modelové vrstvě a zavedení cyklické závislosti přidáním opačného směru by v budoucnu přineslo velké problémy při údržbě nebo rozšiřování aplikace. Bylo by téměř nemožné změnit podstatnou část jedné vrstvy, aniž bychom museli upravit i druhou vrstvu.

Ani samotná modelová vrstva není jednolitá, podle odpovědností ji lze dále rozdělit na více menších vrstev, přičemž pro prezentační vrstvu jsou některé z nich skryté. Základním prvkem modelu jsou *entity* (entities), které reprezentují jednotlivé objekty problémové domény. Pro perzistenci entit se používají *mappery* (mappers), jež představují rozhraní mezi ostatními vrstvami aplikace a *úložištěm* (storage), na příklad databází. Mappery však poskytují pouze základní rozhraní a neprovádí žádnou kontrolu validity entit nebo platnosti akcí, proto jsou pro zbytek aplikace překryté *službami* (services). Ty rozšiřují rozhraní o složitější akce, starají se o atomicitu a konkurenci a kontrolují platnost prováděných akcí. Soustavu částí modelové vrstvy završuje *správce modelu*, který se stará o inicializaci a propojení všech ostatních vrstev. Závis-



Obrázek 3.2: Diagram tříd zachycující vztahy vrstev modelu

losti jednotlivých vrstev přibližuje obrázek 3.2, s výjimkou úložišť, která jsou velmi různorodá a nelze je představit univerzálním způsobem. Třída *Model* reprezentuje správce modelu.

3.3.1 Entity

Entita je základním nositelem dat napříč aplikací. Každá entitní třída představuje jeden druh objektu z modelované domény. Jejich atributy mohou být buď atomické, tedy obsahující hodnoty základních datových typů jako číslo, řetězec, pravdivostní hodnota a časový údaj, nebo složené. V tom případě odkazují na instance jiných entitních tříd a hodnotou atributu je buď jeden ukazatel (modelující vztah typu x:1), nebo kolekce ukazatelů (modelující vztah typu x:N). Pokud je vztah nepovinný, je hodnotou u vztahu x:1 speciální ukazatel *nil*, v případě vztahu x:N jen prázdná kolekce. Označení vztahu x:1 a x:N záměrně nespecifikuje kardinalitu zdrojové entity, jelikož ta je z jejího pohledu neznámá a modeluje ji protější (cílová) entita.

V entitách není obsažena logika aplikace, entity poskytují pouze rozhraní pro přístup k hodnotám atributů a jejich změně, případně jednoduché fasády nad tímto rozhraním. Entity neobsahují ani složitější pravidla k jejich validaci. Ta jsou modelována pomocí metamodelu a jsou oddělená od entit ze dvou důvodů. Prvním důvodem je, že umístění validačních pravidel přímo do entit by vytvořilo závislosti atributů na jiných atributech, závislosti entit na jiných vrstvách modelu kvůli validaci platnosti ukazatelů a v neposlední řadě také závislost entit na prezentační vrstvě, protože povolený rozsah hodnot může u některých atributů záviset i na právě přihlášeném uživateli. Druhým důvodem pro oddělení pravidel je snadnější práce s entitami při jejich vytváření nebo hromadné změně atributů, kdy může chvilkově dojít k porušení validačních pravidel, aby před skončením transakce byla entita opět uvedena do

platného stavu. Metamodelu a validačním pravidlům se dále věnuje kapitola 3.5.

Přestože problémová doména není příliš široká a základních entitních tříd je jen několik, z důvodu většího množství atributů a vztahů vyústil proces modelování v poměrně rozsáhlou strukturu entitních tříd. Mezi základní entitní třídy patří uživatel (*User*) a nabídka práce (*Job*). Při prvním příchodu na portál dostane návštěvník roli anonymního uživatele (*GuestUser*). Po registraci se z něj stává jeden z registrovaných uživatelů (*RegisteredUser*), zejména pracovník (*Worker*) nebo zadavatel práce (*Employer*), existují však i další speciální role, jako je administrátor (*Administrator*). Hlavní činností pracovníka je vybírat si z nabídek práce a přihlašovat se v případě zájmu. Základním posláním zaměstnavatele je vytvářet nové nabídky práce a vybírat vhodné kandidáty z přihlášených pracovníků.

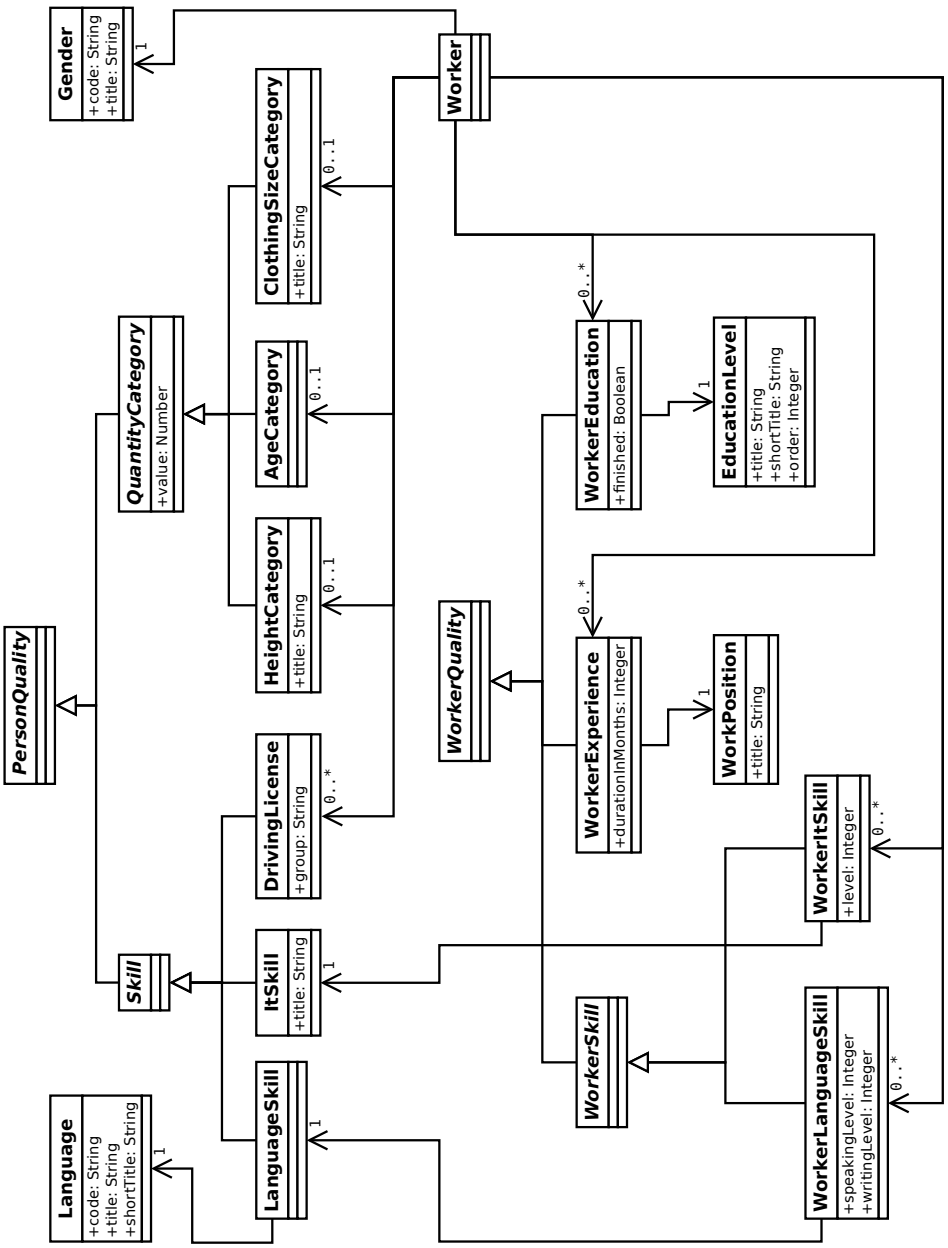
3.3.1.1 Pracovníci

Ke zjednodušení, zkvalitnění a částečné automatizaci procesu výběru a přihlašování na nabídky existuje několik konceptů. Tím prvním je bohatý výběr doplňujících informací, které o sobě může pracovník sdělit, podobně jako ve svém životopise. Vztahy jednotlivých entitních tříd v této oblasti zachycuje obrázek 3.3. Podle kategorií je lze rozdělit na:

- vzdělání (*WorkerEducation*) podle úrovně (*EducationLevel*),
- schopnosti a dovednosti (*Skill* a *WorkerSkill*),
- předchozí pracovní zkušenosti (*WorkerExperience*),
- měřitelné osobní vlastnosti (*QuantityCategory*) jako věk, výška nebo konfekční velikost.

Do informací o vzdělání byly původně zahrnuty i samotné vzdělávací instituce (konkrétní školy a učiliště), nicméně po konzultaci se zadavatelem byl výběr zjednodušen na úroveň vzdělání (např. střední škola, vysoká škola) a informaci, zda pracovník studium na dané úrovni dokončil nebo ne. Třída *EducationLevel* modeluje obecnou úroveň vzdělání, třída *WorkerEducation* slouží jako vazební a označuje vzdělání určitého pracovníka na určité úrovni.

Ze schopností a dovedností jsou modelovány pouze dvě kategorie: jazykové dovednosti (třída *LanguageSkill*) a dovednosti v oblasti informačních technologií (třída *ItSkill*), je však možné kdykoliv přidat další. Obě ze zmíněných kategorií používají vazební třídy (*WorkerEducationSkill*, respektive *WorkerItSkill*), jelikož se uchovává také úroveň, na jaké dovednost ovládají. Jazykové dovednosti rozlišují zvlášť úroveň mluvené a psané formy. Každý pracovník



Obrázek 3.3: Diagram tříd doplňujících informací v profilu pracovníka

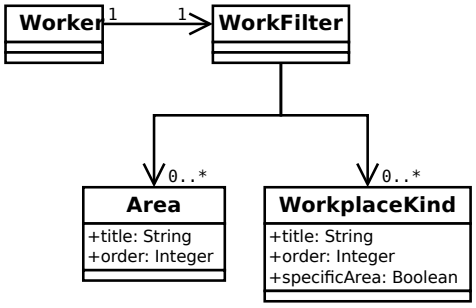
hodnotí sám sebe, což téměř jistě vyústí v ne zcela přesné a objektivní hodnoty napříč všemi pracovníky, nicméně pokud zadavatel práce požaduje určitou úroveň, měl by vždy sám přistoupit k evaluaci každého kandidáta. Z pohledu aplikace není reálné zaručit pravdivost všech hodnot, což platí i o všech ostatních informacích zadaných pracovníkem.

U pracovních zkušeností (vazební třída *WorkerExperience*) se podobně jako u vzdělání nevyplatí požadovat konkrétní instituce (společnosti), jelikož tuto informaci nelze rozumně využít v procesu párování. Proto se výběr omezil na pracovní pozici (třída *WorkPosition*) a délku praxe. I tak by bylo možné vymyslet libovolný počet pracovních pozic, z toho důvodu existuje pouze výběr z několika předdefinovaných, zvolených s ohledem na cílovou skupinu, a možnost zadat vlastní, libovolný řetězec. Při využití této možnosti však nebude zadaná pozice využita k párování.

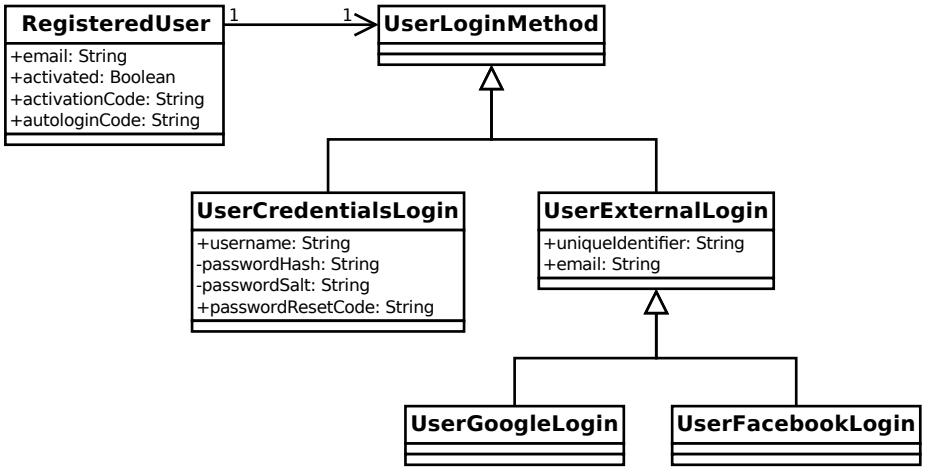
Měřitelné osobní vlastnosti (třída *QuantityCategory*) jsou chápány spíše jako orientační. Pracovník nespecifikuje svůj věk přesně, ale zvolí jednu ze tří skupin: nad 15 let, nad 18 let a nad 21 let (modelované třídou *AgeCategory*). Není totiž cílem znát věk pracovníka, tato informace je důležitá hlavně kvůli legislativním požadavkům na některé pozice, případně druhy smluv. Specifikaci výšky a konfekční velikosti (třídy *HeightCategory* a *ClothingSizeCategory*) využijí zadavatelé na příklad u pozic, kde jsou požadovány uniformy.

Druhým konceptem pro zkvalitnění procesu vyhledávání pracovníků a nabídek práce je filtr (reprezentovaný třídou *WorkFilter*). Pomocí něho si pracovník zvolí druh pracoviště (třída *WorkplaceKind*) a lokalitu (třída *Area*), kde by rád pracoval. Zadavatel při vytváření nabídky volí ze stejných hodnot pro každou nabídku, a pracovníkům se následně zobrazují jen chtěné nabídky. Filtr však na rozdíl od informací o pracovníkovi není závazný a sám pracovník může při manuálním vyhledávání nabídek filtr potlačit nebo dočasně změnit. Vztahy tříd modelující filtr zobrazuje obrázek 3.4. Atribut *specificArea* třídy *WorkplaceKind* značí, zda se má pro daný druh pracoviště použít filtr lokality. Jedním z možných druhů pracovišť je totiž práce z domova, kde na lokalitě nezáleží a při párování se tedy vůbec nebere v potaz.

Předtím, než si pracovník vytvoří svůj profil, vyplní doplňující informace a nastaví filtr, si musí vytvořit uživatelský účet registrací. Kromě klasické registrace vyplněním formuláře pro zjištění povinných údajů jako jména a příjmení, pohlaví, e-mailu a hesla, existuje alternativa v podobě přihlášení přes účet u externích portálů jako Google nebo Facebook. Výhodou je samozřejmě rychlost registrace, automatický přenos osobních údajů a absence dalšího přihlašovacího hesla. Takto registrovaný uživatel by nemusel ani potvrzovat platnost svého e-mailu, jelikož tu ověřil již dříve externí portál. Pořád ale může nastat situace, že uživatel už nemá přístup k e-mailu zadanému na sociální síti Facebook, nicméně dá se předpokládat, že se bude jednat o minimum případů. Stejným způsobem jako při registraci se uživatel bude následně přihlašovat. Jednotlivé způsoby se nedají kombinovat, a proto při registraci externím účtem se uživateli nevytvoří klasické přihlašovací údaje. To může



Obrázek 3.4: Diagram tříd filtru pracovníka



Obrázek 3.5: Diagram tříd přihlašování uživatelů

představovat problém, pokud uživatel přijde o přístup k externímu účtu. Takovéto případy se dají řešit na příklad povolením převodu na běžný účet, kdy nové heslo přijde uživateli e-mailem podobně jako při zapomenutém heslu.

Diagram tříd ukazuje obrázek 3.5. Přihlašovacím jménem u pracovníka je e-mail a duplikace vycházela z požadavku, aby přihlašovací a kontaktní e-mail uživatele mohly být odlišné. Nakonec však bylo od tohoto požadavku upuštěno a implementace může na žádost o přihlašovací e-mail rovnou odpovědět kontaktním e-mailem. E-mail u přihlašování externím účtem poskytuje přímo externí portál a je také duplikovaný, protože se může časem změnit, nicméně kontaktní e-mail v aplikaci by se bez vyžádání uživatele měnit neměl. Jako příjemnou funkci je možné uživateli oznámit změnu e-mailu na externím portálu, aby zvážil i změnu v této aplikaci.

3.3.1.2 Zadavatelé práce

Zadavatel práce se musí stejně jako pracovník nejprve registrovat. Na rozdíl od něj však není možná registrace přes účet u sítí Google nebo Facebook. Zadavatel při registraci zadá své identifikační číslo osoby (IČO) a podle něj se z registru ARES automaticky vyplní název společnosti a její sídlo. Pokud má být kontaktní adresa odlišná od sídla, lze ji ručně doplnit. Zadavatel také volí svůj vztah k nabídkám práce, buď se může jednat přímo o zaměstnavatele, nebo o personální agenturu, která práci pouze zprostředkuje. V současné chvíli mezi nimi není žádný rozdíl, nabídky žádné formy nejsou upřednostňovány ani zvýrazňovány. Zadavatel musí při registraci vyplnit i údaje o kontaktní osobě, která má sloužit jako autoritativní v případě potřeby kontaktu. Pro přihlašování slouží speciální přihlašovací jméno a heslo, nepoužívá se e-mail kontaktní osoby. Je to z důvodu, že kontaktní osoba se může v čase měnit z důvodu personálních změn ve společnosti, přihlašovací jméno nikoliv.

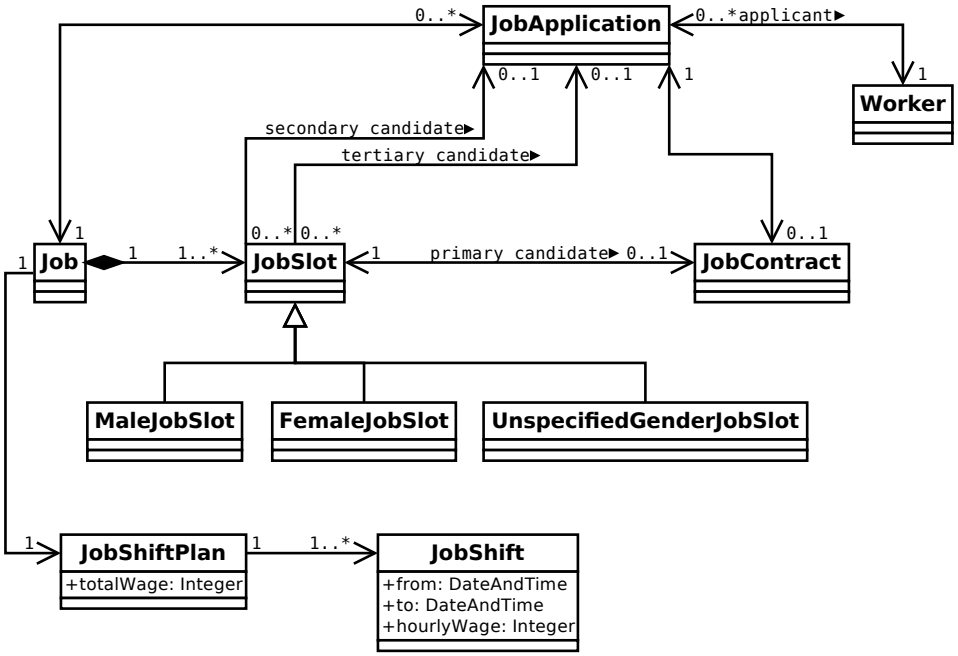
Pro budování značky a nalákání případných zájemců si zadavatel může po registraci vyplnit svůj profil, zadává však pouze logo a krátký text (představení). Vyplnění je jedním z plusových bodů, pomáhá při rozlišování nabídek, a proto nabídky zadavatelů s vyplněným profilem budou částečně zvýhodněny. Po registraci může zadavatel také kdykoliv změnit údaje o kontaktní osobě, přihlašovací heslo nebo kontaktní adresu. Údaje získané z registru, jako název a sídlo společnosti, však měnit nelze, stejně jako identifikační číslo osoby.

3.3.1.3 Nabídky práce

Zadavatel do systému vkládá nové nabídky práce (instance třídy *Job*). Každá nabídka se týká pouze jednoho druhu práce, ale může požadovat obsazení více rovnocenných pozic (instance třídy *JobSlot*). Zadavatel navíc může specifikovat, kolik z těchto pozic je určeno pro muže, kolik pro ženy a u kolika na pohlaví pracovníka nezáleží. Dále zadavatel v kalendáři určí termín brigády (*JobShiftPlan*). Termín nemusí být souvislý a může se jednat o libovolný počet směn (*JobShift*), každá s libovolnou pracovní dobou a hodinovým ohodnocením (mzdou). Mzda nemusí být konstantní, zadavatel by měl mít možnost zadat na příklad vyšší mzdu za víkendovou práci. Celková mzda za všechny směny je spočítána automaticky, ale zadavatel smí finální hodnotu upravit podle svého uvážení. Diagram tříd pozic a směn ukazuje obrázek 3.6.

Na každou pozici může zadavatel předschválit až tři žadatele (jejich žádosti modeluje třída *JobApplication*), prvnímu bude práce závazně nabídnuta (třída *JobContract*) a zbylí dva čekají jako náhradníci pro případ, že první pracovník nabídku odmítne. Náhradníci o zvolení nevědí až do doby, než se dostanou na pozici prvního kandidáta a bude jim učiněna závazná nabídka.

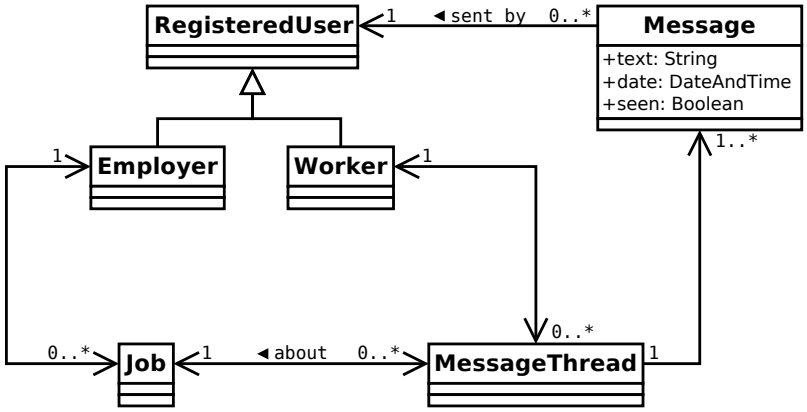
Důležitým bodem je definice pracovního místa a požadavků na pracovníky. Jedná se o protiobraz voleb, které pracovník vyplňuje ve svém profilu a filtru. Nabízelo by se profil oddělit do samostatné entity a použít ho jak u pracov-



Obrázek 3.6: Diagram tříd pozic a směn u nabídek práce

níků, tak u nabídek pro specifikaci požadavků. To nicméně není úplně možné, jelikož mnohé atributy jsou mírně odlišné. Zaprvé, pro zadavatele nabídky jsou všechna pole nepovinná, jejich nevyplnění značí, že na hodnotě nezáleží. U pracovníka jsou naopak některé položky povinné. Zadruhé, pracovník ve svém filtru specifikuje oblasti, ze kterých chce dostávat nabídky. Zadavatel však vyplňuje přesnou adresu pracoviště, případně oblast nebo oblasti, pokud nelze určit konkrétní adresu. Navíc pro některé druhy pracovního místa, jako na příklad práce z domova, je adresa i oblast zcela vynechána. Zatřetí, údaje o postavě jako výšku nebo konfekční velikost zadává pracovník jako jednu hodnotu, zadavatel však volí rozsah hodnot, volitelně otevřený z jedné strany. Začtvrté, u některých atributů může pracovník zadat libovolnou hodnotu, pokud mu žádná z přednastavených nevyhovuje, a tato hodnota se při párování ignoruje. Všechny tyto rozdíly znemožňují modelovat profil pracovníka a požadavky u nabídky jako jednu společnou entitu. Kde to však bylo možné, byly definice v metamodelu (viz kapitola 3.5) znovupoužity.

Kromě toho už nabídka obsahuje jen název pracovní pozice a textové doplnění pracovní náplně, požadavků a benefitů, tedy informací, které se nevejdou do standardizovaných párovacích kategorií. Stejně jako u svého profilu může zadavatel přidat i doplňující fotografie pro zaujetí zájemců.



Obrázek 3.7: Diagram tříd posílání zpráv mezi uživateli

3.3.1.4 Zprávy a notifikace

Způsobem, jakým mohou pracovníci a zadavatelé práce komunikovat, jsou zprávy (třída *Message*). Zadavatelé práce mohou chtít získat více informací, pozvat potenciální kandidáty na pohovor, případně po akceptování pracovníka dohodnout vstupní informace. Zprávy se řadí do vláken (třída *MessageThread*), pracovníkům podle nabídky a zadavatelům podle nabídky a pracovníka. Na novou zprávu bude druhá strana upozorněna pomocí notifikace, případně e-mailu. Zprávy se vždy týkají určité nabídky, s volnými zprávami, na příklad mezi dvěma pracovníky, se v současné chvíli nepočítá. Diagram tříd pro zprávy zobrazuje obrázek 3.7.

Notifikace jsou určeny k upozornění nejen na nové zprávy, ale na všechny důležité události, zejména ty, kde je od uživatele vyžadována nějaká akce. Všechny notifikace mají stanovenu důležitost, podle níž jsou patřičně seřazeny a zvýrazněny. Volitelně může být notifikace časově omezená a po uplynutí stanovené doby zmizí bez ohledu na to, zda ji uživatel viděl a zareagoval na ni nebo ne. Notifikace může také být stanovena jako perzistentní, což znamená, že ji uživatel nemůže odstranit a z výpisu zmizí až na základě jiné akce nebo po vypršení platnosti.

Obecná notifikace (třída *Notification*) zobrazí určený text o určené důležitosti. Text může být volitelně dynamický, což bude využito u časově omezených akcí k zobrazení odpočtu. Může být také zadána akce, která se provede po kliknutí na text notifikace, na příklad pro přesun na relevantní stránku. Některé notifikace však potřebují specifikovat více vlastních akcí ve formě tlačítek pro rychlou reakci, případně s sebou nést další objekty nutné k provedení akcí. Ty jsou potomky třídy *ActiveNotification*.

3.3.2 Úložiště

Spodní vrstvou modelu je úložiště, jehož posláním je zajistit perzistenci dat. Nejčastěji se bude jednat o některou formu databáze, ať už objektovou, která ukládaná data pouze určitým způsobem transformuje nebo serializuje a deserializuje, případně relační, která využívá objektově-relačního mapování. Může se ale jednat i o serializaci objektů do souborů nebo prostou kolekci objektů v paměti, což je v kombinaci s ukládáním celého obrazu virtuálního stroje nejjednodušší způsob uložení pro menší objemy dat. Volba vhodného typu úložiště závisí na požadavcích na objem dat, dostupnost, rychlost přístupu, úroveň konzistence a dalších.

Pro vývoj a testování funkčnosti, kdy je objem dat velmi malý a na dostupnosti a rychlosti příliš nezáleží, lze úspěšně použít prosté kolekce objektů v paměti a ukládání celého běžícího obrazu. Výhodou je jednoduché a velmi rychlé nasazení a žádné závislosti na externích službách. Pro ostrý provoz již však toto řešení z více hledisek nedostačuje: nehodí se pro větší množství dat a není zaručena konzistence ani dostupnost dat, pokud dojde z nějakého důvodu k poškození obrazu. Ukládání obrazu navíc není možné provádět po každé změně z důvodu jeho náročnosti, jelikož na dobu přibližně jedné až dvou sekund pozastaví veškeré běžící procesy.

Při ostrém provozu již bude využita plnohodnotná databáze. Zde se nabízí otázka, jaký typ databáze bude vhodný. Vzhledem ke složitější struktuře dat dává smysl použít objektovou, dokumentově-orientovanou nebo relační. Takových databází je velké množství, problémem je spíše použití z prostředí Pharo, které často nenabízí knihovny pro usnadnění práce. Předběžně byla zvolena dokumentově-orientovaná databáze MongoDB [29], pro kterou existuje jednoduše použitelná mapovací vrstva Voyage. Kvůli možným problémům s výkonem ale rozhodnutí není definitivní a na základě testů výkonnosti a zkušeností z reálného provozu může nakonec dojít k výměně za jinou databázi.

3.3.3 Mappery

Mappery abstrahují zbytek modelu od použitého úložiště tím, že nabízí unifikované rozhraní bez ohledu na rozhraní poskytované samotným úložištěm. Rozhraní zahrnuje základní Create–Read–Update–Delete (CRUD) operace: přidání jednoho či více nových objektů, úprava a smazání objektu, vyhledávání objektů, ale také základní enumerace a agregace. Mapper slouží jen jako adaptér, nezabývá se validací dat ani řešením konkurenčních přístupů. V modelu je vytvořen jeden mapper pro každou kořenovou entitní třídu, tedy takovou, která je uložena nezávisle na ostatních. Ne všechny entitní třídy jsou uloženy samostatně, protože by takový přístup výrazně komplikoval práci s asociacemi.

3.3.4 Služby

Služby implementují veškerou netriviální logiku aplikace. Ostatní vrstvy aplikace komunikují s modelem právě skrz jednotlivé služby. Proto služby svým rozhraním a implementací zaručují, že bez ohledu na požadavky zaslané jinými vrstvami aplikace budou data v úložištích vždy platná a konzistentní. Stejně jako u mapperů, pro každou kořenovou entitní třídu existuje i samostatná služba. Ta ke své činnosti používá jak odpovídající mapper, tak ostatní služby. V základu služba implementuje stejné rozhraní jako mapper a všechny požadavky mu beze změny předává. Takové rozhraní je dostačující pro entitní třídy fungující jako číselníky (oblasti, státy, jazyky, dovednosti, věkové a výškové kategorie a podobné) s výjimkou operací přidávání, kdy je většinou nutné kontrolovat duplicity, a mazání, kdy je potřeba upravit závisející entity vhodným způsobem, na příklad odstraněním vazby nebo přesměrováním na jiný, existující objekt. Služby pro složitější entitní třídy obvykle překrývají základní rozhraní, jelikož obvykle potřebují provést další kontroly, a navíc umožňují provádět pokročilé operace, případně fungují i jako fasáda pro zjednodušení práce.

Každý mapper je zpřístupněn pouze odpovídající službě, ostatní služby k němu nemají přímý přístup a pokud potřebují provést operaci zahrnující více kořenových entitních tříd, musí využít rozhraní ostatních služeb. Takovéto skrývání má několik výhod: každá služba odpovídá za konzistenci a správnost obsahu „svého“ mapperu, veškeré kontroly platnosti záznamů mohou být definované jen na jednom místě a konkurenční přístup lze řešit už na úrovni služby. Poslední jmenovaná vlastnost je zvláště důležitá pro implementaci atomických transakcí (z pohledu aplikace) a kontrolu konzistence.

3.3.5 Správce modelu

Poslední částí modelu je třída *Model*, která se stará o navázání spojení s úložištěm a vytvoření potřebných mapperů a služeb využívajících toto úložiště. Zároveň slouží jako jediný vstupní bod do modelu, ostatní vrstvy aplikace si udržují referenci na instanci této třídy a mohou si od ní vyžádat přístup ke službám, které potřebují. Díky tomu si můžou udržovat pouze jednu referenci místo několika (pro všechny potřebné služby), navíc výměna služby znamená pouze změnu reference uvnitř tohoto správce, není potřeba procházet a aktualizovat ostatní vrstvy. V důsledku je tak dokonce teoreticky možné vyměnit celé úložiště za běhu aplikace.

3.4 Vyhledávání nabídek a přihlašování pracovníků

Službu, která zajišťuje vlastní proces párování nových nabídek brigád a pracovníků, přihlašování pracovníků a jejich schvalování, lze považovat jako jednu ze služeb modelu, ve své podstatě je ale z větší části nezávislá a z důvodu

rozsáhlosti a složitosti je zde popsána samostatně. Navíc nezapadá do dříve stanovené definice služeb modelu, jež jsou navázány na některou z kořenových entitních tříd.

Ve velmi zjednodušené podobě lze základní proces popsat následovně: Zadavatel (společnost) v systému vytvoří novou nabídku práce a určí požadovaný počet pracovníků a požadavky na ně. Na základě těchto požadavků jsou o nově vložené práci informování všichni vyhovující pracovníci. Ti, kteří o nabídku projeví zájem, podají závaznou nebo nezávaznou žádost. Zadavatel práce si ze zájemců vybere vhodné kandidáty a schválí je. V případě, že pracovník zaslal závaznou žádost, je tímto spolupráce definitivně potvrzena. V případě nezávazné žádosti ještě pracovník dostane čas na rozmyšlenou, během kterého musí spolupráci akceptovat a tím definitivně potvrdit. Pokud ji během této doby odmítne nebo vůbec nezareaguje, je jeho místo uvolněno a zadavatel může schválit jiného kandidáta na tuto pozici. Tento podproces se opakuje, dokud nejsou obsazeny všechny pozice. Celý proces včetně všech speciálních případů je modelován v kapitole 3.4.3.

3.4.1 Kritéria pro párování nabídek a pracovníků

Vyhovující pracovníci jsou vybíráni systémem automaticky na základě požadavků zadaných při vkládání práce a vyplněných profilů pracovníků. K párování jsou využity veškeré možné atributy kromě těch, které zadavatel práce označí jako nepodstatné (libovolné).

Jednou z kategorií je nastavení filtru pracovníka, tedy druh pracoviště a oblast. Zadavatel může vyplnit buď přesnou adresu, případně pouze oblast. Vzhledem k tomu, že pracovník vyplňuje pouze oblast, musí dojít ke spárování adresy a celé oblasti. Díky tomu, že aplikace je ze začátku cílena pouze pro Prahu, lze využít převod PSČ na městský obvod, který by až na výjimky měl být přesný. Jinak je možné využít na příklad databázi adresních míst Ministerstva vnitra České republiky [27], případně převod pomocí veřejného rozhraní některé mapové služby.

Další kategorie zahrnuje párování podle termínu práce a kalendáře pracovníka. Aby se pracovník vyhodnotil jako vhodný, nesmí mít v žádném z termínů práce nastaven v kalendáři zaneprázdněný stav, případně už mít akceptovanou jinou práci. Rozjednané práce, tedy takové, na které pracovník zaslal žádost a zatím nebyl schválen, případně byl předběžně schválen a zatím definitivně nepotvrdil spolupráci, nevadí a mohou se překrývat.

Poslední kategorie obsahuje veškeré požadavky na pracovníka, tedy prvky z profilu pracovníka, jako je jeho vzdělání, předchozí pracovní zkušenosti, schopnosti a dovednosti, výška a konfekční velikost, věková kategorie. Pokud zadavatel některé z nich nevyplní, pak na nich nezáleží a vyhoví kterýkoliv pracovník. Pokud ale některé údaje jsou požadovány a pracovník je do svého profilu nevyplnil, dojde k jeho odmítnutí.

3.4.2 Kritéria pro řazení nabídek a pracovníků

I když proces párování pro pracovníka odfiltruje všechny nevyhovující nabídky a zadavateli práce se tak většinou přihlásí pouze vhodní pracovníci, stále může být počet výsledků vyhledávání vysoký. Aby nejvíce relevantní výsledky byly zobrazeny jako první, je nutné výsledky rozumným způsobem řadit.

Při řazení nabídek patří mezi hlavní kritéria jejich termín a stáří. Čím je termín brigády blíže, tím se bude zobrazovat výše, což platí zvláště pro tzv. „last minute“ brigády, jimž do začátku zbývá už jen několik málo dní nebo hodin. Nově přidané nabídky jsou také zvýhodněny, jednak aby se dostaly do povědomí, jednak aby nabídky přidané dlouho před termínem neobsadily první místa na dlouhou dobu. Další, méně vážená kritéria zahrnují reputaci společnosti: počet přidaných nabídek, počet úspěšně přijatých pracovníků, úplnost profilu, hodnocení pracovníky apod. Na pořadí se také podílí negativní vlivy, na příklad počet zrušených nabídek a rozvázaných spoluprací.

Naopak pracovníci jsou seřazeni v první řadě podle jejich aktivity, tedy mimo jiné podle počtu zaslaných žádostí a následně přijatých brigád. Další důležitou složkou řazení je jejich hodnocení společnostmi, ale také rychlost reakce při akceptování nabídky, případně procento následně odmítnutých nabídek. I pracovníkům v menší míře pomáhá dobře vyplněný profil. Nakonec, stejně jako u nabídek, i pracovníkům se můžou některé nešvary promítnout do řazení. Mezi ně patří opakované rušení žádostí nebo dokonce dohodnutých prací, množství stížností od společností a další.

Vzorce jsou modelovány jako strom výrazů. Nemusí počítat pouze hodnotu pro řazení, nicméně v současné chvíli se jedná o jediné využití. Abstraktní třída *FormulaExpression* definuje obecný výraz ve vzorci. Může se jednat o konstantu (instanci třídy *ConstantExpression*), dynamickou hodnotu z předané entity (třída *DynamicValueExpression*), unární a binární funkci (třídy *UnaryExpression*, respektive *BinaryExpression*) nebo agregační funkci (třída *AggregationExpression*). Mezi binární funkce patří klasické sčítání, odčítání, násobení a dělení, případně minimum a maximum ze dvou výrazů, více jich v současnosti není potřeba, nicméně není problém další funkce kdykoliv přidat. Unární funkcí je násobení koeficientem, což by se dalo vyjádřit za pomoci binárního násobení a číselné konstanty, ale jelikož se jedná o velmi běžnou operaci, byla zavedena tato zkratka pro zjednodušení vzorce. Agregačními funkcemi jsou suma a produkt, ale i minimum nebo maximum z více výrazů. Opět, možných agregačních funkcí je mnohem více, navrženy byly pouze ty potřebné. Dynamické hodnoty jsou vlastní vstupní proměnné. Každá podtřída modeluje jedno z kritérií popsaných v předchozích odstavcích. Instance při výpočtu obdrží na vstupu entitu a vrátí číselnou hodnotu odpovídajícího atributu.

Vzorci, respektive kořeni ve stromu výrazů, je předána entita, pro kterou se má spočítat hodnota řazení, tzv. „skóre“, podle něhož jsou entity sestupně seřazeny. Každý uzel ve stromě odpovídá za výpočet hodnoty sebe a svých

podstromů. Celý výpočet může být poměrně náročný, může zahrnovat i několik přístupů k entitě nebo dalším entitám a je nutné hodnotu spočítat pro všechny výsledky v seznamu, kterých můžou být stovky. V tomto případě je tedy více než vhodné denormalizovat model a hodnotu pro řazení ukládat přímo do entity, aby se nemusela počítat pro každý výpis znovu.

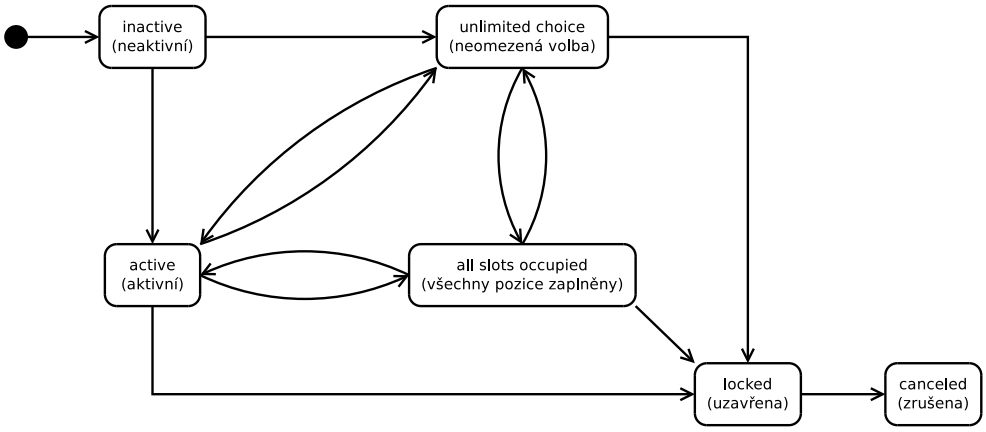
To však s sebou přináší nutnost aktualizace hodnoty, která se obecně může spustit při dvou příležitostech: ihned při změně jakékoliv hodnoty vstupující do výrazu, nebo v pravidelných intervalech bez ohledu na změny. Oba přístupy mají své výhody i nevýhody. Při aktualizaci ihned po změně bude vždy seznam seřazený správně, ale za cenu částečného zpomalení každé operace měnící dotčenou entitu, navíc při změně více hodnot během krátkého času dojde ke zbytečnému opakovanému přepočítávání. Také je tento způsob náchylnější na opomenutí při změně některého parametru a někdy může být obtížné zjistit, které atributy ve skutečnosti vstupují do výpočtu. Aktualizace v pravidelných intervalech se může provádět v odděleném procesu na pozadí, čímž nebude zpomalovat ostatní procesy, ale změny se neprojeví ihned a u některých entit se bude hodnota přepočítávat i přesto, že se nemohla změnit.

Nakonec byl zvolen způsob kombinující oba předchozí. Každá změna označí entitu speciálním příznakem, jenž říká, že některý z atributů entity byl změněn. Oddělený proces pak v pravidelných intervalech vybírá entity s nastaveným příznakem a hodnotu jim přepočítává.

3.4.3 Proces přihlašování a schvalování

Celý proces přihlašování a schvalování pracovníků je poměrně složitý a vyžaduje silnou stavovost zúčastněných objektů. Základní podoba procesu byla popsána v kapitole 3.4, nicméně ve skutečnosti je požadované chování ještě složitější z důvodu mnohých speciálních případů. Navíc není vyloučeno, že v budoucnu se na základě získaných zkušeností a odezvy uživatelů bude proces mírně upravovat.

Patrně nejjednodušším způsobem zavedení stavového chování objektů je zavést do dotčených akcí větvení podle aktuálního stavu, detekovaného buď hodnotami odpovídajících atributů nebo zavedením nového atributu s označením stavu. Toto řešení však trpí závažnými nedostatky. Jedním z nich je porušení takzvaného *principu open-closed*, který tvrdí, že entita by měla být otevřená pro rozšíření funkcionality, ale uzavřená vůči změnám [24]. Případné rozšíření chování by se tedy mělo obejít beze změn stávajícího kódu, což by v tomto případě neplatilo. Dalším z nich je složitost výsledného kódu, obtížná udržovatelnost a náchylnost na chyby. Řešením těchto nedostatků je povýšit stav na takzvaný *first-class objekt*, tedy samostatný objekt s vlastní identitou, a zapojit ho do původního objektu pomocí návrhového vzoru *Stav* (*State*) [13]. Tím pádem bude každý stav modelován samostatně a může být upraven bez zásahu do ostatních objektů včetně toho hlavního. Bez zásahu se obejde i



Obrázek 3.8: Stavový diagram nabídek práce

případné přidání dalšího stavu, rozdělení některého stavu na dva nebo naopak sloučení více stavů do jediného.

Dalším úkolem je identifikovat, které ze zúčastněných entit mají skutečně stavové chování. Zcela jistě se bude jednat o samotnou nabídku práce, která začíná jako neaktivní a po úplném vyplnění všech údajů se stává aktivní (viditelnou pro pracovníky). V této fázi probíhá přihlašování zájemců a jejich výběr na dostupné pozice. Pokud dojde k naplnění všech pozic, je nabídka skryta, nicméně pořád zůstává v pohotovostním stavu, jelikož některý z kandidátů může závaznou nabídku odmítnout nebo spolupráci zrušit. V tom případě je nabídka opět zviditelněna a proces přihlašování a schvalování se znovu spustí. Mimo aktivní stav existuje ještě stav neomezené volby podle specifikace požadavků, kdy nabídka je viditelná pouze těm, které zadavatel vybere. Všem z nich je zároveň práce závazně nabídnuta. Pokud se těmito kandidáty nezaplní všechny pozice, může zadavatel nabídku umístit zpět do aktivního stavu, kdy je nabídka zveřejněna všem pracovníkům. Nakonec se nabízí otázka, v jakém okamžiku nabídku převést z pohotovostního do uzavřeného stavu, kdy je již výběr definitivně ukončen. Uzavření se začátkem první směny není vždy šťastná volba, protože se může stát, že některý z pracovníků na pracoviště nedorazí a bude potřeba narychlo sehnat někoho jiného, v čemž může aplikace také pomoci. Proto se nabídka uzavře až s koncem poslední směny, případně dříve na vyžádání zadavatele. Uzavření je možné provést i před začátkem včetně situace, kdy nejsou všechna místa obsazena. Tím zadavatel dává najevo, že zbývající pracovníci už nejsou potřeba, na příklad proto, že již byl kandidát nalezen způsobem mimo aplikaci. Diagram stavů nabídky ukazuje obrázek 3.8.

Další stavovou entitou je žádost o práci, která může být podána jako nezávazná nebo závazná. Zadavatel nabídky ji může umístit na některou z pozic, případně rovnou odmítnout. Žadatel může závaznou žádost kdykoliv ponížit

na nezávaznou. Když se žadatel dostane na místo prvního kandidáta, je žádost zodpovědná za vytvoření závazné nabídky (kontraktu) spolupráce. Ta představuje další stavovou entitu, protože ji žadatel musí schválit. Pokud byla žádost podána jako závazná, kontrakt je schválen automaticky. Poslední entitou týkající se přihlašování je pracovní pozice, ta však stavovou entitou není, protože její chování se v čase nemění.

Většina změn stavu je iniciována akcí některého z uživatelů (pracovníka nebo zadavatele), případně propagací události do ostatních entit, nicméně některé změny jsou zcela nezávislé na akci a řídí se pouze časem. Jednou z takových je lhůta 24 hodin na definitivní potvrzení nabídky pracovníkem. Pokud se pracovník vyjádří (ať už nabídku akceptuje nebo odmítne), jedná se o změnu stavu iniciovanou akcí. Pokud se ale nestihne vyjádřit, nabídka se zruší sama. K naplánování těchto událostí se využívá speciální služba, popsaná v sekci 3.7.

3.5 Metamodel

Současně s modelem byl tvořen také metamodel. Ten slouží k opatření jednotlivých entit modelu metadaty, která jsou dále využitelná k automatickému generování částí aplikace, ověřování omezujících podmínek nebo dokonce vygenerování samotných entitních tříd. Poslední možnost však nebyla využita z důvodu absence nástrojů, které by tuto funkci realizovaly dostatečně flexibilně. Metadata tak byla přidána jako nadstavba předem vytvořených entitních tříd.

Pro platformu Pharo Smalltalk existuje univerzální metamodelovací framework s názvem Magritte [38]. Zajímavý je především tím, že je zčásti popsán pomocí sebe samotného. Jinak nabízí běžnou sadu definic pro popis standardních datových typů a relací. Existuje dokonce adaptér ke generování relevantních komponent (přehledů a editačních formulářů) pro webový framework Seaside. Magritte je de facto standardem pro popisování entit při vývoji webových aplikací využívajících Seaside, proto bylo rozhodnuto o jeho využití i při vývoji této aplikace. Je však otázkou, zda popularita Magritte není způsobena pouze faktem, že na této platformě neexistuje žádný jiný metamodelovací nástroj, který by byl stejně univerzální jako Magritte. Již na začátku vývoje totiž vyvstalo několik nepříjemných omezení vyplývajících z jeho použití.

První se týkalo komplikovaného znovupoužití definic, a to zvláště omezujících podmínek. Všechny definice totiž Magritte vyžaduje na straně instance, nikoliv třídy, a proto nelze definici jednoduše využít ve více třídách zároveň. Dřívější verze Magritte přitom vyžadovaly definice na straně třídy, čímž bylo znovupoužití jednodušší, i když za cenu zavedení závislosti mezi entitními třídami. Příkladem definice a omezující podmínky může být e-mailová adresa a kontrola její platnosti, kterou využívá více různých entit napříč modelem. Zcela čistým řešením by bylo namodelovat e-mailovou adresu jako samostatnou entitu a původně atomický atribut ve zbytku modelu nahradit vztahem

1:1 k nově vytvořené entitě. To by však komplikovalo kód ve zbytku aplikace, navíc by naráželo na omezení týkající se práce s relacemi, popsané v následujícím odstavci.

Druhým omezením jsou nedostatečné možnosti pro přizpůsobení relací. To se projevuje na příklad při potřebě nastavit omezující podmínky zahrnující atributy z více entit. Někdy je také nutné automaticky filtrovat nevalidní položky ve vztazích typu x:N tak, aby nedošlo k vypsání chybových hlášek uživateli nebo blokaci provedení změn. Při generování formulářů a editaci je totiž vhodné uživateli předvytvořit prázdný objekt, který může rovnou vyplnit. Pokud tak neučiní a vztah je označen jako nepovinný, tento prázdný objekt by měl být automaticky odstraněn, jako kdyby vůbec nebyl vytvořen. Nicméně vzhledem k tomu, že atributy samotného objektu mohou být povinné, kvůli jejich nevyplnění Magritte vypíše chybové hlášky a zablokuje aktualizaci dat, dokud uživatel hodnoty nevyplní nebo vytvořený objekt ručně neodstraní.

Třetím omezením je již zmíněné umístění definic na straně instance entity, což v určitých případech do modelu zanáší nepříjemné závislosti. To platí zejména pro atributy definované jako výběr z hodnot. Pokud je seznam hodnot dynamický, tedy není napevno definovaný v entitní třídě a místo toho se načítá z některé služby modelu, vzniká tím závislost entitní třídy na službách modelu. Entitní třídy by však měly být zcela nezávislé, naproti tomu služby modelu závisí na jejich struktuře, jak ukazuje obrázek 3.2.

Posledním omezením je automatické generování komponent (formulářů) pro Seaside, které postrádá flexibilitu a rozšiřitelnost. Není tak možné generovat formuláře s vlastním, nestandardním rozložením. Adaptér umožňuje generovat pouze uniformní formuláře obsahující dvojice *(popisek, pole pro vstup)* a ani úprava vzhledu pomocí kaskádových stylů (CSS) nepomůže vytvořit požadovanou strukturu. Právě možnost specifikace rozvržení formulářů se ukázala jako klíčová pro vyvíjenou aplikaci, jelikož formulářů je mnoho a jsou rozsáhlejší, je tedy potřeba šetřit místem a dodržet jednotný vzhled podle grafického návrhu. Také práce s relacemi je v existujícím adaptéru neintuitivní a svým rozhraním neodpovídá jednak požadavkům grafického návrhu, jednak obecným pravidlům použitelnosti.

Úprava Magritte pro adresaci zmíněných omezení by zabrala mnoho času, především kvůli výrazným zásahům do jádra knihovny, ale také kvůli chybějící dokumentaci k některým částem. To nakonec vedlo k rozhodnutí opustit Magritte a vyvinout vlastní metamodelovací framework, který bude vycházet z Magritte a bude přizpůsobený požadavkům aplikace, byť mu bude částečně scházet univerzálnost. Návrhem architektury a popisem rozdílů oproti Magritte se zabývá následující kapitola.

3.5.1 Architektura metamodelovacího frameworku

Navržený framework se skládá ze tří hlavních částí: definic jednotlivých typů atributů a omezujících podmínek, tříd pro zaručení konzistence dat (tzv.

„mementa“) a adaptérů pro framework Seaside.

Definice typů atributů přímo vycházejí z Magritte a prošly pouze malými změnami. Na rozdíl od Magritte však navržený framework netrvá na umístění metadat na stranu instance entity, nýbrž přenáší odpovědnost za přístup k popisům na uživatele, případně na entitu samotnou.

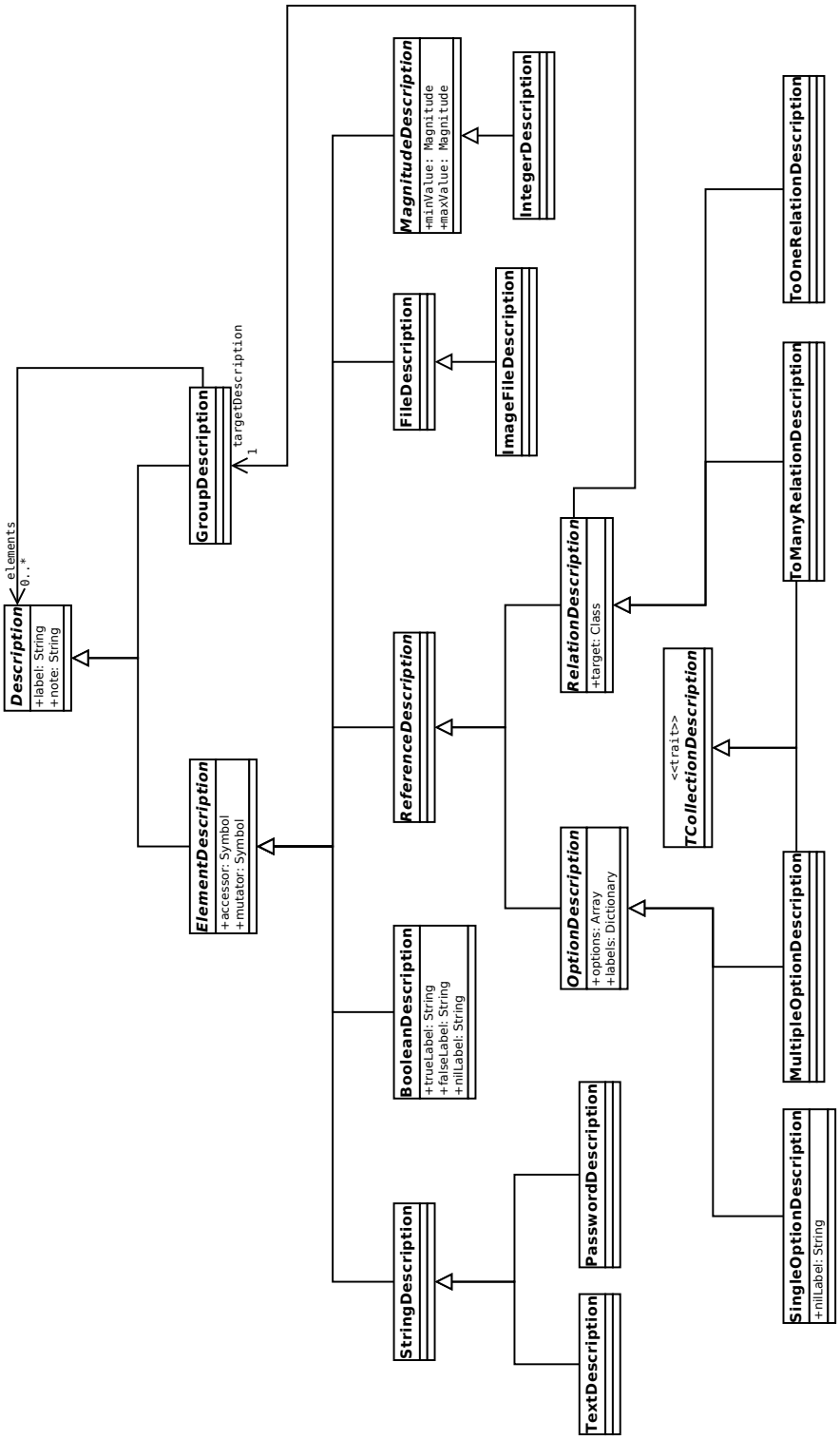
Omezující podmínky jsou dvojího druhu, buď se váží na jednotlivé atributy entity a kontrolují tak jedinou hodnotu nezávisle na ostatních, nebo se váží na celou skupinu atributů a kontrolují více hodnot na sobě závislých. Jako příklad podmínky vázané na skupinu atributů lze uvést kontrolu platnosti časového intervalu: čas začátku musí být před časem konce, na samotných hodnotách však nezáleží.

3.5.2 Definice typů atributů

Opatření atributu modelu metadaty znamená přidání *popisu* (description). Ten může být umístěn kdekoliv, vhodným místem je buď třídní metoda přímo v entitní třídě, nebo třídní metoda v samostatné, nově vytvořené třídě. Při vývoji aplikace byly popisy umístěny do samostatných tříd pro rozbití kruhových závislostí. Metadata v popisu totiž mohou záviset na jiných entitách nebo dokonce na službách modelu, ty však opět závisí na entitách. Závislost na službě modelu je potřeba na příklad u volby více hodnot, kdy hodnoty jsou také reprezentovány vlastními entitami a získávány rovnou z modelové služby.

Celý strom tříd pro definice atributů ukazuje obrázek 3.9. Popis atributu je reprezentován třídou *Description* a v základu definuje pouze název, případně delší poznámku, omezující podmínky atributu a společné rozhraní pro atomické i složené popisy. Atomické popisy (třída *ElementDescription*) se používají pro popis jednotlivých atributů, kdežto složené popisy (třída *GroupDescription*) pouze definují skupinu popisů, ať už atomických, nebo opět složených. Složené atributy jsou využitelné pro definici ucelené skupiny atributů, jejichž elementy se mohou časem měnit. Při přidání jednoho atributu stačí na jednom místě změnit definici skupiny a změna se automaticky promítne do všech míst, které skupinu využívají. Na příklad může automaticky dojít k přidání odpovídajícího pole do všech formulářů. To však bohužel není možné kombinovat s vlastním rozvržením formulářů, jelikož je nutné ho definovat po jednotlivých atributech, a proto se v aplikaci k tomuto účelu nepoužívá. Stále se však složené popisy využívají pro tzv. memento, popsané v kapitole 3.5.4, a pro relace, popsané níže.

Atomické popisy určují obor hodnot atributu a selektory (jména zpráv) pro získání a změnu hodnoty. Selektory slouží k navázání popisu na atribut. Ke každému atributu lze teoreticky definovat více popisů, nicméně nemůžou být použity současně. Možným případem užití více popisů je generování několika různých formulářů, z nichž každý zobrazí pole pod jiným názvem nebo povolí jiné hodnoty atributů. Nic z toho však nebylo ve vyvíjené aplikaci potřeba. Obor hodnot je určen dvěma způsoby. Základním způsobem je typ hodnot



Obrázek 3.9: Diagram tříd definicí atributů v metamodelu

určený přímo jednou z podtříd třídy *ElementDescription*, existují definice pro obvyklé základní datové typy, jako:

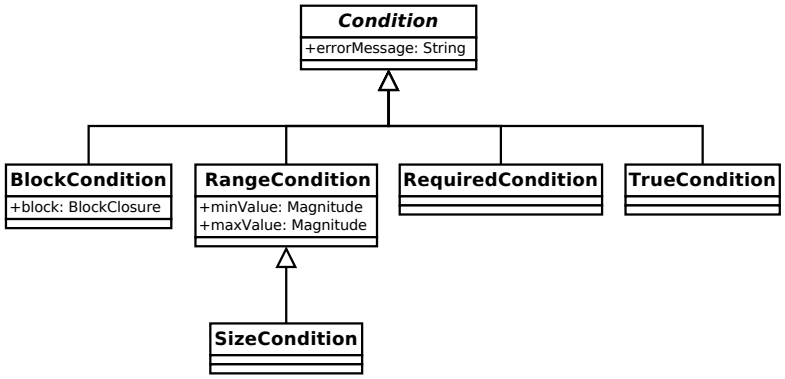
- řetězce (třída *StringDescription* a její specializované podtřídy *TextDescription* pro víceřádkové texty a *PasswordDescription* pro hesla),
- pravdivostní hodnoty (třída *BooleanDescription*),
- typy s úplným uspořádáním (třída *MagnitudeDescription*), na příklad celá čísla (třída *IntegerDescription*),
- ukazatele na soubory (třída *FileDescription* a specializovaná podtřída *ImageDescription*),
- referenční typy (třída *ReferenceDescription*), rozdělené na relace (třída *RelationDescription*) a volby z předem definovaných hodnot (třída *OptionDescription*).

Referenční typy se dále dělí na odkazy na jednu hodnotu (třídy *ToOneRelationDescription* pro vztah x:1 a *SingleOptionDescription* pro výběr maximálně jedné hodnoty z N) a kolekce odkazů (třídy *ToManyRelationDescription* pro vztah x:N a *MultipleOptionDescription* pro výběr M hodnot z N, kde platí $0 \leq \mathrm{M} \leq \mathrm{N}$). Kolekce odkazů můžou mít navíc druhou sadu podmínek, týkající se kolekce jako celku, standardní sada podmínek se vztahuje na každý jednotlivý odkaz zvlášť. Relace si kromě typu (třídy) cílové entity drží také složený popis atributů cílové entity. Není tedy jednoduše možné relaci definovat pro množinu tříd s různými popisy, práce s nimi by byla velmi složitá, zejména ve formulářích. Magritte množinu tříd podporuje, ale jak již bylo zmíněno v úvodu kapitoly 3.5, generované formuláře jsou nepoužitelné. Proto byl návrh omezen na jedinou třídu. Doplňujícím způsobem určení oboru hodnot jsou omezující podmínky, v případě relací i typ cílové entity.

3.5.3 Omezující podmínky

Jednou z výhod opatření atributů metadaty je možnost automatického ověřování hodnot zadaných do automaticky vygenerovaného formuláře. Ověřování hodnot je definováno sadou podmínek, tedy potomků abstraktní třídy *Condition*. Jednou ze základních podmínek je podmínka vyplnění (třída *RequiredCondition*), která pouze ověří, zda hodnotou není speciální hodnota *nil*. Tato podmínka má v celé sadě zvláštní místo, je uložena samostatně a vyhodnocena jako první. Pokud selže, proces kontroly ostatní podmínky přeskočí. Je to z důvodu, aby případná další omezení, které dávají smysl pouze při vyplněné hodnotě, nemusely všechny znovu kontrolovat vyplněnost a duplikovat tak chování pro tento účel určené podmínky.

Další jednoduchou podmínkou je kontrola na hodnotu *true*, jež dává smysl u pravdivostních atributů (popsaných třídou *BooleanDescription*). Typickým



Obrázek 3.10: Diagram tříd podmínek v metamodelu

příkladem použití je pole, které získává souhlas uživatele s všeobecnými podmínkami. U typů s úplným uspořádáním, jako jsou čísla, se často hodí podmínka na rozsah (třída *RangeCondition*), tedy minimální hodnotu, maximální hodnotu nebo obě dohromady. Speciálním případem podmínky na rozsah je kontrola velikosti kolekce, případně jiného objektu schopného určit svou velikost. Je definována třídou *SizeCondition* a rozdílem oproti původní podmínce je jen to, že neporovnává objekt samotný vůči horní a dolní mezi, ale jen jeho velikost (odpověď na zprávu *size*).

Všechny ostatní možné podmínky pojme třída *BlockCondition*, které je jako argument předán libovolný blok (ekvivalent anonymní funkce) přijímající jeden parametr: hodnotu atributu. Blok je při každé kontrole vyhodnocen a musí vrátit pravdivostní hodnotu *true*, pokud hodnota vyhovuje, případně *false*, pokud má být odmítnuta. Všechny výše zmíněné podmínky se samozřejmě dají implementovat i využitím obecné blokové podmínky, ale jsou definovány samostatně kvůli jejich rozšířenosti a kvůli vhodnějším standardním chybovým hláškám.

Podmínky se standardně váží na jeden atomický atribut a kontrolují jeho platnost izolovaně, bez přístupu k hodnotám jiných atributů. Někdy je však potřeba zanést do modelu podmínku kontrolující vztah mezi dvěma hodnotami, tedy hodnotu dvou atributů současně, jako dříve zmíněný příklad s datem začátku před datem konce. Z toho důvodu je možné přiřazovat podmínky i složeným popisům. Pro každou entitu je dostupný minimálně jeden složený popis, a sice ten obsahující všechny definované popisy. V tomto případě dává smysl pouze obecná podmínka (blok), která jako argument dostane všechny hodnoty ve formě slovníku (*IdentityDictionary*), kde klíčem pro každou hodnotu je selektor atributu.

Přehled tříd modelujících omezující podmínky je na obrázku 3.10.

3.5.4 Memento

Pokud je vygenerován formulář pracující s existující entitou, je potřeba při každém odeslání hodnoty uložit. Jednak kvůli perzistenci, jednak kvůli možnosti validace a jednak pro opětovné vyplnění do formuláře. Ukládání přímo do entity by bylo nejjednodušší, nicméně pro zajištění možnosti validace by bylo nutné nejprve hodnoty uložit a až poté validovat. To by však znamenalo, že do entity se mohou dostat neúplné nebo neplatné hodnoty. Proto je využita třída *Memento*[1], respektive její podtřída *ValidatedMemento*, která zaručuje konzistenci a platnost dat v entitách. Memento obsahuje kopii dat z původní entity a všechny operace a formuláře pracují s touto kopií. Teprve ve chvíli, kdy je ověřena validita a konzistence hodnot, jsou data v jediném kroku propsána do entity, včetně všech vazeb a relací. Proto nezáleží na tom, kolikrát byl formulář nesprávně vyplněn a odeslán, entita samotná zůstává beze změny. Vždy se tak entita nebo celý strom entit dostane z platného stavu do jiného platného stavu, bez neplatných mezistavů.

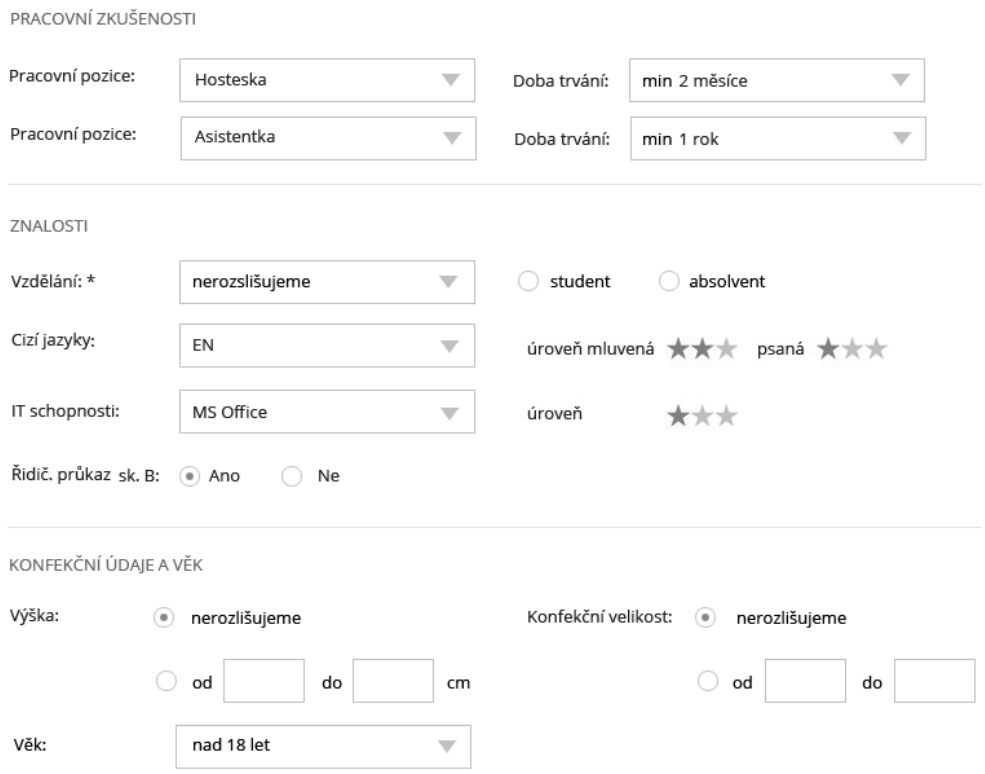
3.5.5 Adaptér pro generování formulářů

Jednou z předností Magritte byla funkce automatického generování formulářů jako komponent frameworku Seaside, bohužel ale s nevyhovujícím rozhraním. Jelikož aplikace bude obsahovat větší množství formulářů, z nichž některé budou poměrně rozsáhlé, bylo nevyhnutelné s touto funkcí počítat.

Generování navíc musí být dostatečně flexibilní, aby bylo možné vytvořit rozložení podobné tomu na obrázku 3.11. Je převzato přímo z grafického návrhu a všechny ostatní formuláře mají podobný styl. Jednotlivé atributy jsou namapovány na blokové a řádkové prvky. Ty jsou v základu rozvrženy do mřížky, byť nepravidelné. Prvek může obsadit celý řádek, více řádků, jednu či více buněk v jednom řádku nebo jen část jedné buňky. Každá buňka může zasahovat do více sloupců, také může obsahovat vnořenou podmřížku. Atributy, které jsou v modelu relacemi, jsou do formuláře integrovány tak, že je nelze rozpoznat od ostatních prvků. Na příklad rozložení z obrázku zahrnuje ve skutečnosti několik entit ve vztahu 1:1 a 1:N. Formulář může obsahovat i další prvky jako dělicí čáry, nadpisy, popisky, odstavce textu nebo jiné, libovolné komponenty.

Diagram tříd formulářových komponent ukazuje obrázek 3.12. Každá komponenta je navázána na memento, do kterého zapisuje hodnoty zadané uživatelem. Formulář po odeslání požádá memento o validaci a akceptaci hodnot. Pokud je vše v pořádku, memento hodnoty zapíše do původní modelové entity. Pokud ne, vrátí formuláři seznam chyb. Ty jsou vypsány uživateli, který může chyby opravit a formulář znovu odeslat. Formulářové komponenty představující vzdálenou entitu v relačním atributu jsou integrovány stejně jako lokální

[1] Termín *memento* byl převzat z frameworku Magritte.



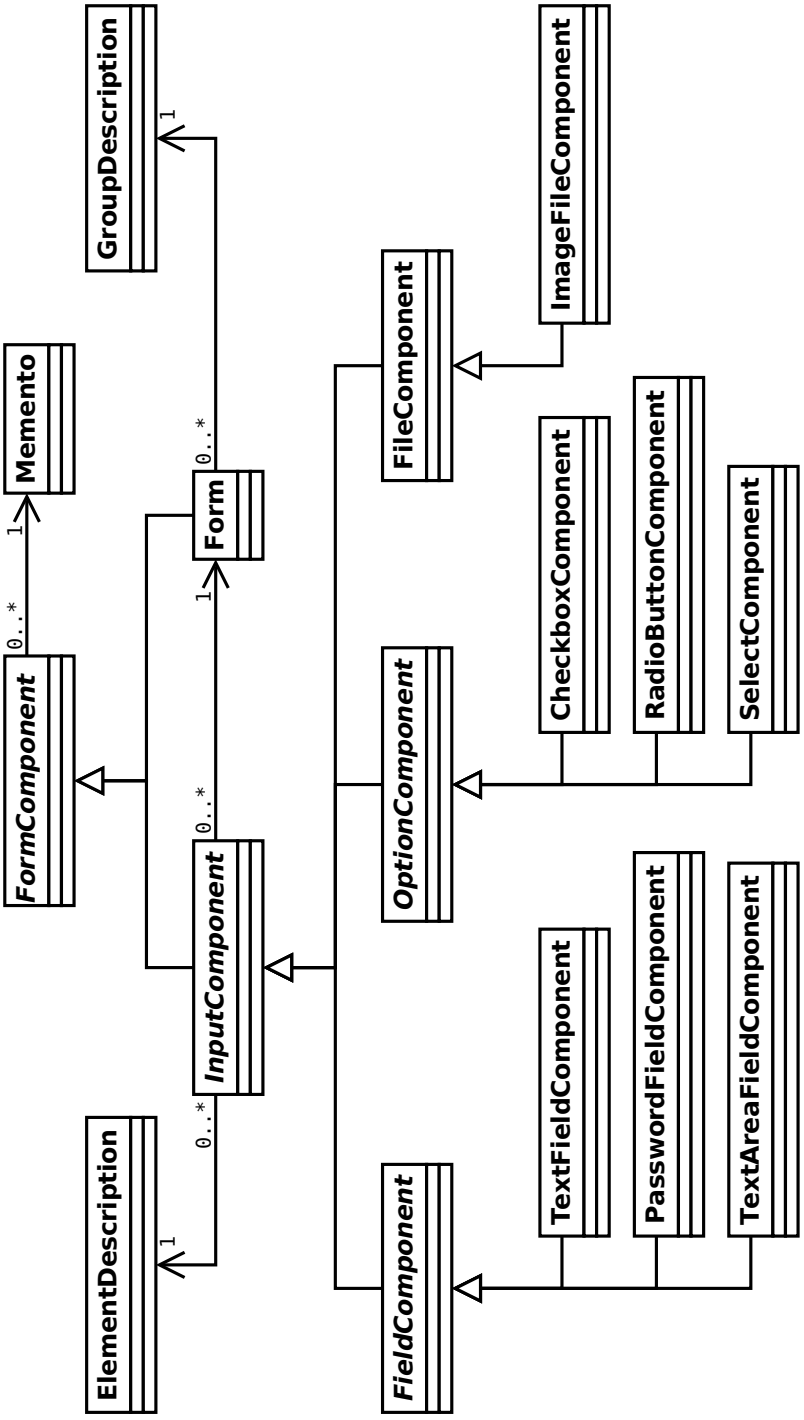
Obrázek 3.11: Ukázka jednoho z formulářů podle grafického návrhu, zdroj: dodáno zadavatelem práce

atributy, jen zapisují hodnoty do vnořeného slovníku v mementu. Memento se postará o správné zapsání hodnot do všech návazných entit.

3.5.5.1 Definice rozložení formulářů

Specifikace komponent v popisech nestačí pro generování libovolně rozložených formulářů, kde více atributů může sdílet jeden řádek nebo dokonce jednu buňku. Navíc může být některý atribut použit ve více formulářích a v každém musí vypadat jinak, je tedy potřeba specifikovat komponenty pro atributy i na úrovni formuláře. Řešením je vytvořit způsob, jak jednoduše specifikovat řádky a buňky mřížky a v nich jednotlivé popisy převedené na komponenty. U každého popisu a každé komponenty ale musí být možnost určit další parametry, jako použití jiné komponenty pro jeden konkrétní formulář nebo počet buněk, který má komponenta zabrat.

Pravděpodobně nejčistší možností je vytvoření soustavy komponent pro tabulku, řádky a buňky (diagram tříd ukazuje obrázek 3.13), které dostanou příkazy jako „vytvoř řádek“, „přidej řádkovou komponentu“, „vytvoř buňku“, „přidej dělicí čáru“ a podobné. Pro lepší flexibilitu a znovupoužitel-



Obrázek 3.12: Diagram tříd formulářových komponent

nost komponent i částí formulářů bude lepší místo příkazu „přidej řádkovou komponentu“, případně analogických pro celou tabulku i buňky, použít jen „přidej komponentu“ a nechat obě komponenty (na příklad tabulku a formulářové pole), ať se dohodnou na požadované úrovni (tabulka, řádek, buňka, část buňky) a případné rozdíly vyřeší automatickým přidáním meziúrovní.

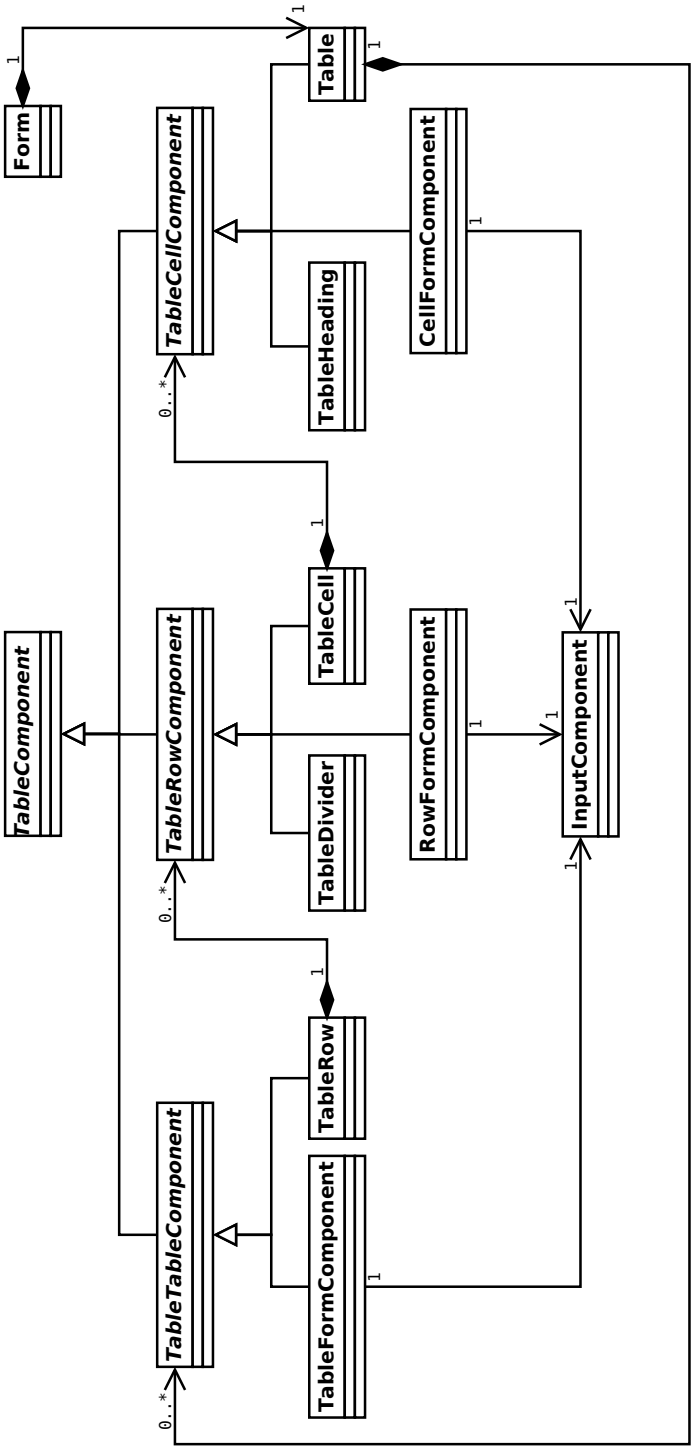
Na příklad, formulářová komponenta vyžadující více řádků může být rovnou vložena do tabulky beze změn. Pokud komponenta vyžaduje pouze jednu celou buňku a je přidána do tabulky, je pro ni automaticky vytvořen samostatný řádek. Obdobně, pokud komponenta nevyžaduje ani samostatnou buňku a je přidána do řádku, je pro ni vytvořena buňka. Pokud by byla stejná komponenta vložena rovnou do tabulky, bude automaticky vytvořen celý řádek, v něm jedna buňka a do ní teprve bude komponenta vložena. Může však nastat i opačný případ, kdy komponenta bude vyžadovat více řádků, ale bude přidána do jednoho řádku. V tom případě je nutné vnořit podmřížku: vytvoří se jedna buňka, v ní nová tabulka a do ní bude víceřádková komponenta vložena.

Takový způsob výrazně podporuje znovupoužitelnost a omezuje duplikaci. Dovoluje vytvořit samostatné části formulářů, umístit je do zvláštních metod a formuláře poté skládat z předem definovaných částí, které se rozložením částečně přizpůsobí místu vložení. Časem může vzniknout požadavek přidat zcela nový atribut do modelu (a jeho popis do metamodelu). Díky tomuto řešení bude stačit přidat pole do relevantní části formuláře, čímž dojde k automatické změně všech dotčených formulářů. Nemůže dojít k opomenutí, aby v některém z desítek formulářů nové pole chybělo.

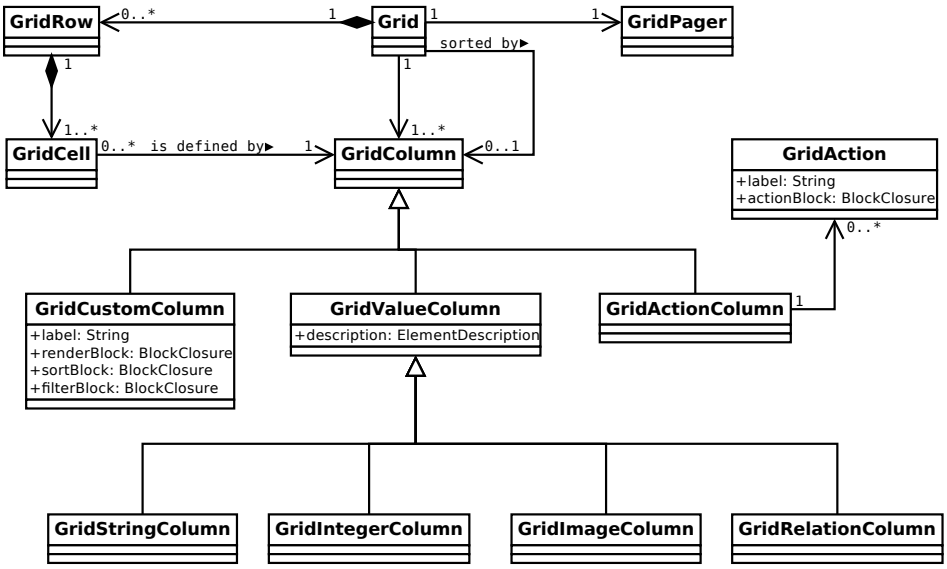
Při množství a rozsáhlosti formulářů však pořád tento způsob znamená relativně hodně práce a psaní. Proto byl navržen ještě méně čistý, ale výrazně jednodušší způsob zápisu rozložení. V podstatě se jedná o deklarativní zápis využívající víceúrovňová (vnořená) pole. První úroveň odpovídá tabulce a jsou na ní uvedeny řádkové a víceřádkové komponenty. Vložené pole odpovídá jednomu řádku a jsou v něm buď buňkové nebo vícebuňkové komponenty, případně další pole odpovídající jedné buňce. U popisů atributů lze přímo v poli určit další atributy, jako počet sloupců, případně předefinovat standardní komponentu pro vykreslení.

3.5.6 Adaptér pro generování přehledů

Přehledy, někdy označované jako *grids*, tvoří další z oblastí, kde se s výhodou využijí informace dostupné z metamodelu. Přehled nabízí rychlé vyhledávání (filtrování záznamů) podle více kritérií, řazení podle libovolné hodnoty, pokud dává smysl, a u každého řádku sadu běžných akcí. Každý přehled je instancí třídy *Grid*. Obsahuje definici zdroje dat, definice sloupců, aktuální filtr a řazení, řádky odpovídající záznamům (entitám) a stránkovač. Diagram tříd zobrazuje obrázek 3.14.



Obrázek 3.13: Diagram tříd rozložení formulářů



Obrázek 3.14: Diagram tříd adaptéru pro generování přehledů

Sloupce (potomci třídy *GridColumn*) určují způsob vykreslení všech buněk v daném sloupci, možnost řazení a možnost filtrování, u filtrování i formulářovou komponentu pro zadání vyhledávané hodnoty. Podobně jako u formulářů existují sloupce pro veškeré možné typy atributů z metamodelu (potomci třídy *GridValueColumn*), vztah však není 1:1, nýbrž M:N, protože jeden typ může být zobrazen více způsoby a více typů může sdílet jeden způsob zobrazení. Každý sloupec je navázán na odpovídající popis atributu v metamodelu (*ElementDescription*). V přehledu nejsou potřeba jen datové sloupce obsahující hodnoty jednoho atributu, ale také obecné sloupce, které se zobrazují vlastním způsobem. Využívají instance třídy *GridCustomColumn*, které předají blok přijímající dva parametry: standardní plátno pro vykreslování z frameworku Seaside a entitu, která je zobrazena v tomto řádku. Pokud sloupec umožní řazení přehledu podle sebe, musí definovat také dvouparametrový řadicí blok, a pokud umožní i filtrování, musí specifikovat ještě komponentu pro změnu filtru a dvouparametrový blok vracející pravdivostní hodnotu, pokud entita vyhovuje zadanému filtru. Běžný je také sloupec s akcemi (*GridActionColumn*), na příklad s tlačítky pro úpravu nebo smazání entity. Akce (*GridAction*) jsou zadány jako blok přijímající entitu jako parametr.

Vykreslení záhlaví tabulky je delegováno na sloupce, každý zobrazí svou buňku hlavičky včetně filtru. Řádek pro každý záznam je instancí třídy *GridRow* a obsahuje zejména jednotlivé buňky (*GridCell*), z nichž každá je navázána na odpovídající definici sloupce a deleguje na ni vykreslení obsahu. Protože záznamů bývá obvykle mnoho, je zobrazeno jen prvních několik desítek a

zbytek je skryt do dalších stránek. Stránkovač (*GridPager*) slouží k zobrazení celkového počtu záznamů a k přepínání na další stránky.

3.6 Prezentační vrstva

Na rozdíl od mnoha jiných frameworků postavených nad vzorem Model–View–Controller (MVC) nebo Model–View–Presenter (MVP), Seaside neodděluje pohledy (*views*) do samostatné vrstvy, ale chápe je jako část vrstvy prezentační. To přináší významné zjednodušení vývoje a možnost tvorby samostatných komponent, které obalují jak strukturu a způsob vykreslení, tak definici reakcí na události. Cenou za tento přístup je vytvoření vrstvy se dvěma odpovědnostmi, což však většinou nepředstavuje žádný problém, jelikož jsou obě části obvykle pevně svázány (není možné vytvořit samotnou vrstvu reakce na události bez znalosti struktury komponenty). Argumenty pro sloučení jsou detailněji rozebrány v kapitole 2.4.

3.6.1 Základní struktura

Framework Seaside klade důraz na modelování prezentační vrstvy jako dynamického stromu komponent, z nichž každá je v ideálním případě nezávislá a znovupoužitelná. Kořenem tohoto stromu je třída *Application*, jež je registrována jako vstupní komponenta celé aplikace. Průchod aplikací vždy začíná vytvořením této komponenty. Poté se může kořenem stát jakákoliv jiná komponenta, v případě této aplikace ale kořenová komponenta zůstává beze změny, jelikož je zodpovědná za nastavení a vykreslení rozvržení stránky (layoutu). Sama však nevypisuje žádný obsah, a proto tento úkol deleguje na svého potomka, kterým se může stát libovolná komponenta.

Třída *Session* uchovává sezení, tedy aktuální stav aplikace. Tato třída je již definována frameworkem Seaside, zde však bude rozšířena o uložení aktuálně přihlášeného uživatele, respektive jeho role. Sezení zahrnuje veškerou historii průchodu aplikací, aby uživatel měl možnost vrátit se v čase použitím historie uchované v jeho webovém klientu (typicky po kliknutí na tlačítka *Zpět* a *Vpřed* v grafickém klientu). Pokud taková situace nastane, aplikace musí být schopna synchronizovat stav v paměti serveru a vykreslený u klienta.

3.6.2 Navigace mezi stránkami

Framework Seaside tím, že pracuje s konceptem komponent a nikoliv stránek, v základu neřeší navigaci mezi stránkami. Místo toho mohou komponenty takzvaně zavolat jiné komponenty, čímž dojde k jejich dočasnému nahrazení jinou komponentou. Běh původní komponenty je dokonce pozastaven přesně v okamžiku, kdy došlo k volání. Nová komponenta může poté v libovolný čas, případně nikdy, ukončit svůj běh a vrátit řízení původní komponentě, volitelně i s předáním návratové hodnoty. Volaná komponenta tím opět odkryje volající

komponentu a běh pokračuje v místě, kde k volání došlo. Tento koncept se obecně označuje jako *continuation* [8].

Celý proces volání, běhu volané komponenty a následné odpovědi obvykle přesahuje rámec jednoho cyklu HTTP požadavku a odpovědi, čímž framework efektivně abstrahuje běh aplikace od nižší, transportní vrstvy. Díky tomu se vývojář nemusí téměř vůbec zabývat zpracováním požadavků a generováním odpovědí, aplikace se chová, jako kdyby neběžela nad bezestavovým zprávově-orientovaným protokolem. Serializaci akcí a stavu do zpráv provádí framework automaticky.

Aplikace využívá koncept komponent a jejich vzájemného volání a rozšiřuje ho o podporu volání celých stránek. Celá stránka však není nic jiného než komponenta obalená základním rozvržením stránky (layoutem). S ohledem na flexibilitu a znovupoužitelnost je možné jako stránku zobrazit libovolnou komponentu. Neexistuje tedy samostatný koncept stránek jako speciálních komponent. Protokol komponent byl rozšířen o volání celých stránek, které kromě samotného zavolání komponenty navíc aktualizuje rozvržení stránky. Každá komponenta si totiž určuje, který druh rozvržení bude použit v případě jejího zobrazení jako celé stránky.

3.6.3 Statické stránky

Návrh aplikace počítá kromě dynamicky generovaných stránek také se statickými stránkami. Za statické stránky se považují sekce tvořené výhradně předem definovaným a neměnných obsahem, zejména textem, společným pro všechny uživatele. Statické stránky obsahuje zejména veřejná část aplikace a jako příklad lze uvést sekci vypisující všeobecné podmínky používání. Neměnnost obsahu je myšlena z pohledu stavu aplikace, nikoliv v čase. Uživatel s dostatečnými právy (administrátor) může obsah pozměnit nebo aktualizovat, stále však půjde o obsah zobrazený všem návštěvníkům stejně, bez ohledu na jejich identitu.

V souladu s filozofií frameworku Seaside jsou i tyto stránky modelovány jako samostatné, znovupoužitelné komponenty. Aby administrátor mohl obsah kdykoliv změnit, nemůže definice komponenty zahrnovat i samotný obsah, ale je nutné obsah spravovat samostatně, na příklad v modelové vrstvě. Dalo by se namítnout, že takto vytvořená stránka již není statická. Pravdou je, že veškeré stránky jsou frameworkem generovány dynamicky, na příklad kvůli vkládání základního rozvržení stránky. Z toho důvodu je nutné oslabit definici statické stránky na stránku se statickým obsahem.

3.6.4 Komponenty

Komponenty jsou, jak již bylo zmíněno, základním stavebním kamenem prezentační vrstvy aplikace. Veškeré uživatelské rozhraní aplikace se skládá z komponent, případně jejich dekorací (popsaných v následující sekci). Komponenty

se mohou výrazně lišit velikostí, stále by však měl platit princip jediné zodpovědnosti (*Single Responsibility Principle*, SRP), kdy každá komponenta by měla mít na starost pouze jednu činnost. Jako u všech ostatních částí je důležitý i princip neopakování se (*Don't Repeat Yourself*, DRY), tedy pokud vznikne potřeba duplikovat kód, pravděpodobně by měl být oddělen do samostatné komponenty.

Mezi nejjednodušší komponenty použité v aplikaci patří jediný odstavec textu, který obvykle není potřeba odlučovat do zvláštní komponenty, s vhodnými dekoracemi se ale může jednat o plnohodnotný a důležitý prvek ve stránce. Podobně jednoduchou komponentou je zobrazení jediného obrázku, použité mimo jiné při otevírání zvětšeného obrázku ve vyskakovacím okně.

Jednou ze složitých komponent je zobrazení profilu uživatele, které má sice více částí a mohlo by být rozděleno do několika samostatných komponent, nedává ale smysl zobrazovat jen některé části, vždy je zobrazen kompletní profil, proto se jedná o celistvou komponentu. Profil vypisuje téměř všechny informace, které o sobě pracovník zadal, kromě těch osobních nebo citlivých (datum narození nebo rodné číslo, adresa, telefon, e-mail).

Další komplexní komponentou je měsíční kalendář. Využívá se na více místech, jak pro pracovníka, tak pro zadavatele práce. Pracovník pomocí něj nastavuje čas zaneprázdněnosti, případně prohlíží konkrétní pracovní dobu u nabídky. Zadavatel jej využívá pro nastavení pracovní doby. Samotný kalendář umožňuje procházet předchozí i následující měsíce, může pro pracovníka zobrazit všechny ostatní nabídky práce včetně jejich stavu a časy zaneprázdněnosti. Jednotlivá místa, kde se kalendář použije, si nastaví, jaké události se mají vypisovat a co se má dít při zvolení některého dne, případně celého rozsahu dní.

Mezi složitější komponenty patří také procesy, formuláře a přehledy, popsané v sekcích 3.6.6 a 3.6.7.

3.6.5 Dekorace

Dekorace jsou dalším z konceptů frameworku Seaside. Podporují znovupoužitelnost komponent tím, že umožňují obalit komponentu a určitým způsobem ji upravit. Obecně se dají rozdělit na tři druhy: behaviorální, které upravují chování, vizuální, které mění nebo doplňují způsob vykreslení, a speciální, které jsou použity v jádře frameworku. Na příklad proces volání komponenty jinou komponentou je realizován pomocí speciální dekorace. V aplikaci se využijí zejména vizuální dekorace.

Některé dekorace jsou velmi jednoduché a slouží k doplnění prvku společného pro více komponent, na příklad nadpisu, upozornění nebo prostého odstavce textu. U formulářů se využívají dekorace pro doplnění tlačítek k odeslání a opatření vstupních polí vysvětlením nebo poznámkou. Pro stránky, které se dále dělí na podsekce, je dekorací možné doplnit menu pro přepínání těchto podsekcí.

Ze složitějších dekorací je možné zmínit modální vyskakovací okno, které zobrazí obsah pouze v malé části obrazovky a všechny ostatní prvky zneaktivní tak, že je překryje tmavou průsvitnou vrstvou. Dekorace jako tyto kromě podpory znovupoužitelnosti také výrazně usnadňují vývoj. Na příklad pro zobrazení důležité zprávy stačí zavolat komponentu zobrazující odstavec textu (vlastní obsah zprávy), přidat dekoraci s tlačítkem pro zavření a přidat dekoraci pro zobrazení ve vyskakovacím okně. Díky vhodným metodám se bude jednat o zhruba tři nebo čtyři jednoduché řádky kódu a výsledkem bude zcela funkční vyskakovací informační okno.

Rozvržení stránky (layout) je také implementováno jako dekorace, v tomto případě kořenové komponenty, která zajišťuje jeho aktualizaci při změně stránky. Rozvržení definuje hlavičku, patičku, hlavní menu, uživatelské menu a další prvky, které se vykreslí okolo obsahu každé stránky. Jednotlivé prvky rozvržení nemusí být statické, na příklad uživatelské menu se mění podle aktuálně přihlášeného uživatele a jeho role. Každá stránka navíc může vyžadovat jiné rozvržení, i když s přihlédnutím ke grafickému návrhu jsou v tomto případě změny spíše kosmetické.

3.6.6 Procesy

Přestože se framework orientuje na komponenty a komunikaci mezi nimi, nabízí také velmi snadný způsob modelování procesů, který do této filozofie dobře zapadá. Procesy jsou zvláštním druhem komponent, které samy nemají žádný obsah, ale můžou volat jiné komponenty. Díky využití *continuations* je zajištěno, že proces volající dvě komponenty za sebou nejprve aktivuje první z nich, pozastaví svůj běh a teprve při návratu řízení a přijetí odpovědi aktivuje druhou komponentu. Vzhledem k tomu, že je skutečně pozastaven běh uprostřed metody, je možné využít běžných možností Smalltalku pro definici složitějších procesů, včetně větvení a cyklů. Vše je přitom zapsáno pomocí standardního kódu v jazyce Smalltalk, není potřeba definovat procesy pomocí speciálního deklarativního jazyka nebo XML dokumentu, jako je tomu v některých jiných systémech.

Jednotlivé procesy jsou implementovány jako potomci třídy *Task*, která do standardního *WATask* z frameworku Seaside doplňuje zkratky pro přístup k aplikaci a modelu a funkce pro navigaci mezi stránkami, stejně jako u ostatních komponent. Za zmínku stojí abstraktní podtřída *SteppedTask*, která modeluje procesy tvořené sekvencí kroků. Umožňuje libovolně se vracet na předchozí kroky a doplňuje i dekoraci (*StepperDecoration*) zobrazující všechny kroky a zvýrazňující aktuální krok. I pomocí ní se uživatel může přepínat mezi jednotlivými kroky, ovšem pouze na takzvaně „odemčené“, což znamená, že nemůže některé kroky přeskočit. Jinými slovy, může se přepínat pouze na kroky, na které alespoň jednou došel standardním způsobem (sekvenčně). Vícekrokové formuláře jsou následně velice snadno implementovány právě pomocí této komponenty.

Poněkud nepříjemným omezením je však nemožnost vnořovat procesy, tedy z jednoho procesu zavolat místo komponenty jiný proces. Toto omezení se ukázalo až při implementaci a na jeho základě musel být návrh upraven. V aplikaci se víceúrovňový proces využívá pro aktivaci profilu uživatele, kdy nejprve v několika krocích vyplní profil, následně v několika krocích nastaví filtr a následně může upravit svůj kalendář. Každý z těchto podprocesů je samostatně krokovaný a hlavní proces může být pozastaven a obnoven na kterémkoliv z podprocesů.

3.6.7 Formuláře a přehledy

Díky existenci metamodelu mohou být veškeré formuláře vycházející z modelových entit generovány automaticky. U formulářů, které nelze jednoduše namapovat na existující entity, existuje možnost vytvořit speciální entitu pouze pro účely formuláře. Navíc vzhledem k návrhu metamodelovacího frameworku je možné znovupoužít popisy atributů z již existujících entit a vyhnout se tak možné duplikaci.

Přehledy (*grids*) jsou rovněž generovány automaticky pomocí metamodelu. Využívají se výhradně v administrační, tedy neveřejné části aplikace. Slouží k procházení, vyhledávání a úpravě uživatelů (pracovníků i zadavatelů) a nabídek, ale také pro úpravu „číselníků“, tedy hodnot výběrových seznamů v profilech pracovníků.

3.7 Podpůrné služby

Kromě modelových služeb a služby pro párování nabídek a poptávek potřebuje aplikace ještě další služby pro svůj chod. Ty se dají označit jako podpůrné, jelikož zajišťují spíše technické požadavky, nicméně i tak mají své místo a je důležité s nimi počítat při návrhu. Zde se jedná o služby zajišťující odesílání e-mailů, správu nahraných fotografií a spouštění časovaných úloh. Služba pro odesílání e-mailů je velice jednoduchá, funguje pouze jako rozhraní pro program, který e-maily skutečně odesílá. Nahrané fotografie je nutno zmenšovat do různých rozměrů (pro zobrazení, náhled nebo profilovou fotografii). S rostoucími rozměry fotografie roste i výpočetní výkon nutný ke kvalitnímu zmenšování. Je nutné rozumně omezit maximální rozměry i datovou velikost, aby nedocházelo k úmyslnému i neúmyslnému zbytečnému přetěžování systému. Samotné zmenšování bude pro menší spotřebu prostředků provádět program běžící mimo aplikaci, spouštěný přímo webovým serverem při prvním požadavku na zatím neexistující velikost. Možné velikosti jsou explicitně definované, povolené jsou jen ty, které se v systému skutečně využívají.

Služba pro spouštění časovaných úloh zajišťuje provádění změn, které nejsou navázány na žádnou uživatelskou akci. Příkladem je stažení potvrzené nabídky spolupráce, pokud ji pracovník do 24 hodin neakceptuje nebo neodmítne, a odeslání nabídky dalšímu předschválenému zájemci. Tato událost

musí nastat vždy, bez ohledu na aktivitu, respektive neaktivitu uživatelů. Rozhraní služby je velice jednoduché, přijímá pouze registrace (vykonání předaného bloku po uplynutí určené doby) a případné deregistrace (zrušení časovače, pokud dříve nastane jiná událost).

3.8 Komunikace mezi klientskou a serverovou částí

Jelikož většinu prvků klientské části generuje sám framework Seaside, stará se také o jejich propojení se serverovou částí. Propojení sestává z více různých činností, mimo jiné z generování správných adres odkazů, zpracovávání odeslaných formulářů nebo reakce na manipulaci s historií a stavem na straně klienta. Veškeré akce jako kliknutí na odkaz nebo odeslání formuláře, které spustí komunikaci mezi klientem a serverem, iniciuje uživatel aplikace prostřednictvím svého webového klienta (prohlížeče). Ten na základě interakce s aplikací odesílá požadavky a přijímá odpovědi za využití standardního protokolu HTTP.

Do zmíněné komunikace není potřeba zasahovat, jelikož framework zaručuje správné odbavení všech požadavků a synchronizaci stavu aplikace na klientské i serverové straně. Výjimkou jsou situace, kdy se cyklus odeslání požadavku, přijetí odpovědi a překreslení celé stránky jeví z pohledu klienta příliš dlouhý, na příklad vzhledem k velikosti provedené akce. V takovém případě přichází v úvahu částečná změna tohoto cyklu technologií AJAX. Ta se využívá v klientském skriptovacím jazyce JavaScript k úpravě granularity cyklu požadavek–odpověď–překreslení. Díky ní není nutné od serveru přijímat a následně překreslovat celou stránku, ale jen její část, případně je možné tento krok zcela vynechat a odesílat požadavky pouze kvůli změně stavu aplikace na serveru, bez čekání na odpověď.

3.9 Shrnutí

Správný návrh je nutnou podmínkou snadné implementace a udržovatelnosti aplikace. Podstatou práce ve Smalltalku jakožto čistě objektovém jazyce je vytvoření čisté a srozumitelné hierarchie tříd, z nichž každá bude mít svůj účel a jasnou odpovědnost. Zároveň by celá aplikace neměla být monolitická, ale rozdělená do více částí nebo vrstev, které na sobě budou co nejméně závislé, v ideálním případě zcela nezávislé. V opačném případě by nebylo možné učinit významnější změnu v jedné části aplikace, aniž by bylo nevyhnutelné přizpůsobit této změně také všechny ostatní části.

V této kapitole byl předložen návrh aplikace, jenž má za cíl splnit zmíněná kritéria. Automatické generování některých aspektů klientské části redukuje množství chyb, které při implementaci mohou nastat. V následující kapitole je popsán převod navržené aplikace do praxe, v kapitole 5 je aplikace podrobena důkladnému testování.

Kapitola 4

Implementace

Při vývoji byl použit obraz platformy Pharo ve verzi 3.0 [37] s předinstalovaným frameworkem Seaside ve verzi 3.1 [6]. U obou se jednalo v době vývoje o nejnovější stabilní verzi. I když čtvrtá verze platformy Pharo byla těsně před dokončením a vydáním, z důvodu některých zpětně nekompatibilních změn pro ni framework Seaside nebyl zatím přizpůsoben.

Díky tomu, že prostředí platformy Pharo Smalltalk a filozofie frameworku Seaside umožňuje implementovat funkčnost velmi rychle a po samostatných částech, probíhala fáze implementace někdy souběžně s fází návrhu. Obvykle po dokončení návrhu jedné části byly myšlenky ihned převedeny do funkční podoby, což pomohlo odstraňovat případné chyby v návrhu velmi rychle. Tento agilní přístup se jednoznačně pozitivně podepsal na rychlosti vývoje a kvalitě návrhu, protože bylo možné vše ověřit a případně upravit bez zbytečných prodlev. Vývoj po částech také podporuje nezávislost jednotlivých vrstev a komponent, jelikož mohla být každá z nich implementována samostatně, tedy předtím, než byly dokončeny ostatní vrstvy a komponenty. Dokonce i testování funkčnosti mohlo probíhat po částech před dokončením vývoje, jelikož živé prostředí a orientace na komponenty umožňuje implementovat jen ty části, které jsou momentálně potřeba, a ostatní prozatím ignorovat.

I tak ale bohužel nebyla implementace v rámci práce zcela dokončena, zejména po stránce prezentační vrstvy, kvůli průtahům při návrhu uživatelského rozhraní i celkové funkčnosti. Návrh popsaný v předchozí kapitole byl dokončen podle posledních známých požadavků a informací, implementace však bude pokračovat i nadále, již mimo rozsah této práce.

Při doplňování interaktivity pomocí skriptů v jazyce JavaScript byla využita knihovna jQuery ve verzi 1.11.0 [16] pro zrychlení vývoje, částečně také díky její integraci s frameworkem. Při implementaci však vyvstaly problémy vycházející z neúplné dokumentace tohoto integračního rozhraní, a to zejména asynchronní komunikace klientské a serverové části technologií AJAX. Právě tato asynchronní komunikace je poměrně důležitá, díky ní byla zkrácena doba odezvy na některých problematických místech.

Během implementace také v nepravidelných intervalech probíhalo testovací nasazení na veřejně dostupný virtuální server, jednak k náhledu a testování pro zadavatele práce, ale také kvůli vyzkoušení procesu nasazení nové verze, případně migrace dat. Různé aspekty nasazení jsou popsány v kapitole 6. V tomto prostředí také probíhalo testování (kromě testů výkonnosti), jehož detaily a výsledky jsou uvedeny v následující kapitole.

Kapitola 5

Testování

Testování navržené a implementované aplikace tvoří nedílnou součást procesu vývoje. Bez extenzivního a důkladného testování není možné zaručit správnost návrhu, funkčnost aplikace ani splnění všech požadavků. Na čistě objektové platformě záleží zejména na čistotě objektového návrhu, proto byla provedena jeho verifikace na základě několika hledisek. Aplikace by měla splňovat požadavky a dělat to, co má, a tak byly zahrnuty jednotkové a funkční testy, ale i uživatelské testování. U veřejně přístupných aplikací velmi záleží i na bezpečnosti, která byla ověřena bezpečnostním testováním. Aby aplikace zvládla nápor uživatelů a zadavatel práce měl představu o možnostech škálování, byl orientačně změřen výkon aplikace.

5.1 Verifikace objektového návrhu

Není snadné testovat čistotu objektového návrhu, neexistují žádná definitivní vodítka k určení jeho správnosti. Jednou cestou k nalezení potenciálních chyb je kontrola základních aspektů a hledání běžně rozšířených prohřešků. Nalezení prohřešku však automaticky neznamená chybu v návrhu, vždy je nutné každé problémové místo individuálně přezkoumat a rozhodnout, zda se skutečně jedná o prohřešek, nebo je použití daného konceptu legitimní či dokonce z nějakého neovlivnitelného důvodu nutné. Jako podklad pro hledání prohřešků velice dobře posloužily materiály a přednášky od profesora Oscara Nierstrasze z Univerzity v Bernu, který při jejich tvorbě vycházel z více zdrojů, na příklad [1], [3], [7], [13] a [43]. Následuje výběr podstatných bodů při kontrole kvality kódu a v důsledku i objektového návrhu.

5.1.1 Jednou a pouze jednou

V originále nazvané *once and only once* [3], toto pravidlo říká, že každá, byť sebemenší myšlenka by se v systému měla objevit jen jednou. Ať už jde o použití hodnot, algoritmů nebo celých metod, nikdy by neměly být duplikovány.

Hledání míst porušujících tento princip není snadné, nicméně při vývoji může pomoci jednoduché pravidlo: pokud nastane potřeba kopírovat nebo opisovat část kódu, jedná se pravděpodobně o porušení principu a daná část by měla být extrahována do samostatného celku (metody, třídy a podobně). Případná oprava nalezených prohřešků je obvykle stejně snadná, stačí duplikovanou část vyčlenit na jediné místo. Při vývoji aplikace proto bylo při každé potřebě kopírovat kód nebo část algoritmu zváženo, zda se skutečně jedná o duplikaci, a případně byl problém rovnou na místě vyřešen.

5.1.2 Velké množství malých částí

Pravidlo pojmenované *lots of little pieces* [3] souvisí s předchozím pravidlem. Dobrý kód bude téměř vždy rozčleněn na spoustu malých tříd a krátkých metod, z nichž každá bude provádět jen jednu činnost a bude zodpovídat jen za ni. Teprve kód takto jemně strukturovaný může docílit splnění pravidla *jednou a pouze jednou*, jelikož bude snadno rozšiřitelný a upravitelný bez potřeby duplikace kódu. Ze statistik zdrojového kódu aplikace bylo zjištěno, že všechny metody mají méně než 20 řádků. Pouze několik metod má mezi 15 a 20 řádky kódu. Přes 95 procent metod má 10 řádků nebo méně, více než 80 procent dokonce 5 řádků nebo méně. Co se týče velikosti tříd, většina (okolo 96 procent) obsahuje 10 metod a méně, pokud nejsou zahrnuty jednořádkové metody (většinou pro přístup k atributům, případně konstanty).

5.1.3 Selektory odhalující záměr

Selektory zpráv, které si objekty posílají mezi sebou, by měly být nazvány podle jejich účelu (*co* provádí), nikoliv implementace (*jak* toho dosáhnou). Klienti by totiž podle principu zapouzdření neměli vědět (a ani chtít vědět), jak je operace uvnitř implementována, právě díky rozhraní jsou od implementace odstíněni. Výhodou je pak možnost změnit způsob implementace bez vědomí klientů a hlavně bez potřeby jejich změny.

Jednoduchým způsobem, jak tento princip splnit, je nejprve psát klientský kód zasílající zprávu a až poté přemýšlet nad vlastní implementací metody. V okamžiku psaní selektoru tak je obvykle jasné jen to, co je potřeba provést, ale nikoliv, co všechno bude splnění dané akce obnášet a jak se konkrétní kroky provedou. Tento postup psaní byl použit i při vývoji aplikace.

5.1.4 Pojmenování proměnných a parametrů

Instanční proměnné by měly být pojmenovány podle účelu, jaký mají. Pokud je jejich hodnotou kolekce, měly by být pojmenovány množným číslem role, jakou její elementy mají. Stejně tak dočasné proměnné v metodách by měly být pojmenovány podle jejich účelu v metodě. Naproti tomu parametry metod by obvykle měly být pojmenovány podle obecného typu, jaký je očekáván.

Přestože Pharo je dynamicky typovaný jazyk, toto pomáhá vývojáři poznat, jak by měl objekt užívat, jaké rozhraní se od parametru očekává. Pokud více parametrů sdílí stejný očekávaný typ, měly by být navíc rozlišeny podle jejich účelu. Parametr není potřeba pojmenovávat podle očekávaného typu, pokud je již z názvu metody nebo kontextu jasný.

V aplikaci byla dle možností tato pravidla dodržena. Jen v privátních metodách, u kterých se nepředpokládá využití externími klienty, byla v určitých případech použita jména parametrů podle jejich účelu, nikoliv očekávaného typu, pokud to zlepšilo čitelnost kódu metody.

5.1.5 Princip open-closed

Princip říká, že objekt by měl být otevřený pro rozšíření funkcionality, ale uzavřený vůči změnám [24]. Jinými slovy, jakákoliv část systému by měla být jednoduše rozšiřitelná, avšak nikoliv změnou existujícího kódu. Správný způsob rozšíření je na příklad vytvoření podtřídy a překrytí potřebných metod, vytvoření nové třídy do existující hierarchie, případně přidání metody. Změna by však neměla vyžadovat zásah do existujících metod a úpravu kódu. Ověření tohoto principu není jednoduché a pozná se nejlépe při reálném požadavku rozšířit funkčnost. Tento princip byl brán v potaz již při návrhu, nicméně v některých případech mohla být zvýhodněna varianta zjednodušující návrh i výsledný kód před rozšiřitelnou variantou, pokud bylo potenciální rozšíření při komunikaci se zadavatelem prohlášeno za velmi nepravděpodobné.

5.1.6 Delegace a zapouzdření

Objekt by neměl provádět činnost, kterou může delegovat na někoho jiného. Zároveň by však neměl zůstat objekt, který sám nemá žádnou odpovědnost a pouze všechnu práci přenáší na ostatní objekty. Zapouzdření značí, že objekt by měl skrývat svou vnitřní implementaci a nedovolit ostatním objektům manipulovat s vnitřním stavem přímo, pouze pomocí veřejného rozhraní, které objekt nabízí. Ve Smalltalkovém systému je každý objekt automaticky zapouzdřen, pouze objekt sám může přistoupit ke svému vnitřnímu stavu. Ostatní objekty musí zaslat zprávu.

5.1.7 Princip minimální znalosti (zákon Deméter)

Tento princip souvisí s principem zapouzdření a říká, že objekt by neměl komunikovat s „cizinci“, pouze s objekty, které mu jsou známé, což zahrnuje:

- sebe samotného,
- argumenty aktuální metody,
- objekty vytvořené v rámci aktuální metody,

- vnitřní komponenty (objekty v instančních proměnných),
- globální proměnné.

Počet přístupů ke globálním proměnným by však měl být omezen na co nejmenší úroveň, aby se snížila závislost na prostředí a ostatních knihovnách nebo komponentách.

Tento princip je v aplikaci porušen u entit, které fungují jako téměř transparentní přepravky dat. Aby mohly služby modelu a prezentační vrstva správně fungovat, potřebují znát strukturu entit a pracovat s jednotlivými hodnotami atributů a relací. Teoretickým řešením je přidat do entity metody delegující akce a přístup k datům na jednotlivé atributy, nicméně výsledkem by byly desítky zcela zbytečných metod, které pouze přidají šum do kódu. V tomto případě je zřejmě výhodnější netrvat na principu minimální znalosti tak striktně.

5.1.8 Typové testy

Dalším běžným prohřeškem je testování typu objektu, ať už speciální zprávou, nebo přímo kontrolou jeho třídy. Používá se pro změnu chování metody na základě typu cílového objektu. Princip se někdy označuje jako „tell, don't ask“: objekt by se neměl ptát jiných objektů na jejich vlastnosti, typ nebo stav. Místo toho by jim měl pouze říct, co mají udělat, a nechat každého z nich, ať akci provede správným způsobem (*single dispatch*). Pokud si ani cílový objekt neví rady nebo akce nepatří pod jeho zodpovědnost, měl by ji delegovat na odpovědný objekt s tím, že mu o sobě sdělí informace, které odpovědný objekt vyžaduje k rozlišení chování (*double dispatch*).

V aplikaci je v určitých případech testování typu využito pro zjednodušení prezentační vrstvy. Využívá se zejména zpráva pro zjištění druhu uživatele (na příklad *isRegistered*, *isWorker* a podobné), jelikož na mnoha místech se každému uživateli zobrazí jiná část rozhraní. Dvojitá delegace by tyto testy mohla řešit, nicméně kód by byl méně přehledný a hůře pochopitelný než prostá podmínka typu „zobraz tento prvek, pokud je uživatel přihlášen“.

5.1.9 Nesprávně umístěné metody

Každá metoda by měla přistupovat k vnitřnímu stavu objektu buď přímo, nebo prostřednictvím ostatních metod. Metoda, jejíž běh závisí na příklad pouze na vlastnostech parametru, je obvykle špatně umístěná a měla by se nacházet přímo ve třídě parametru. Toto pravidlo nelze brát striktně, na příklad metody vracející konstanty nezávisí na ničem a přesto je správné takové metody používat. V aplikaci se nachází metody závisející jen na vlastnostech parametru, které vznikly rozdělením jedné větší metody na několik menších. Nově vzniklé metody můžou jen zjišťovat hodnotu některé vlastnosti parametru a nějakým způsobem ji transformovat, bez ohledu na vnitřní stav samotného objektu. I takové metody jsou ale odůvodnitelné a neměly by být umístěny

jinde. Umístění do třídy parametru by nedávalo smysl, jelikož tento výpočet není jeho zodpovědností.

5.2 Jednotkové testy

Pro ověření správné funkce hlavních služeb systému, tedy vyhledávání nabídek, párování pracovníků a nabídek práce, přihlašování a schvalování pracovníků a dalších, byly vytvořeny jednotkové testy pokrývající velkou část potřebné funkčnosti a možných případů. Tyto testy dokážou zcela automaticky rozpoznat, zda služby fungují tak, jak by měly. Jejich výhodou je také relativně krátká doba běhu, čímž se otevírá možnost spouštět je po každé provedené změně pro kontrolu, zda do systému nebyly zaneseny žádné regresní chyby. Díky tomu je mnohem snazší aplikaci udržovat, vývojář dostane okamžitou zpětnou vazbu, že jeho úprava je pravděpodobně bezpečná a může být nasazena. Nejedná se sice o úplnou jistotu, nicméně většinu závažnějších chyb by testy měly odhalit.

5.3 Funkční testy

Funkční testy probíhaly zejména ve spolupráci se zadavatelem práce, protože požadavky nebyly zcela ukotveny a měnily se v průběhu vývoje na základě výsledků testování. Funkční testy probíhaly nad lokální vývojovou verzí i verzí nasazenou na veřejně přístupném testovacím serveru. Pro testování byla využita testovací data, jež se zčásti zakládala na realitě, zejména v oblasti profilu pracovníků již byly využity seznamy hodnot korespondující s reálnými.

5.4 Uživatelské testy

Testování za pomoci reálných uživatelů vybraných z cílové skupiny přináší výhody v podobě jiného způsobu myšlení a úhlu pohledu, který člověk mimo cílovou skupinu nedokáže věrně napodobit. Výsledná aplikace bude proto těmto testům také podrobena. Testování bude zajišťovat zadavatel práce a bude spočívat ve výběru vzorku uživatelů z cílové skupiny a sledování jejich chování v reprezentativních případech užití. Zjištěné problémy, připomínky a návrhy budou analyzovány a případně vyřešeny a zapracovány do aplikace před jejím otevřením široké veřejnosti.

5.5 Bezpečnostní testy

U aplikací, které používá více uživatelů, a zejména těch částečně nebo úplně veřejně přístupných, je velmi důležité dbát na bezpečnost a ochranu proti

útokům. Zvláště u webových aplikací, na které se útoky stále častěji zaměřují [44], je kritické nepodcenit rizika, která veřejný provoz přináší.

Jedním ze zdrojů informací při kontrole bezpečnosti může být projekt *The Open Web Application Security Project* (OWASP), což je globální nezisková organizace zabývající se mimo jiné sběrem informací a zdrojů týkajících se bezpečnosti webových aplikací [35]. Veškeré materiály, organizované do projektů, jsou publikovány zdarma. Mezi projekty patří také *Top 10* přehledně informující o deseti nejzávažnějších rizikách webových aplikací v současné době [34]. U každého z nich nabízí popis, příklady útoků nebo rady, jak se mu vyhnout. Aktualizovaný přehled vychází každé tři roky, nejnovější byla v čase vývoje verze z roku 2013. Následující odstavce zhodnocují, jaký dopad jednotlivá rizika mají a jak je proti nim aplikace chráněna. Běžně se rizika označují anglickými názvy, pro lepší srozumitelnost však byly u většiny z nich použity české názvy převzaté z českého překladu dokumentu Top 10 [34].

5.5.1 Injektování

Riziko injektování vzniká, když aplikace používá ke komunikaci s jinou částí nebo službou textové rozhraní, které je druhou stranou interpretováno, a když do tohoto rozhraní vkládá uživatelem předkládané řetězce bez patřičného ošetření. Typickými místy, ve kterých vzniká zranitelnost, jsou komunikace s SQL nebo NoSQL databázemi a spouštění externích programů nebo příkazů operačního systému. Komunikace probíhá podle stanovené gramatiky, kde některá slova nebo znaky mají speciální význam. Aplikace v některých případech potřebuje jako některý z argumentů příkazu vložit řetězec zadaný uživatelem. Útočník však zadá speciálně připravený řetězec, který využije některého ze speciálních znaků nebo slov a pozmění význam příkazu nebo spustí další, zcela odlišný příkaz.

Řešením je veškeré uživatelské vstupy důsledně kontrolovat a ošetřovat veškeré výskyty speciálních znaků nebo slov, to se ale neděje vždy. Příčinou může být nedostatečné odlišení bezpečných a potenciálně nebezpečných řetězců, jejich nesprávné ošetření nebo prosté opomenutí. Zranitelnost je častá u aplikací, které vytváří příkazy pro textové rozhraní pomocí prostého spojování řetězců, případně spoléhají na ošetření jinou vrstvou, která tak ale nečiní. Proto je vhodnějším řešením používat pro komunikaci knihovny, které automaticky ošetří vše a vůbec nedovolí vložit neošetřené argumenty. Navíc se veškeré ošetřování soustředí na jedno místo, kde je mnohem snazší ověřit, že je prováděno úplně a správně.

Navržená aplikace je téměř samostatná a používá jen minimum komunikace s externími službami. Jednou z nich je spouštění programu pro odeslání e-mailů, kde jsou argumenty ošetřeny. Další z nich je komunikace s registrem ARES, kde je vstup od uživatele automaticky validován už na straně formuláře, který nepropustí neplatné řetězce. Třetí z nich je komunikace s úložištěm, kde ale není použito interpretované textové rozhraní, překlad do textové po-

doby zajišťuje komunikační vrstva a ta se také stará o automatické ošetření předaných hodnot.

5.5.2 Chybná autentizace a správa relace

Toto riziko zahrnuje více zranitelností. Jednou z nich je nedostatečná ochrana přihlašovacích údajů, na příklad ukládání hesel v čitelné podobě. Pokud se útočník využitím jiné zranitelnosti dostane k databázi, může získat přihlašovací údaje ke všem uživatelským účtům. Navržená aplikace ukládá pouze otisk hesla silným algoritmem, nikoliv heslo samotné. Související zranitelností je odesílání přihlašovacích údajů přes nešifrovaná spojení. Řešením je vynucení šifrovaného spojení, to se ale týká způsobu nasazení a ne aplikace samotné.

Další zranitelností je nesprávné chování funkce pro změnu a obnovu hesla. V případě, že funkce pro změnu hesla nepožaduje současné heslo pro kontrolu, může kdokoliv změnit heslo právě přihlášeného uživatele buď fyzickým přístupem ke stanici v době jeho nepřítomnosti, případně využitím jiných metod k odcizení relace a změně hesla. Také funkce pro obnovu hesla by nikdy neměla rovnou změnit heslo a odeslat ho na kontaktní e-mail uživatele, jednak z důvodu odepření přístupu skutečnému uživateli, když o obnovu požádal někdo jiný, ale také kvůli možnému odposlechnutí e-mailu s heslem nebo kompromitaci e-mailového účtu. Navržená aplikace ke změně e-mailu nebo hesla požaduje zadání hesla současného, a při obnově hesla na kontaktní e-mail zašle pouze instrukce ke změně. Teprve následováním odkazu si uživatel může zvolit nové heslo a v tomto okamžiku také dojde ke skutečné změně, do té doby bude účet přístupný pod původním heslem. Tento přístup je stále náchylný k odposlechnutí e-mailu s odkazem pro obnovení a kompromitaci e-mailového účtu. Jedním z řešení jsou na příklad bezpečnostní otázky, které jsou sice pro uživatele pohodlné (uživatel si snáze vzpomene na odpověď na otázku o své osobě), ale pro bezpečnost kontraproduktivní, protože nahrazují (v ideálním případě) neuhodnutelné heslo uhodnutelnou odpovědí na otázku, jelikož se obvykle jedná o nějaký fakt o uživateli [39]. Vždy tak jde o nalezení kompromisu mezi pohodlím a bezpečím [40]. Pohodlné je na příklad i přihlášení pomocí účtu u portálů Google nebo Facebook, nicméně kompromitace tamějšího účtu vyústí i v možnou kompromitaci účtu v navázané aplikaci.

Běžné jsou zranitelnosti týkající se ochrany samotné relace nebo jejího identifikátoru. Mezi ně patří únik identifikátoru v adrese, kterou uživatel sdílí s jinými uživateli, krádež identifikátoru za pomoci Cross-Site Scriptingu (popsaného v následující sekci), nebo dlouhá či chybějící expirace identifikátorů, kdy uživatel se neodhlásí, útočník získá za nějakou dobu fyzický přístup ke stanici a stále může provádět akce pod účtem předchozího uživatele. Framework Seaside v základu předává identifikátor relace v adresách, nicméně je velmi snadné tuto vlastnost změnit na bezpečnější způsob předávání, pomocí tak zvaných *cookies*. Expirace relace je ve frameworku standardně nastavena na dobu 10 minut, pro potřeby aplikace však byla zvýšena, jelikož pro vyplňování

delších formulářů často 10 minut nestačí a stálé odhlašování je velice nepříjemné. Bezpečnější by však bylo dobu zkrátit a po určité době neaktivity se uživatele zeptat, zda je pořád přítomen. Pokud potvrdí, že ano, pak se na pozadí odešle požadavek automaticky prodlužující platnost relace. Pokud do určené doby nezareaguje, relace vyprší. Framework Seaside také automaticky provádí další kontroly klienta pro případ krádeže identifikátoru relace.

5.5.3 Cross-Site Scripting (XSS)

Cross-Site Scripting, zkracovaný jako XSS kvůli možné zaměnitelnosti s kaskádovými styly (CSS), je patrně nejrozšířenějším rizikem webových aplikací. Zranitelné jsou aplikace, které přijímají a poté ve stránkách používají vstupy od uživatelů a provádí jejich nedostatečnou validaci a ošetření. Útočník poté může na stránku propašovat svůj vlastní škodlivý kód, který se spustí ve webových klientech obětí, tedy ostatních návštěvníků napadené stránky, a může bez jejich vědomí ukrást citlivá data, na příklad identifikátory relací a jiné bezpečnostní klíče. Rozšířenost tohoto druhu zranitelností pramení zejména z velkého množství druhů škodlivého kódu, možných problémových míst i kontextů, z nichž v každém je nutné vstup od uživatele ošetřovat odlišným způsobem.

Existují tři varianty: tak zvané *reflected*, *stored* a *DOM based* [36]. První dvě se týkají zranitelného kódu v serverové části aplikace, třetí se týká klientské části. Varianta *reflected* zneužívá stránky, které ve svém obsahu vypisují některé z parametrů adresy nebo formuláře. Velmi častým případem je na příklad vyhledávací pole, které ve stránce znovu zmíní hledaný výraz. Varianta *stored* zneužívá perzistovaných vstupů od uživatelů, na příklad komentářů, diskusních příspěvků, ale i uživatelských jmen a podobných informací. Tato varianta je o to nebezpečnější, protože obětí se automaticky stává každý návštěvník stránky. Konečně varianta *DOM based* vyžaduje manipulaci se stromem elementů v dokumentu, kdy se nebezpečný vstup dostane bez řádného ošetření do hodnoty některého atributu nebo obsahu elementu.

Framework Seaside nepracuje s HTML šablonami ani nemíchá HTML kód a jiné vstupy, místo toho nabízí rozhraní pro budování struktury stránky, jehož metody mohou každou hodnotu automaticky ošetřit, protože znají přesný kontext. Framework umožňuje vložit neošetřený HTML kód pouze zasláním speciální zprávy *html:*, jejíž výskyty se dají snadno najít a zkontrolovat. Navržená aplikace používá tuto zprávu pouze na jednom místě, a sice výpisu obsahu statické stránky, jenž však nepochází od uživatelů, nýbrž pouze od administrátora. Ten tedy má možnost vložit případný škodlivý kód, proto by bylo vhodnější nahradit ve statických stránkách přímé vkládání HTML za bezpečnější, interpretovaný formátovací jazyk.

5.5.4 Nezabezpečený přímý odkaz na objekt

Riziko se objevuje v aplikacích, které poskytují uživatelům unikátní přímé odkazy na jednotlivé zdroje. V případě, že odkazy jsou rozlišeny jen číselným identifikátorem zdroje, může uživatel adresu snadno změnit a vyzkoušet jiný identifikátor. Pokud aplikace již nekontroluje, zda uživatel má k tomuto zdroji přístup, vzniká zranitelnost. I pokud jsou identifikátory ostatních zdrojů v rozumném čase neuhodnutelné, na příklad obsahují dlouhý náhodný řetězec, může dojít k jejich úniku jak ze strany uživatele, který odkaz zveřejní, tak ze strany aplikace, která odkaz zobrazí i přesto, že by neměla. Framework Seaside standardně negeneruje přímé odkazy na zdroje, tudíž ke změně identifikátoru v adrese nemůže dojít. Může však dojít k zobrazení odkazu vedoucího na nepovolenou akci. V tom případě je nutné u všech akcí důsledně kontrolovat, zda k jejich provedení má uživatel právo nebo ne. Při přechodu na jinou stránku v aplikaci jsou práva pro přístup zkontrolována, odkazy jsou důsledně generovány jen pro uživatele, kteří je vidět mají.

5.5.5 Nezabezpečená konfigurace

Toto riziko zdůrazňuje, že zranitelná nemusí být jen samotná aplikace, ale i všechny další úrovně, jako platforma, webový server, databáze, použitý framework, použité knihovny a další. V každé z nich může existovat zranitelnost, každá z nich může být špatně nakonfigurována. Na příklad rozhraní pro správu některé služby může stále obsahovat přednastavené základní heslo. Je důležité udržovat každou ze součástí systému, tedy správně nakonfigurovat i promptně aktualizovat na novější verze opravující bezpečnostní chyby. Nevyužívané služby a nepotřebné funkce by měly být odstraněny nebo alespoň deaktivovány. Problém se týká oblasti nasazení, které je částečně popsáno v kapitole 6.

5.5.6 Expozice citlivých dat

Problém se týká zejména ukládání citlivých dat v čitelné podobě, bez adekvátního zabezpečení. Pokud aplikace pracuje mimo jiné s hesly, citlivými osobními údaji uživatelů nebo čísly kreditních karet, měly by být ve všech oblastech manipulace, přenosu a ukládání tato data chráněna silným šifrovacím nebo hashovacím (v případě hesel) algoritmem. Navržená aplikace hesla uživatelů řádně hashuje, ostatní citlivá data zahrnují adresu, telefon, datum narození a rodné číslo uživatelů. Ta jsou v současném návrhu ukládána v čitelné podobě, proto by měla být provedena podrobnější analýza rizik a návrh možností, jak tato data s ohledem na způsob využití a platnou legislativu co nejlépe chránit. Ochrana dat při přenosu, případně zálohování, se týká oblasti nasazení.

5.5.7 Chyby v řízení úrovní přístupu

Toto riziko souvisí s rizikem nezabezpečeného přímého odkazu na objekt, popsaného výše. Zranitelná je aplikace, která dostatečně nekontroluje práva uživatele pro přístup k prováděné akci nebo k zobrazenému zdroji. Na příklad pokud akce, která se provede, závisí pouze na vstupu od uživatele, ale již nekontroluje skutečnou roli a práva uživatele. Jiným příkladem je přístup ke skupině zdrojů, z nichž některé jsou přístupné všem registrovaným uživatelům a některé jen administrátorům. Aplikace sice zkontroluje přihlášení uživatele, ale už nezkontroluje práva administrátora u zdrojů, kterých se omezení týká. Jak již bylo zmíněno, jednotlivé stránky v navržené aplikaci i odkazy pro provedení akcí důsledně kontrolují práva uživatele a přímé odkazy na akce neexistují, tudíž nejsou závislé na vstupu od uživatele.

5.5.8 Cross-Site Request Forgery (CSRF)

Riziko podstrčení požadavku z externí stránky spočívá ve vložení obrázku či rámu s podvrženou adresou nebo automaticky odeslaného formuláře na útočníkovu stránku a nalákání uživatele k přístupu na tuto stránku. Vložený obrázek nebo rám se automaticky načte, a pokud je uživatel zároveň přihlášen k zranitelné aplikaci, bez jeho vědomí je vykonána akce zadaná v adrese obrázku nebo datech formuláře, což může být na příklad administrační úkon. Útok je možné zkombinovat s využitím zranitelnosti Cross-Site Scripting a vložit obrázek přímo do zranitelných stránek, čímž odpadá nutnost nalákat uživatele na útočníkovu stránku. V tomto případě je ale útok téměř nerozeznatelný od regulérního požadavku, jelikož webový klient oběti automaticky odešle identifikátor relace a další informace potřebné k autorizaci akce. Prvním nápadem může být provádění citlivých operací pouze metodou *POST* místo *GET*, nicméně taková změna neposkytne žádnou ochranu, jelikož je možné falešný požadavek metodou *POST* provést automaticky odeslaným formulářem [33]. Řešením je každou akci měnící stav či jinak citlivou akci opatřit unikátním, náhodně generovaným identifikátorem (*tokenem*), jenž bude před provedením akce ověřen.

Aplikace vyvinutá ve frameworku Seaside je proti útokům tohoto typu imunní, protože Seaside standardně negeneruje přímé adresy na prováděné akce. Místo toho si pro každou relaci (každého uživatele) buduje strom stavů, ve kterých se aplikace nachází. Každý další stav může vycházet pouze ze stávajících stavů a všechny možné nové stavy jsou definované právě stávajícími stavy, uživatel nemůže vykonat akci, aniž by k ní došel povoleným průchodem. Každý stav je označen náhodným identifikátorem, který v podstatě slouží i jako token k ochraně proti CSRF. Je tedy nemožné cestu k akci i její provedení zfalšovat.

5.5.9 Použití známých zranitelných komponent

Problém zranitelných komponent už byl částečně zmíněn v části o riziku nezabezpečené konfigurace. Aplikace ke své činnosti obvykle potřebuje větší či menší množství knihoven a komponent. Každá z nich může záviset na dalších knihovnách a výsledný počet externích knihoven se může pohybovat v desítkách, o mnohých z nich ani vývojář nemusí mít tušení. Zranitelnost se může objevit v libovolné z nich, je tedy nutné sledovat nově objevené chyby pro všechny použité knihovny a vždy včas aktualizovat na novější verze. To však bude fungovat pouze v případě, že všechny knihovny jsou stále aktivně udržované a nalezené chyby jsou opravovány. U aplikace běžící několik let je pravděpodobnost, že některá z použitých knihoven zanikne nebo přestane být podporována, rozhodně nezanedbatelná.

Možným řešením je používat pouze knihovny a komponenty vlastní výroby nebo takové, které je snadné udržovat v případě, že jejich vývoj skončí. Většinou to ale není reálné, ať už z důvodu časové náročnosti, složitosti nebo omezených zdrojů. Navržená aplikace využívá kromě základního systému platformy Pharo pouze tři externí knihovny, jednou je framework Seaside, na kterém je celá aplikace od základu postavena. Druhou je knihovna usnadňující implementaci přihlášení pomocí účtů u portálů Google a Facebook, která by v případě ukončení podpory mohla být s nevelkou náročností nahrazena jinou, jelikož podobných knihoven existuje více. Třetí je vrstva pro komunikaci s dokumentově-orientovanou databází MongoDB. V případě nutnosti je díky architektuře modelové vrstvy možné bez větších problémů vyměnit úložiště za jiné.

5.5.10 Neošetřené přesměrování a předávání

Riziko vzniká tehdy, když aplikace používá přesměrování na jinou stránku a cíl přesměrování je určen neošetřeným parametrem v adrese. V takovém případě může útočník parametr změnit a docílit přesměrování na úplně jinou adresu. Díky tomu může odkaz využít k takzvanému „rhybaření“ (*phishing*), kdy speciálně upravený odkaz zašle oběti, která na něj klikne a dostane se na falešnou stránku, jejíž cílem je vylákat citlivé údaje. Vzhledem k tomu, že adresa ukazuje na známou doménu, vypadá odkaz důvěryhodně a uživatel se tak může chovat méně ostražitě. Navržená aplikace otevřená přesměrování nepoužívá, a i kdyby je používala, nemohou záviset na neošetřených parametrech v adrese kvůli neexistenci přímých odkazů ve frameworku Seaside.

5.5.11 Další rizika

Výše rozebraná rizika ze seznamu OWASP Top 10 nejsou jediná. Jejich mitigace však zamezí výraznému podílu útoků. Otázka bezpečnosti webových aplikací je mnohem komplexnější a, jak již bylo zmíněno, zahrnuje kromě samotné aplikace i všechny další úrovně a části systému. Zajištění bezpečného

provozu je soustavný proces, neustále se objevují nová rizika, nové zranitelnosti na všech úrovních. Důležité je však podotknout, že bezpečnost z principu nemůže nikdy být stoprocentní. Někdy se objeví dříve neznámá zranitelnost, někdy je potřeba zajistit pohodlí uživatelů za cenu mírně snížené bezpečnosti a někdy může být skutečně důkladné zabezpečení tak nákladné, že se vzhledem k případným obchodním dopadům zneužití ani nevyplatí.

5.6 Testy výkonnosti

Důležitým kritériem při zhodnocení aplikace je také její výkonnost a škálovatelnost, tedy schopnost reagovat na požadavky uživatelů při zachování rozumné doby odezvy a s využitím pokud možno malého množství systémových prostředků. Je nutno také počítat s výraznými výkyvy v zátěži, jelikož požadavky u webových aplikací přichází velmi nerovnoměrně. Prvním důvodem je neuniformní rozložení návštěv přes den, protože naprostá většina uživatelů bude žít ve stejné oblasti a tedy i časovém pásmu. Nejvíce uživatelů proto využije služeb aplikace v ranní nebo večerní špičce, naopak v noci se bude počet návštěv blížit nule. Druhým důvodem jsou možné rozsáhlejší reklamní akce. Během nich se o aplikaci může dozvědět velké množství lidí ve velmi krátkém čase a hromadným klikáním tak zaplaví web mnohonásobně větším množstvím požadavků, než je obvyklé.

Určit očekávanou velikost uživatelské základny a její růst může být velmi obtížný úkol. V případě výkonnosti je nutné brát v úvahu horní hranici odhadů, ideálně ještě s dostatečnou rezervou. Pokud dojde k saturaci systémové kapacity, budou uživatelé čekat na vyřízení požadavků delší dobu nebo systém začne požadavky odmítat, v extrémním případě může dojít k úplnému zahlcení, pádu aplikace nebo systému a totálnímu odmítnutí služby. Taková situace vyústí v nespokojenost a frustraci uživatelů.

Aplikace byla otestována pro orientační představu o výkonnosti. Po konzultaci se zadavatelem práce bude rozhodnuto, zda je naměřený výkon dostačující. Testy však nebyly provedeny na skutečném serveru, proto pro lépe vypovídající výsledky by měla aplikace být před nasazením otestována ve skutečně serverovém prostředí. Měření bylo provedeno na běžném laptopu s procesorem Intel Core i5-4210U o základní frekvenci 1,7 GHz, majícím dvě fyzická a čtyři logická jádra, a 64bitovým operačním systémem Debian GNU/Linux 8.0 s jádrem verze 3.16.0. Zatížení systému ostatními službami bylo nenulové, ale pro potřeby testování zanedbatelné, jelikož při reálném nasazení budou také přítomny podpůrné služby systému a je nutné s nimi počítat.

Tabulka 5.1 ukazuje hodnoty zprůměrované ze čtyř samostatných a izolovaných měření. Seaside dokázal vyřídit více než 140 požadavků za sekundu. Hodnota v reálném provozu bude zcela jistě nižší, jelikož tento umělý test nezahrnoval akce, které provádí intenzivnější výpočty nebo složitější komunikaci s úložištěm. Ty jsou však mnohem méně častější. Parametrem, který

Tabulka 5.1: Testy výkonnosti platformy Seaside

Kritérium	Prům. naměřená hodnota
Rychlost odbavení požadavků (více je lépe)	141,9 / s
Prům. spotřebovaný CPU čas (méně je lépe)	6,5 ms / požadavek
Maximální využití paměti (méně je lépe)	174 MB
Klidové využití paměti (méně je lépe)	65 MB

výsledky nejvíce omezuje, je rychlost procesorové jednotky. Při započítání pouze spotřebovaného CPU času by teoretická rychlost odbavení požadavků vyšla přibližně 154 požadavků za sekundu, což je jen o málo vyšší než skutečně naměřená hodnota (přibližně 142 požadavků za sekundu). Seaside navíc kvůli omezením virtuálního stroje nemůže využít více jader procesoru, celý obraz je interpretován v jediném vlákně. Pro efektivní využití všech jader by bylo nutné spustit více instancí aplikace najednou. Naopak využití paměti je poměrně nízké, i malé virtuální servery typicky nabízejí řádově více paměti pro běžící aplikace. Skutečné výsledky tak budou nejvíce záviset právě na výkonnosti procesorové jednotky serveru, na kterém bude aplikace nasazena.

Kapitola 6

Nasazení do provozu

Oblast nasazení do provozu sice patří mimo rozsah této práce, přesto je vhodné zmínit několik aspektů nasazení aplikací vyvinutých na platformě Pharo či konkrétně ve frameworku Seaside, jelikož podobného materiálu je poměrně nedostatek. Navíc kvůli testování a k náhledu pro zadavatele práce bylo nutné provést byť jen testovací nasazení na veřejně přístupný server, po dokončení implementace a důkladném otestování bude aplikace nasazena do ostrého provozu.

Společností zabývajících se hostováním webových aplikací na platformě Pharo (tedy na úrovni PaaS – *Platform as a Service*) není mnoho, v současné době existují pouze dvě: Seaside-Hosting [31] a Pharocloud [10]. Seaside-Hosting je poskytován zdarma pro propagaci frameworku Seaside a lze na něm hostovat jen nekomerční aplikace [31]. Nabízí pouze 256 megabajtů prostoru pro obraz virtuálního stroje a statické soubory, nenabízí databázi, plný přístup k serveru ani žádné další služby [31]. Navíc neposkytuje žádnou záruku provozu. Není tedy vhodný pro provoz seriózní aplikace. Pharocloud již nabízí placené služby i pro komerční aplikace, nicméně i tak neposkytuje žádné záruky a vyhrazuje si právo poskytovanou službu kdykoliv zrušit [11]. Všechny služby včetně primárního jmenného serveru (DNS) jsou navíc hostovány na jediném serveru, což znemožňuje nasazení aplikace, která vyžaduje vysokou dostupnost. Služba sekundárního jmenného serveru je také v současnosti provozována na připojení domácího nebo kancelářského typu s IP adresou přiřazenou z dynamického fondu, spolu s absencí zabezpečeného spojení (TLS) to otevírá cestu útokům typu *Man-in-the-Middle* (MITM). Ani Pharocloud tudíž nelze doporučit pro běh důležitých nebo komerčních aplikací.

Zbývající možností nasazení je provoz vlastního serveru, ať už dedikovaného nebo virtuálního (na úrovni IaaS – *Infrastructure as a Service*), případně skupiny serverů pro zajištění vysoké dostupnosti nebo rozložení zátěže. Obraz s běžící aplikací by obvykle neměl být dostupný napřímo, ale přes předsazený reverzní proxy server, jako Nginx, Apache nebo lighttpd. Výhodou předsazení je menší spotřeba systémových prostředků pro servírování statického obsahu

a obsluhu šifrovaných spojení, případně zajištění lepší bezpečnosti díky rozsáhlejším možnostem nastavení a optimalizaci chování vůči některým typům útoků. Samotný obraz s běžící aplikací by měl běžet v co nejvíce omezeném prostředí, aby případné průniky nebo chyby nemohly napáchat větší škody. V ideálním případě by měl běžet pod uživatelským účtem s co nejvíce omezenými právy, v případě unixových systémů je možné přidat uzamčení do určitého adresáře pomocí funkce označované jako *chroot*.

Spuštěnou aplikaci je během jejího života nutné aktualizovat, buď kvůli přidání nové funkcionality, změně stávající nebo pro opravu nalezených chyb. Obvykle však aplikace potřebuje běžet trvale a není možné si dovolit ji vypnout, nasadit novou verzi a znovu spustit. Kromě nedostupnosti pro nové návštěvníky by došlo ke ztrátě sezení (a možné ztrátě dat) aktuálních návštěvníků. Jednoduchým, ale ne zcela ideálním řešením je aktualizace v době, kdy je provoz nejmenší nebo dokonce žádný, což obvykle bývá v pozdní noční nebo brzkou ranní hodinu, pokud všichni návštěvníci žijí v podobné časové zóně. Spolehlivějším řešením je spustit novou verzi vedle staré a využít funkce rozložení zátěže (*load balancing*). Zde je však nutné kvůli silné stavovosti aplikace dbát na to, aby požadavky v každém sezení byly směrovány na verzi aplikace, ve které byly vytvořeny. V praxi by to znamenalo všechna nová sezení směrovat na novou verzi aplikace, přičemž stará verze obslouží dříve vytvořená sezení a po nějaké době bude ukončena. Aplikace však musí počítat s více běžícími instancemi, na příklad kvůli synchronizaci nebo atomicitě a izolaci operací s úložištěm. Nejsložitější, ale nejefektivnější je přímá aktualizace běžícího obrazu. U serveru bez grafického prostředí ji lze provést na příklad pomocí speciálního administrátorského rozhraní, nebo využitím vestavěného virtuálního VNC serveru, který přenáší grafické rozhraní vývojového prostředí a umožní tak interaktivní manipulaci s obrazem, což je užitečné pro řešení případných problémů.

Pro zajištění vysoké dostupnosti je vhodné nasadit i automatické monitorování dostupnosti, které je schopno ihned zareagovat na nastalé komplikace. Jelikož platforma Pharo a framework Seaside zatím nejsou vyzkoušeny takovým množstvím webů jako některé jiné technologie, můžou se vyskytnout chyby nebo nestabilita. Z toho důvodu je vhodné nastavit systém, který na příklad při nedostupnosti automaticky obraz restartuje (i za cenu ztracených sezení) a informuje správce. Pokud problém i po restartu přetrvává, může automaticky zkusit starší verzi aplikace nebo obrazu.

Kvůli ochraně citlivých dat, které do aplikace zadávají uživatelé, nelze než doporučit nasazení zabezpečené (šifrované) komunikace mezi klientskou a serverovou částí aplikace. Tím by se měla omezit možnost odposlechnutí dat při komunikaci po nedůvěryhodné síti. Samotné šifrování komunikace však nestačí, je nutné zajistit komplexní zabezpečení na všech vrstvách nasazené aplikace, jak zmiňuje už sekce 5.5.5 v bezpečnostních testech.

Kapitola 7

Porovnání čistě objektového a procesního přístupu

V předchozích kapitolách byl popsán vývoj aplikace pro párování nabídek práce a poptávek, respektive pracovníků, pomocí čistě objektového přístupu na platformě Pharo Smalltalk s frameworkem Seaside. Podstatu správné funkčnosti představuje proces přihlašování a schvalování pracovníků, který byl v kapitole 3.4.3 modelován jako sada stavových entitních tříd, jež svůj stav mění na základě uživatelských akcí, zpráv od jiných entitních tříd nebo nezávislého časovače. Proces tak byl vytvořen od základu specificky pro tuto aplikaci, bez podpory použité platformy nebo jiných knihoven.

Jelikož se jedná o jádro celé aplikace, nabízí se otázka, zda by nebylo výhodnější aplikaci vyvinout na platformě, která již systémově podporuje tvorbu aplikací využívajících explicitně definované procesy k manipulaci s entitami a jejich chováním. Jedna z takových platforem se jmenuje Liferay [17] a byla využita pro vytvoření Portálu spolupráce s průmyslem. Tato kapitola se zaměřuje na porovnání čistě objektového přístupu platformy Pharo Smalltalk a frameworku Seaside právě s procesním přístupem platformy Liferay.

7.1 Popis Portálu spolupráce s průmyslem

Portál spolupráce s průmyslem (dále jen *SSP*) byl vytvořen pro propojení univerzitního světa s průmyslovými partnery a umožňuje studentům získat praxi na reálných projektech. Zároveň za odvedenou práci získají odměnu formou stipendia a výsledek mohou využít i jako semestrální práci do relevantního předmětu.

SSP je v některých ohledech podobný aplikaci vyvinuté v této práci. Také slouží k hledání pracovníků na jednorázové nebo krátkodobé práce, také používá automatické párování nabídek s vhodnými kandidáty podle různých kritérií a také se snaží zjednodušit proces výběru a přihlašování. Oproti ní však

zavádí do procesu ještě další zúčastněnou stranu v podobě školy a jejích učitelů, expertů a dalších pracovníků, kteří se starají o bezproblémový chod, pomáhají oběma zbylým stranám, kontrolují odevzdaná řešení a jejich kvalitu, určují předměty, ve kterých se práce může využít, a podobně.

SSP je poměrně úzce zaměřen, je určen pouze pro studenty Fakulty informačních technologií a zejména na práce typu řešení úlohy nebo vývoje systému, kdežto aplikace vyvíjená v této práci má mnohem širší záběr a zabývá se zejména klasickými brigádami. SSP také s výhodou využívá data z informačních systémů školy, jako informace o absolvovaných předmětech a dosažených výsledcích, pro objektivní hodnocení všech studentů v relevantních oblastech a dovednostech. I tak si každý student může dovednosti včetně jejich úrovně podle svého uvážení upravit, aplikace vyvíjená v této práci však musí spoléhat pouze na data od uživatelů samotných.

Proces přihlašování zájemců je u aplikace vyvíjené v této práci o něco složitější kvůli nabídkám s více pracovními pozicemi, na které zadavatel může předschválit kromě hlavních kandidátů i náhradníky, a kvůli funkcím jako neomezená volba, kdy více předschválených kandidátů zaplňuje volné pozice podle pořadí jejich akceptace. Přiřazování předschválených kandidátů na pozice pak funguje téměř automaticky, bez přičinění zadavatele nabídky. Na druhou stranu aplikace neřeší samotný průběh ani výsledek spolupráce nebo vyplacení odměny, tyto aspekty si zajišťují a řeší obě zúčastněné strany mezi sebou.

7.2 Popis platformy Liferay

Liferay je platforma pro vývoj podnikových portálů v jazyce Java. Zaměřuje se na správu obsahu a podnikových dat, obchodních procesů a tvorbu webových aplikací nad těmito daty. Poskytován je ve dvou edicích: komunitní, dostupné zdarma pod open-source licencí, a enterprise, placené verzi s rozšířenou funkčností a dlouhodobou podporou [20]. Již v základu poskytuje podporu uživatelů a uživatelských rolí, komunikace mezi uživateli, správu obsahu a dat různých druhů, podporu jednoduché tvorby stránek a v neposlední řadě podporu procesů při manipulaci s daty [22] [23].

Srovnání níže vychází kromě vyzkoušení komunitní verze také z rozsáhlé dokumentace, zejména vývojářské [18] a uživatelské [19].

7.3 Komponenty

Framework Seaside staví stránky z komponent a jejich dekorací, uspořádaných do stromové struktury. Každá komponenta se může vykreslit libovolným způsobem, s pomocí kaskádových stylů není problém vytvořit jakýkoliv finální vzhled, odpovídající na příklad grafickému návrhu. Navíc spolu mohou komponenty libovolným způsobem komunikovat a navzájem se ovlivňovat. Seaside

nepoužívá koncept stránek s unikátní adresou, ale nechává tok aplikace na bedrech komponent. Ty se můžou navzájem volat a předávat si řízení, případně spolu s řízením předat návratovou hodnotu. Všechny komponenty jsou definovány v jednom prostředí (obrazu virtuálního stroje) a mají společného předka, který definuje základní funkčnost a chování při volání jiných komponent a návratu řízení, podporu dekorací a všech ostatních funkcí nutných pro správný chod frameworku. Pro celkové pochopení systému a aplikace stačí znát jen základní principy frameworku, kterých není mnoho.

Liferay je postaven na konceptu takzvaných *portletů*, což jsou víceméně samostatné aplikace, silněji oddělené než komponenty v Seaside. Každý z portletů může být implementován samostatně nad libovolnou technologií, s využitím libovolných knihoven nebo frameworků. To přináší velkou výhodu v podobě flexibility, kdy na vývoj každého portletu lze využít přístup, který mu vyhovuje nejvíce. Na druhou stranu v tomto případě stoprocentně platí „méně je více“ a s přílišnou volností přichází řada nástrah, se kterými je nutné počítat. Pokud více portletů využije odlišnou technologii od všech ostatních, výrazně to navýší počet závislostí aplikace, což s sebou přináší jednak nutnost všechny udržovat, jednak bezpečnostní rizika zmíněná v sekci 5.5.9 a jednak rizika, že některá z knihoven bude výrazně upravena nebo bude její vývoj ukončen, zmíněná v téže sekci. Výrazně se ztíží údržba, bude potřeba udržovat velké množství samostatných a zcela rozdílných částí. Srozumitelnost systému jako celku bude velice obtížná, pro pochopení bude nutné seznámit se se všemi použitými technologiemi.

Portlety nejsou zobrazeny samostatně, ale jsou vkládány do stránek, takzvaných *kontejnerů*. Kontejner obvykle pouze určí rozložení, do kterého budou portlety uspořádány, na příklad dva sloupce, tři sloupce s horní lištou a podobné. Jedná se tedy o plochou strukturu, případné vkládání portletů do sebe musí řešit portlet sám. Portlety jsou navíc vkládány jako oddělené bloky, proto vytvoření vlastního, nestandardního rozložení, na příklad kvůli přiblížení se grafickému návrhu, je komplikované. Každá stránka kromě obsažených portletů a jejich rozložení definuje doplňující informace, jako přístupová práva, vzhled v mobilních zařízeních, metadata a další.

Výhodou jsou již v základní distribuci obsažené standardní portlety, na příklad pro výpis různých typů obsahu nebo sociálních funkcí, jako blogů, diskusních fór, anket, wiki systému a podobných. Dalšími výhodami je stavba stránek pomocí drag-and-drop přístupu, kdy administrátor může do zvoleného rozložení přetáhnout požadované portlety, a okamžitý náhled výsledné stránky, včetně simulace zobrazení na mobilních zařízeních.

7.4 Definice procesů

Seaside není procesně-orientovaným frameworkem, přesto nabízí jednoduchý způsob definice komponent implementujících libovolný proces. Ten je vytvořen

pomocí standardní metody v prostředí Pharo, není nutná na příklad deklarativní specifikace. Výsledné komponenty jsou však určeny pouze pro směrování průchodu uživatelským rozhraním v jednom sezení aplikace, nejsou určeny k modelování obchodních procesů ani procesů zahrnujících více zúčastněných stran. I přesto byly využity při vývoji aplikace, jak je popsáno v sekci 3.6.6. Obchodní procesy, jako právě proces přihlašování a schvalování pracovníků, musely být modelovány implicitně využitím stavu u entitních tříd.

Platforma Liferay již v základní verzi podporuje procesy pro jednotlivé entity. Nazývají se *workflow*, dají se přiřadit kterékoliv entitě a platí, že jedna instance procesu se týká jedné entity. Při vytvoření entity se automaticky vytvoří stanovený proces a práce s entitou po celou dobu její existence probíhá v rámci procesu. Hlavním účelem procesů je definovat činnosti, které mají být provedeny, a uživatele, respektive uživatelské role, odpovědné za jejich splnění. Výkonnou složkou jsou tedy sami uživatelé, proces jen slouží k jejich organizaci, dodržení stanoveného postupu a respektování úrovně odpovědnosti.

Fakt, že workflow se primárně týká každé entity samostatně, je poměrně omezující. Není možné jednoduchým způsobem zavést proces zahrnující více entit vytvářených v průběhu procesu, ani více procesů na sobě závislých nebo nějakým způsobem synchronizovaných. V navržené aplikaci závisí některé změny stavu na výsledku jiného procesu nebo na zprávě od jiného objektu, ne však přímo na uživatelské akci, s čímž workflow primárně počítá.

Navíc v základní podobě se koncept workflow v Liferayi příliš nehodí pro procesy různých druhů, jelikož jsou všechny procesy postaveny na stejnou úroveň a zobrazují se uživatelům stejně. Liferay zobrazuje procesy, v nichž aktuálně uživatel nebo jeho role musí provést nějakou činnost, jako úkoly ve speciálních sekcích nebo portletech. V navržené aplikaci musí některé změny stavu projít bez povšimnutí uživatele, některé jsou naopak tak důležité, že se zobrazí tak výrazně, jak jen to je možné. Také v některých stavech není přímo vyžadována akce od žádného uživatele, na příklad pracovníka ani zadavatele, ale jednu z akcí může provést kterýkoliv z nich. To příliš nezapadá do myšlenky workflow, kde stav obvykle znamená nutnost provedení činnosti odpovědným uživatelem.

Co se týče samotné specifikace workflow, Liferay používá deklarativní zápis ve formátu XML. S rostoucí komplexitou a počtem stavů roste i složitost ruční editace procesu. Ke zjednodušení existuje nástroj pro modelování procesů v grafickém prostředí, ten však není volně dostupný, obsahuje ho pouze placená enterprise edice. Uživatelům volně dostupné verze nezbývá než psát celou specifikaci ručně. Pokud je potřeba doplnit vlastní akci provedenou v určitém přechodu nebo stavu, je možné do XML definice vložit část kódu v některém z jazyků JavaScript, Ruby, Groovy nebo Python. Takové míchání deklarativního a procedurálního kódu, navíc ještě v odlišném jazyce (Liferay samotný používá jazyk Java), ale rozhodně nepředstavuje cestu k čisté a udržovatelné aplikaci.

Workflow tedy nelze úplně doporučit pro tvorbu jakýchkoliv procesů. Sil-

nou stránkou je oblast, pro kterou je původně určen: definice činností a odpovědností při vytváření obsahu, nebo obecně dat, v podniku, než dojde k jejich publikaci na podnikovém portálu.

7.5 Rozšiřitelnost

Seaside a celá platforma Pharo je velmi snadno rozšiřitelná, jelikož všechny úrovně jsou transparentní a dostupné ze všech míst. Není problém upravit libovolný aspekt frameworku Seaside v případě, že nevyhovuje. Relativně malý rozsah a jednoduchost kódu tomu napomáhá.

Základní funkcionalita portálu Liferay je také upravitelná, ale používá k tomu pluginy, které se navážou na požadované události nebo akce pomocí takzvaných *hooks* a mohou přidat nebo upravit funkčnost. Pokud hooks nestačí, je úprava jádra sice pořád možná, nicméně velice obtížná. Každý plugin se definuje samostatně do zvláštní třídy a k ní je nutné přidat další dva konfigurační soubory s definicí návaznosti. Funkcionalita je pak kvůli pluginům někdy zbytečně roztroušena na mnoha místech, což snižuje přehlednost systému.

7.6 Model a metamodel

Liferay již v základu přichází s částí funkčnosti a tudíž i modelu, zahrnující uživatele a jejich role, zprávy mezi uživateli a některé typy obsahu. Entitní třídy lze upravovat a přidávat pomocí administrátorského rozhraní. Podobně jako v metamodelovacím frameworku Magritte nebo navržené aplikaci lze definovat metainformace o entitních třídách, typech atributů a podobně. Stejně tak lze rovnou definovat šablony pro zobrazení a editaci. Zároveň však stejně jako Magritte i Liferay trpí malou flexibilitou rozložení formulářů, dostupné je pouze řazení polí, shlukování do skupin nebo přidávání oddělovačů, vytvořit formulář s libovolně složitým rozložením a bezešvou integrací závislých entit není možné. Přidání podpory vlastních rozložení by navíc bylo odhadem daleko složitější než stejná funkčnost implementovaná pro vlastní metamodelovací framework a Seaside.

Předností platformy Liferay oproti navržené aplikaci je snadnější definice dat použitých jako číselníky, což jsou v navržené aplikaci například jazyky, dovednosti, pracovní pozice, druhy pracoviště a jiné entity. Všechny je možné definovat a dynamicky upravovat přímo z administračního rozhraní, případně uživateli s dostatečnými právy.

7.7 Pohodlí vývoje

Seaside je podobně jako Pharo distribuován mimo jiné jako takzvaný one-click obraz o velikosti zhruba 25 megabajtů, který stačí stáhnout, rozbalit a

spustit a vývojář může okamžitě začít pracovat. Současně dojde ke spuštění vývojového webového serveru pro živý náhled aplikace. Liferay je distribuován podobně, i když je proces o trochu složitější a delší. K Liferayi je totiž přibalen externí aplikační server, na příklad Apache Tomcat, který je nutné spustit postupem závislým na platformě. Ten s sebou spustí webové administrační rozhraní, kde probíhá tvorba a konfigurace aplikací. Spolu s Tomcatem má balíček komunitní verze ke stažení velikost necelých 300 megabajtů a spuštění trvá o poznání déle než v případě Seaside (84 sekund oproti dvěma sekundám na běžném laptopu). Seaside nicméně spustí kompletní vývojové prostředí, kde probíhá veškerý vývoj aplikace, tedy všech jejích částí. V případě portálu Liferay se jedná jen o webové administrační rozhraní. Pro vývoj portletů, definici procesů a části konfigurace je nutné pracovat v samostatném vývojovém prostředí (IDE) podle vlastních preferencí, případně využít oficiální Liferay IDE.

Pro Liferay jednoznačně mluví komunita, která je mnohem větší a aktivnější než u Seaside nebo platformy Pharo obecně. Existuje více blogů, uživatelských skupin, aktivnější fórum, uživatelé pořádají setkání nebo vytvářejí videonávody [21]. V případě problému je tak větší pravděpodobnost, že se najde člověk ochotný poradit.

V čem však Liferay neposkytne velkou pomoc, je srozumitelnost, nebo spíše nesrozumitelnost kódu. Na příklad 40 procent zdrojových souborů předdefinovaných stránek má více než 100 řádků a míchá kód v jazyce Java s XHTML kódem, šablonovacím a prezentačním systémem založeným na XHTML syntaxi a dokonce i JavaScriptem.

7.8 Systémové nároky

Nároky na systém jsou pro Liferay vyšší než pro Seaside, nicméně je nutno dodat, že Liferay je již v základu poněkud rozsáhlejší a obsahuje některé aspekty, které v Seaside chybí nebo které navržená aplikace nevyužívá. Tabulka 7.1 orientačně srovnává výkon obou platforem, všechna data jsou průměrem ze čtyř měření. Parametry systému použitého pro provedení měření jsou popsány v kapitole 5.6. Zatížení systému ostatními službami bylo v rámci možností drženo na stabilní úrovni. Lépe vypovídající je však spotřebovaný procesorový čas, který by měl být více nezávislý na zatížení systému než reálný čas. I když Seaside dokázal každou sekundu vyřídit téměř čtyřikrát více požadavků než Liferay, spotřeba procesorového času byla na každý požadavek šestnáctkrát menší. Hlavním důvodem je, že Seaside kvůli omezení virtuálního stroje běží pouze v jednom vlákně, kdežto Liferay jako vícevláknová aplikace může využít všechen dostupný výkon procesoru. I proto je potřeba výsledky brát pouze orientačně, pro skutečně seriózní srovnání by byl potřeba vlastní dedikovaný server, nicméně to není cílem této práce. Za povšimnutí stojí také mnohem výraznější klidová paměťová náročnost platformy Liferay, která limituje nasazení

Tabulka 7.1: Srovnání výkonu platforem Seaside a Liferay

Kritérium	Seaside	Liferay
Rychlost odbavení požadavků (více je lépe)	141,9 / s	38,0 / s
Prům. spotřebovaný CPU čas (méně je lépe)	6,5 ms/pož.	107,5 ms/pož.
Maximální využití paměti (méně je lépe)	174 MB	1507 MB
Reálný čas spouštění (méně je lépe)	cca 2 s	cca 84 s
CPU čas při spouštění (méně je lépe)	1,32 s	150,91 s
Využití paměti při spouštění (méně je lépe)	65 MB	1053 MB

na méně výkonné virtuální servery pro aplikace, které nepotřebují odbavovat velké množství uživatelů.

7.9 Shrnutí

Liferay se zdá být velmi dobrou platformou pro „standardní“ aplikace, takové, které mohou využít funkčnost obsaženou v systému bez větších změn a nevyžadují speciální chování, nestandardní vzhled nebo přílišnou různorodost rozhraní. Zvláště aplikace, pro které je Liferay primárně určen, tedy portály pro správu dat v organizaci, budou těžit z hotových aspektů jako uživatelů a uživatelských rolí, workflow a odpovědností, snadné definice entitních tříd, budování stránek nebo správy obsahu, což výrazně zrychlí vývoj. Jakmile však bude potřeba funkčnost výrazněji odlišná od té standardní, zvláštní manipulace s každým druhem entit nebo netriviální vzhled, změna nemusí být úplně snadná. Vadou na kráse je také složitější definice workflow pomocí XML souborů a celkový větší počet konfiguračních souborů roztroušených po adresářové struktuře.

Seaside je pouze webový framework, nenabízí proto předvytvořené aspekty běžných aplikací jako uživatelské role, přihlašování nebo modelovou vrstvu. Poskytuje však solidní základ pro rychlé budování i složitějších aplikací a vítězí ve srozumitelnosti, jednoduchosti a čistotě kódu. Bohužel neexistuje univerzální vrstva pro Pharo a Seaside, která by nabídla již předvytvořené zmíněné aspekty a tím ještě více zrychlila vývoj.

Závěr

Cílem této práce bylo navrhnout, implementovat a otestovat webovou aplikaci pro párování pracovníků a nabídek práce. Tento cíl nebyl zcela naplněn, jelikož se nepodařilo v rámci práce dokončit implementaci, zejména prezentační vrstvy. Pro ni do okamžiku odevzdání práce ještě nebyly hotovy návrhy uživatelského rozhraní a požadovaného chování. Dokončen tak byl kompletní návrh systému a z větší části také vrstva modelu a metamodelu včetně metamodelovacího frameworku.

Druhým cílem bylo porovnání platformy Pharo a frameworku Seaside s procesně-orientovanou platformou Liferay. Z něj nevyšel jednoznačný vítěz, obě platformy mají svá specifika a hodí se pro jiné způsoby použití. Platforma Pharo vyhrála jednoduchostí a srozumitelností, Liferay naopak nabídl velké množství předvytvořených aspektů, které ve Pharu a Seaside nejsou v základu dostupné.

Výhled do budoucna

Jelikož nebyla implementace zcela dokončena v rámci této práce, bude následovat právě její dokončení. Stejně tak dojde k důkladnému otestování zadavatelem práce včetně uživatelského testování. Poté bude aplikace nasazena do ostrého provozu. Zkušenosti uživatelů, ale i správců budou dále využity při rozšiřování a zlepšování aplikace, další směřování tedy bude záviset zejména na nich. Vzhledem k tomu, že použitá platforma se ukázala jako velice vhodná, neměly by s dalším vývojem nastat žádné výrazné problémy.

Přínos pro platformu Pharo

Platforma Pharo je poměrně mladá, stále se rozvíjející, a proto na svém kontě zdaleka nemá tolik úspěšných projektů jako některé jiné platformy nebo implementace Smalltalku. Ostré spuštění aplikace navržené v této práci tak bude

pro Pharo dalším cenným zářezem. Tyto takzvané „success stories“ velmi pomáhají v budování jména a hlavně důvěry, jelikož každý potenciální zájemce si může hned zkontrolovat, že platforma je odzkoušena reálným provozem mnoha aplikací.

Osobní přínos

Mně osobně práce přinesla zejména další hodnotné zkušenosti s platformou Pharo, frameworkem Seaside, ale i dalšími knihovnami. Mimo to, pokud se podaří aplikaci úspěšně nasadit a provozovat, se bude jednat o velmi cennou referenci. V neposlední řadě mi provedená práce pomáhá dokazovat, že myšlenky jazyka Smalltalk jsou skutečně nadčasové a i po tak dlouhé době od vzniku stále dokáže nejen držet krok s novějšími jazyky a platformami, ale v některých ohledech je dokonce předčit tak, že pořád lze pozorovat využití Smalltalku jako zdroje inspirace pro jejich další vývoj.

Použité zdroje

[1] ALPERT, S. R., K. BROWN a B. WOOLF: *The Design Patterns Smalltalk Companion*. The Software Patterns Series, Reading, MA: Addison-Wesley, první vydání, February 20, 1998, ISBN 978-0201184624.

[2] APPLE INC.: *Safari 8.0.3* [software]. [přístup 2015-03-07]. Dostupné z: `https://www.apple.com/safari`

[3] BECK, K.: *Smalltalk Best Practice Patterns*. Prentice Hall, první vydání, 1996, ISBN 978-0134769042.

[4] BLACK, A. P., S. DUCASSE, O. NIERSTRASZ aj.: *Pharo by Example*. Kehrsatz: Square Bracket Associates, první vydání, October, 2009, ISBN 978-3-9523341-4-0.

[5] BRYANT, A.: *Where are the templates?* [online]. March 26, 2004, [cit. 2015-03-29]. Dostupné z: `http://www.cincomsmalltalk.com/userblogs/avi/blogView?entry=3257728961`

[6] BRYANT, A., J. FITZELL, L. RENGGLI aj.: *Seaside 3.1* [software]. [přístup 2014-11-14]. Dostupné z: `http://seaside.st/`

[7] DEMEYER, S., S. DUCASSE a O. NIERSTRASZ: *Object-Oriented Reengineering Patterns*. San Francisco, CA: Morgan Kaufmann, první vydání, 2002, ISBN 1558606394.

[8] DUCASSE, S., A. LIENHARD a L. RENGGLI: Seaside: A Flexible Environment for Building Dynamic Web Applications. *IEEE Software*, ročník 24, č. 5, September 2007: s. 56–63, ISSN 0740-7459, doi:10.1109/MS.2007.144. Dostupné z: `http://scg.unibe.ch/archive/papers/Duca07a-SeasideIEEE-SCG.pdf`

[9] DUCASSE, S., L. RENGGLI, C. D. SHAFFER aj.: *Dynamic Web Development with Seaside*. Kehrsatz: Square Bracket Associates, první vydání, August, 2010, ISBN 978-3-9523341-1-9.

[10] FILONOV, M.: *Pharocloud* [online]. [2015], [cit. 2015-04-10]. Dostupné z: http://www.pharocloud.com/

[11] FILONOV, M.: *Pharocloud: TERMS OF SERVICE* [online]. 24.12.2013, [cit. 2015-04-10]. Dostupné z: http://assets.rackincloud.com/public/rack/TermsOfService.pdf

[12] FITZELL, J.: *Seaside History* [online]. September 20, 2008, [cit. 2015-03-29]. Dostupné z: http://blog.fitzell.ca/2008/09/seaside-history.html

[13] GAMMA, E., R. HELM, R. JOHNSON aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, první vydání, 1994, ISBN 0-201-63361-2.

[14] GOOGLE INC.: *Google Chrome 41.0* [software]. [přístup 2015-03-07]. Dostupné z: https://www.google.com/chrome

[15] HERLIHY, P.: *How many people are missing out on JavaScript enhancement?* [online]. 21 October 2013, [cit. 2015-04-01]. Dostupné z: https://gds.blog.gov.uk/2013/10/21/how-many-people-are-missing-out-on-javascript-enhancement/

[16] JQUERY FOUDNATION, INC.: *jQuery 1.11.0* [software]. [přístup 2014-11-14]. Dostupné z: https://jquery.org/

[17] LIFERAY INC.: *Liferay Portal* [software]. [přístup 2015-04-13]. Dostupné z: https://www.liferay.com/

[18] LIFERAY INC.: *Liferay Portal 6.2 Developer's Guide* [online]. [2013], [cit. 2015-04-13]. Dostupné z: https://www.liferay.com/documentation/liferay-portal/6.2/development

[19] LIFERAY INC.: *Using Liferay Portal 6.2* [online]. [2013], [cit. 2015-04-13]. Dostupné z: https://www.liferay.com/documentation/liferay-portal/6.2/user-guide

[20] LIFERAY INC.: *Get Liferay Portal: Comparison* [online]. [2015], [cit. 2015-04-13]. Dostupné z: https://www.liferay.com/downloads/liferay-portal/overview

[21] LIFERAY INC.: *Liferay Community* [online]. [2015], [cit. 2015-04-13]. Dostupné z: https://www.liferay.com/community

[22] LIFERAY INC.: *Liferay Portal Features* [online]. [2015], [cit. 2015-04-13]. Dostupné z: https://www.liferay.com/products/liferay-portal/tech-specs

[23] LIFERAY INC.: *Liferay Portal Technical Specifications* [online]. [2015], [cit. 2015-04-13]. Dostupné z: https://www.liferay.com/products/liferay-portal/tech-specs

[24] MEYER, B.: *Object-Oriented Software Construction*. Prentice Hall, první vydání, 1988, ISBN 0-13-629049-3.

[25] MICROSOFT CORPORATION: *Internet Explorer 11.0* [software]. [přístup 2015-03-07]. Dostupné z: http://ie.microsoft.com/

[26] MINISTERSTVO FINANCÍ ČR: *Administrativní registr ekonomických subjektů* [online]. [2012], poslední aktualizace 25.5.2012, [cit. 2015-03-22]. Dostupné z: http://wwwinfo.mfcr.cz/ares/

[27] MINISTERSTVO VNITRA ČR: *Adresy v České republice* [online]. [2015], [cit. 2015-03-22]. Dostupné z: http://aplikace.mvcr.cz/adresy/

[28] MIRANDA, E.: The Cog Smalltalk Virtual Machine. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, New York: ACM, 2011, ISBN 978-1-4503-1183-0, [cit. 2015-03-30]. Dostupné z: http://design.cs.iastate.edu/vmil/2011/papers/p03-miranda.pdf

[29] MONGODB, INC.: *MongoDB* [software]. [přístup 2015-04-08]. Dostupné z: https://www.mongodb.org/

[30] MOZILLA FOUNDATION: *Mozilla Firefox 36.0.1* [software]. [přístup 2015-03-07]. Dostupné z: https://mozilla.org/firefox

[31] NETSTYLE.CH GMBH: *Seaside-Hosting* [online]. [2015], [cit. 2015-04-10]. Dostupné z: http://www.seasidehosting.st/

[32] OPERA SOFTWARE ASA: *Opera 27* [software]. [přístup 2015-03-07]. Dostupné z: http://www.opera.com/

[33] OWASP FOUNDATION: *Cross-Site Request Forgery (CSRF)* [online]. 10 March 2015, [cit. 2015-04-10]. Dostupné z: https://www.owasp.org/index.php?title=Cross-Site_Request_Forgery_%28CSRF%29&oldid=191123

[34] OWASP FOUNDATION: *OWASP Top 10 – 2013* [online]. 2013, [cit. 2015-04-10]. Dostupné z: https://www.owasp.org/images/f/f3/OWASP_Top_10_-_2013_Final_-_Czech_V1.1.pdf

[35] OWASP FOUNDATION: *About The Open Web Application Security Project* [online]. 23 March 2015, [cit. 2015-04-10]. Dostupné z: https://www.owasp.org/index.php?title=About_The_Open_Web_Application_Security_Project&oldid=191949

[36] OWASP FOUNDATION: *Types of Cross-Site Scripting* [online]. 29 October 2013, [cit. 2015-04-10]. Dostupné z: https://www.owasp.org/index.php?title=Types_of_Cross-Site_Scripting&oldid=161960

[37] PHARO PROJECT, INRIA aj.: *Pharo 3.0* [software]. [přístup 2014-11-14]. Dostupné z: http://pharo.org/

[38] RENGGLI, L., aj.: *Magritte 3* [software]. [přístup 2014-11-14]. Dostupné z: https://code.google.com/p/magritte-metamodel/

[39] SCHNEIER, B.: *The Curse of the Secret Question* [online]. February 11, 2005, [cit. 2015-04-09]. Dostupné z: https://www.schneier.com/blog/archives/2005/02/the_curse_of_th.html

[40] SCHNEIER, B.: *Balancing Security and Usability in Authentication* [online]. February 19, 2009, [cit. 2015-04-09]. Dostupné z: https://www.schneier.com/blog/archives/2009/02/balancing_secur.html

[41] SCHÄRLI, N., S. DUCASSE, O. NIERSTRASZ aj.: Traits: Composable Units of Behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, Berlin Heidelberg: Springer Verlag, 2003, ISBN 978-3-540-40531-3, ISSN 0302-9743, s. 248–274, doi: 10.1007/b11832, [cit. 2015-03-30]. Dostupné z: http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf

[42] SLÁDEK, J.: *Graceful degradation vs. progressive enhancement* [online]. 4.3.2009, [cit. 2015-03-31]. Dostupné z: http://www.zdrojak.cz/clanky/graceful-degradation-vs-progressive-enhancement/

[43] TAGUE, N. R.: *The Quality Toolbox*. Milwaukee, WI: ASQ Quality Press, druhé vydání, March 30, 2005, ISBN 978-0-87389-639-9.

[44] WARTEL, R.: *Web applications security: risks and countermeasures* [online]. Nov 27, 2007, [cit. 2015-04-09]. Dostupné z: http://cerncourier.com/cws/article/cnl/31988

[45] WORLD WIDE WEB CONSORTIUM: *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification [online]*. 07 June 2011, [cit. 2015-03-31]. Dostupné z: http://www.w3.org/TR/2011/REC-CSS2-20110607/

[46] WORLD WIDE WEB CONSORTIUM: *HTML5 [online]*. 28 October 2014, [cit. 2015-03-31]. Dostupné z: http://www.w3.org/TR/2014/REC-html5-20141028/

[47] ZAKAS, N. C.: *How many users have JavaScript disabled?* [online]. Oct 13, 2010, [cit. 2015-04-01]. Dostupné z: https://developer.yahoo.com/blogs/ydn/many-users-javascript-disabled-14121.html

Příloha **A**

Seznam použitých zkratek

AJAX Asynchronnous JavaScript And XML

ARES Administrativní registr ekonomických subjektů

CPU Central Processing Unit

CRUD Create–Read–Update–Delete

CSRF Cross-Site Request Forgery

CSS Cascading Style Sheets

DNS Domain Name System

DOM Document Object Model

DRY Don't Repeat Yourself

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IaaS Infrastructure as a Service

IČO identifikační číslo osoby

IDE Integrated Development Environment

IP Internet Protocol

MB megabajt

MITM Man-in-the-Middle

MVC Model–View–Controller

MVP Model–View–Presenter

NoSQL Not only Structured Query Language

OWASP The Open Web Application Security Project

PaaS Platform as a Service

PHP PHP: Hypertext Preprocessor

PSČ poštovní směrovací číslo

REST Representational State Transfer

SQL Structured Query Language

SRP Single Responsibility Principle

SSP Portál spolupráce s průmyslem

TLS Transport Layer Security

UML Unified Modeling Language

VNC Virtual Network Computing

XHTML Extensible HyperText Markup Language

XML Extensible Markup Language

XSS Cross-Site Scripting

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `media` ..... obrázky použité v textu
- `text` ..... text práce
  - `DP_Balda_Michal_2015.pdf` ..... text práce ve formátu PDF
  - `DP_Balda_Michal_2015.ps` ..... text práce ve formátu PS