INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Platforma průmyslové spolupráce

CZ.1.07/2.4.00/17.0041

Název

Aplikace analytický vzorů při vývoji IS pro malý podnik

Popis a využití

- studijní materiál pro studium architektury rozsáhlých aplikací
- analytické vzory a jejich použití
- výuka: pokročilá Java

Jazyk textu

- český

Autor (autoři)

- Tomáš Skopal

Oficiální stránka projektu:

- http://lasaris.fi.muni.cz/pps

Dostupnost výukových materiálů a nástrojů online:

- http://lasaris.fi.muni.cz/pps/study-materials-and-tools

Obsah

1. Úvod 1
2. Analýza 2
  2.1 Požadavky na systém 2
    2.1.1 Funkční požadavky 3
    2.1.2 Nefunkční požadavky 6
3. Návrhové vzory 6
  3.1 Analytické vzory 7
4. Analytické vzory podle domén 8
  4.1 Odpovědnost 8
    4.1.1 Vzor Skupina 8
    4.1.2 Vzor Hierarchie organizace 9
    4.1.3 Vzor Heterogenní seznam 10
  4.2 Dohled a měření 10
    4.2.1 Vzor Veličina 10
    4.2.2 Vzor Měření 11
  4.3 Dohled nad podnikovými financemi 11
    4.3.1 Vzor Rozsah 11
  4.4 Odkazy na objekty 12
  4.5 Účetnictví a evidence 12
    4.5.1 Vzor Účet 13
    4.5.2 Vzor Transakce 13
  4.6 Plánování 13
    4.6.1 Vzor Plánovaná a realizovaná akce 14
    4.6.2 Vzor Plán 14
5. Nasazení vzorů do modelu 15
  5.1 Aplikace vzorů 15
    5.1.1 Náhodná aplikace vzorů 15
    5.1.2 Systematická aplikace vzorů 16
  5.2 Měření 17
  5.3 Účet 19
  5.4 Rozsah 20
  5.5 Hierarchie organizace 21

5.6 Akce ... 23
5.7 Plán a Heterogenní seznam ... 24
5.7.1 Průběh synchronizace ... 24
6. Použité technologie ... 26
6.1 Java EE ... 27
6.2 Technologie datové vrstvy ... 27
6.3 Spring ... 29
6.4 Wicket ... 30
6.5 Dozer ... 32
6.6 Maven ... 33
7. Závěr ... 34
Literatura ... 36
Rejstřík ... 38
Přílohy ... 39
Příloha A - Funkční požadavky ... 39
Příloha B - Textová specifikace případů užití ... 40
Příloha C - Elektronické přílohy ... 44

1. Úvod

Díky software je dnes možné jednoduše vytvořit věci ještě před deseti lety nemyslitelné. Informační a operační systémy, sociální sítě, to vše jsou programy obsahující miliony řádků. Takovýto software nám umožňuje efektivněji pracovat, avšak cesta k jeho vzniku není vůbec jednoduchá. To, jak správně vybudovat software, zkoumá a řeší obor „softwarové inženýrství“ [7]. Od jeho vzniku v 70. letech se objevilo několik modelů, jak by měl vývoj software vypadat. Byly definovány tzv. softwarové procesy: „Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla.“ [6].

Jedním z těchto kroků je analýza. Analýza má za cíl ze specifikace požadavků na systém vytvořit analytický diagram tříd, který je použit jako základ při implementaci. Analýzu tvoří velké množství technik a metod jak jí správně provést. Zmínil bych rozbor textu, tvorbu modelu nebo používání vzorů. Právě použitím vzorů v průběhu analýzy systému se zabývá tato práce.

Cílem práce bylo použít analytické návrhové vzory při analýze a implementaci malého informačního systému. Konkrétně se jedná o systém pro plánování směn stevardů autobusových linek pro společnost Student Agency k.s. Výsledkem práce pak je seznam použitých vzorů, které jsou podrobně rozebrány, a je diskutován jejich přínos do systému.

Práce je rozdělena na čtyři hlavní části. Nejdříve je analýza obsahující specifikaci systému v podobě digramu užití. Také jsou zde popsány a zasazeny do kontextu analytické návrhové vzory. Druhá část práce se sestává z obecné charakteristiky konkrétních vzorů. Základním zdrojem informací o uvedených vzorech je kniha Martina Flowera [1], která je dosud považována za hlavní zdroj informací v oblasti analytických návrhových vzorů. Třetí a zároveň největší částí práce je samotná aplikace vzorů v informačním systému. U každého vzoru je uveden popis, jak byl nasazen a co je jeho přínosem. Je zde také nastíněna možnost použití vzorů podle domén. Na konec jsou popsány použité technologie a stručně ukázáno uživatelské rozhraní.

Aplikace je nazvána jako „ShiftPlanner“. Je reálně používána zaměstnanci společnosti a je dostupná z webu. Celkově použití vzorů zpřehlednilo výsledný diagram tříd. Jistou mírou přispělo k rozšiřitelnosti a udržovatelnosti aplikace, i když jsem předpokládal, že tento přínos bude markantnější.

2. Analýza

Většina knih o zabývajících se softwarovým inženýrstvím nebo objektovým modelováním, mluví o analýze a návrhu. Jen zřídka je specifikována hranice mezi těmito dvěma pojmy. Velice zjednodušeně můžeme říci, že analýza znamená „co je třeba udělat“ a návrh je „jak to bude fungovat“.

Důležitým principem v objektovém vývoji je navrhovat software tak, aby jeho struktura odrážela podstatu problému. Pokud děláme analýzu, snažíme se porozumět problému. Součástí analýzy jsou samozřejmě diagramy užití (use case), ale tím to nekončí. „Analýza také spočívá v rozkrytí požadavků, pochopení základních souvislostí a sestavení modelu, který odpovídá tomu, jak věci fungují.“ [1] Součástí analýzy je také vytvoření struktury projektu zahrnující navržení vrstev, konceptu zabezpečení, uživatelského rozhraní atd. Naproti tomu návrh aplikace spočívá v navržení konkrétní implementace.

Analýza je jednou z klíčových součástí životního cyklu softwaru. Je to základní stavební kámen, na kterém závisí úspěch projektu. Obvykle začíná u myšlenek a nápadů. Pokračuje přes specifikaci požadavků, vytvoření diagramu užití a je ukončena návrhem analytického diagramu tříd. [9]

2.1 Požadavky na systém

Na začátku vývoje softwaru stojí specifikace požadavků. Nejčastěji se jedná o popis funkcí, které by budoucí software měl zvládat a jak by se měl systém chovat. U větších systémů, které obsahují stovky tříd, je i tato na první pohled jednoduchá věc otázkou několika týdnů práce. Jasně totiž definují, co má být implementováno. [10] Požadavky na systém můžeme obecně dělit do několika kategorií. Uvedl bych dvě základní.

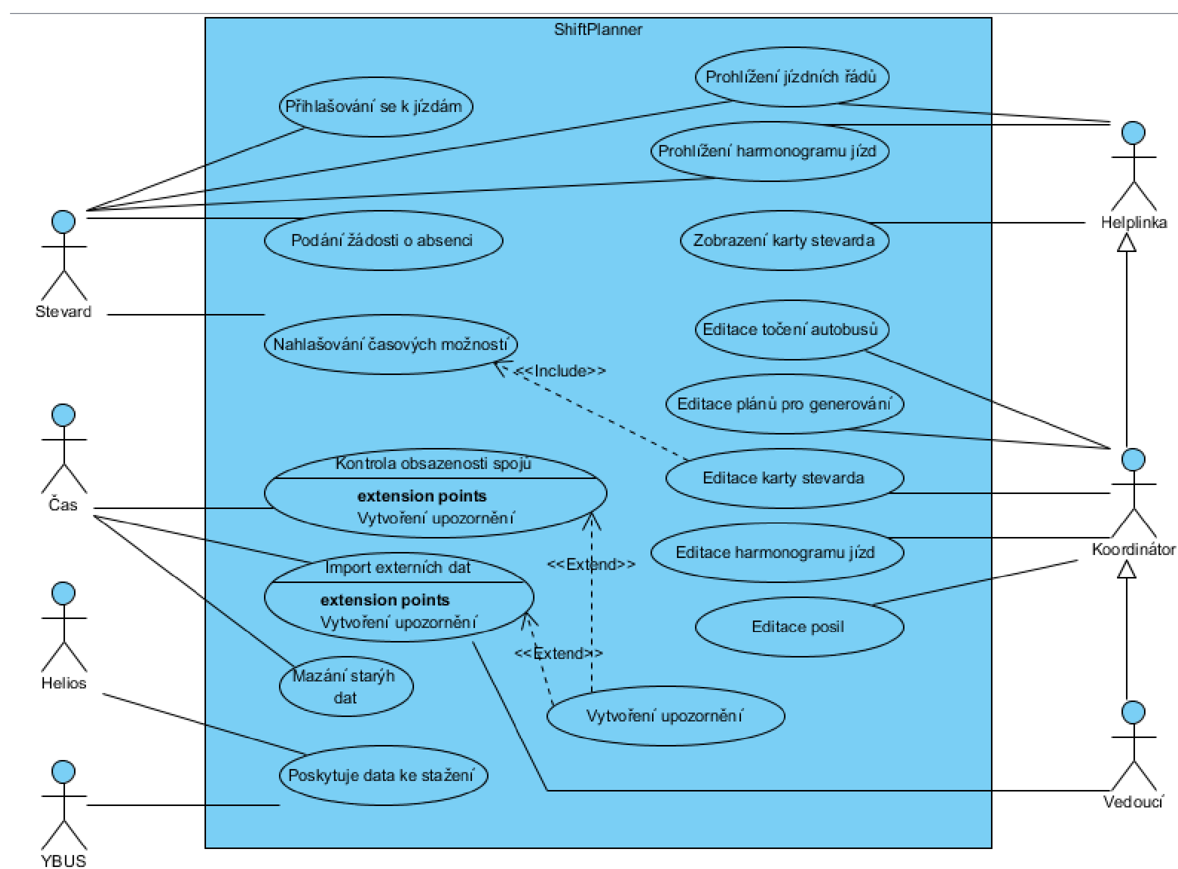
První kategorií jsou funkční požadavky. Již podle názvu je patrné, že popisují funkcionalitu systému. Jsou vyjádřením toho, co by měl systém dělat, respektive nedělat. Pro přehlednost se doporučují zapisovat ve tvaru *<id><aktér/systém><funkce>*. Jako příklad takového požadavku bych uvedl „ 1. Bankomat ověří platnost kreditní karty“. Kde vidíme rozdělení na jednoznačný identifikátor, určení aktéra nebo systému, na který se funkcionalita vtahuje a na akci, která se vykonává. Funkční požadavky se někdy zapisují jako souvislý text, ale není to doporučováno.

Druhou kategorií jsou nefunkční požadavky na systém. Jedná se o omezení kladená na daný systém. Často se jedná o omezení na výkon, dostupnost, bezpečnost, vzhled nebo vyhovění standardům. Někdy se dělí na podkategorie, jako jsou výkonnostní požadavky

nebo designové požadavky. U menších systémů si však bohatě vystačíme s pojmem nefunkční požadavky. Jako příklad takového požadavku bych uvedl „1. Bankomat ověří platnost kreditní karty za méně než 3 sekundy a bude implementovaný v C++“.

2.1.1 Funkční požadavky

Hlavní úlohou aplikace „ShiftPlanner“ (dále uváděno bez uvozovek) je zlepšení a zjednodušení práce koordinátorům autobusové dopravy společnosti *Student Agency, k.s.* Systém má být koncipován jako webová aplikace. Nevyhovující stav, kdy koordinátoři musejí pracovat s desítkami tabulkových souborů, má být nahrazen jednoduchým funkčním rozhraním. Seznam požadavků je relativně objemný. Hlavní body jsou znázorněny na diagramu užití (obr 2.1). Několik je jich detailně rozepsáno pod diagramem. Zbytek požadavků je uveden v příloze (Příloha B - Textová specifikace případů užití).



Obrázek 2.1: Zachycení požadavků na systém ve formě diagramu užití.

Do systému mají přístup čtyři skupiny uživatelů. Konkrétně mají role nazvané Vedoucí, Koordinátor, Stevard a Helplinka. Tyto role je potřeba důsledně rozlišovat. Jak je vidět z obrázku 2.1 každá z rolí má více funkcí, níže jsou specifikovány ty nejdůležitější:

- **Helplinka** – Má přístup ke většině sekcí, protože v případě problému musí pracovník helplinky podat maximální množství dostupných informací. Důležité je, že nemůže nic editovat.
- **Koordinátor** – Plánuje směny, tudíž má přístup jak k harmonogramům jízd na všech linkách, které může editovat, tak k osobním kartám stevardů.
- **Vedoucí** – Může nahradit v práci koordinátora. Navíc má právo na některé systémové operace. Například jednorázové stažení informací o stevardech ze systému Helios.
- **Stevard** – Zobrazuje si svůj plán jízd a může se dohlásit k volným směnám, které korespondují s linkou, na které pravidelně jezdí.

Dalšími aktéry, kteří komunikují se systémem, jsou YBUS a Helios. Helios slouží ke správě osob, mezd, dokladů, tedy poskytuje data o stevardech. YBUS naproti tomu poskytuje data o spojích a jízdních řádech. Posledním aktérem, který hraje nemalou roli, je Čas. Plánované úlohy se starají o synchronizaci dat s výše zmíněnými systémy, kontrolují obsazenost spojů nebo mažou stará data.

Vybrané případy užití:

Editace točení autobusů – slouží k vytvoření obecného nebo konkrétního točení autobusů. Konkrétní točení se týká především státních svátků.

Aktéři: Koordinátor, Vedoucí

Vstupní podmínky: Z firemního systému YBUS musí být staženy jízdní řády. Uživatel musí být přihlášený.

Tok událostí:

1. Případ užití (PU) začíná, když uživatel klikne na tlačítko "trasy autobusu" v levém menu aplikace.
2. Uživatel pokračuje kliknutím na "točky" u určité skupiny linek.
3. Uživatel si vybere, jestli chce editovat obecné točení autobusů v sudý nebo lichý týden nebo vybere konkrétní časový úsek, kdy může být točení jiné.
4. Systém zobrazí jízdní řád podle vybraného kritéria.
5. Uživatel do políček nad časové údaje vyplní čísla autobusů. Ta slouží systému k spárování jednotlivých spojů.
6. Uživatel stiskne tlačítko "uložit".
   IF uživatel zaškrtnul políčko pro vyplnění sudého i lichého týdne naráz
   6.1. Systém zkopíruje údaje pro sudý / lichý týden v závislosti na aktuálně zvoleném.
7. PU končí, když systém uloží čísla do databáze.

Točení autobusů spočívá v efektivnosti pokrytí všech spojů v jízdním řádu co nejmenším počtem autobusů. Je tedy potřeba spárovat jednotlivé časy v jízdním řádu, aby na sebe odjezdy a příjezdy dobře navazovaly. Může je pak jet jeden autobus. Také je snaha o to, aby

s jedním autobusem jel jeden stevard po celou dobu jeho denního provozu, aby se nemuseli zbytečně stevardi střídat a složitě přestupovat. Dobré nastavení točení autobusů této snaze přispívá.

Uvedu příklad. Na následujících obrázcích je vidět část jízdního řádu na lince Praha – Brno. Obrázek 1.2 ukazuje jízdní řád s údaji o výjimkách. Na obrázku 1.3 je vidět jízdní řád jak je zobrazen v editaci točení autobusů.

	x 6	x 6		x 6				x 6							
		44		44				44	22						
Brno AN	05:00	05:30	06:00	06:30	07:00	07:30	08:00	08:30	09:00	09:30	10:00	10:30	11:30	12:00	12:30
Praha ÚA	07:20	08:15	08:30	09:00	09:30	10:15	10:30	11:00	11:30	12:15	12:30	13:00	14:15	14:30	15:00

	x 6	x 6			x 6						x 6				
		44			44				22		44				
Praha ÚA	05:00	05:30	06:00	07:30	08:00	09:00	09:30	10:00	11:00	11:30	12:00	12:30	13:00	13:30	14:00
Brno AN	07:30	08:15	08:30	10:15	10:30	11:30	12:15	12:30	13:30	14:15	14:30	15:00	15:30	16:15	16:30

Obrázek 1.2: Jízdní řád autobusů. Čísla nad časy znamenají výjimky (např. nejde od-do).

PONDELI - tyden 1

	144		130	193	143	140	128	134		129	145	133	132	146	136	137	135	144
Brno AN	05 : 00	05 : 00	05 : 30	06 : 00	06 : 30	07 : 00	07 : 30	08 : 00	08 : 30	08 : 30	09 : 00	09 : 30	10 : 00	10 : 30	11 : 30	12 : 00	12 : 30	13 : 30
Praha ÚA	07 : 20	07 : 20	08 : 15	08 : 30	09 : 00	09 : 30	10 : 15	10 : 30	11 : 00	11 : 00	11 : 30	12 : 15	12 : 30	13 : 00	14 : 15	14 : 30	15 : 00	16 : 15

	145	133	132	136	137	135	144	130	193	143	140		128	134	129	145	132	133
Praha ÚA	05 : 00	05 : 30	06 : 00	07 : 30	08 : 00	09 : 00	09 : 30	10 : 00	11 : 00	11 : 30	12 : 00	12 : 00	12 : 30	13 : 00	13 : 30	14 : 00	15 : 00	15 :
Brno AN	07 : 30	08 : 15	08 : 30	10 : 15	10 : 30	11 : 30	12 : 15	12 : 30	13 : 30	14 : 15	14 : 30	14 : 30	15 : 00	15 : 30	16 : 15	16 : 30	17 : 30	18 :

Obrázek 1.3: Točky autobusů. Čísla nad časy párují spoje. To znamená že je jede jeden autobus a ideálně jeden stevard.

Editace harmonogramu jízd – Obnáší přidávání a odebírání stevardů z harmonogramu jízd, který slouží jako přehled všech směn. Je rozdělen podle skupin linek kvůli přehlednosti. Programové přidávání stevardů má oproti současnému řešení (tabulky v Excelu) hlavní výhodu v možnosti detekce kolizí.

Primární aktéři: Koordinátor
Sekundární aktéři: Vedoucí, Stevard
Vstupní podmínky: Z firemního systému YBUS musí být staženy spoje. Uživatel musí být přihlášený. Musí být sestaveno točení autobusů pro platný jízdní řád.
Tok událostí:

1. PU začíná, když uživatel klikne na tlačítko "trasy autobusu" v levém menu aplikace.
2. Uživatel pokračuje kliknutím na "harmonogram jízd" u určité skupiny linek.
3. Uživatel si vybere konkrétní časový úsek, ve kterém chce pracovat se spoji.
4. Systém zobrazí spoje sdružené podle točení autobusů po jednotlivých dnech.
5. Uživatel vybere stevarda ze seznamu a přidá jej k celé "točce" nebo jen k několika spojům

OR

5. Uživatel odebere stevarda.
6. Systém aktualizuje seznam a přidá nebo odebere stevardovi příslušný počet hodin z je-ho

měsíčního pracovního fondu.

Alternativní tok událostí:

6. Systém detekuje některou z kolizí (stevardovi se kryjí časy směn, stevard překročí měsíční limit odpracovaných hodin)
7. Systém vypíše varovnou hlášku s důvodem kolize a nic neuloží.

2.1.2 Nefunkční požadavky

Nejen kvůli velké dynamičnosti společnosti, ale také kvůli již existujícím požadavkům do budoucnosti, je systém potřeba vybudovat flexibilní. Předpokládá se, že v budoucnu bude aplikaci spravovat více než jeden člověk. Dalšími požadavky na systém jsou:

1. Systém je implementován v jazyce Java (backend[1]) a ve vývojovém rámci Wicket (frontend[2])
2. Veškerá funkcionalita je dostupná až po přihlášení uživatele.
3. Je důležité precizní rozdělení oprávnění. V systému mají být rozlišeny čtyři role (Vedoucí, Koordinátor, Stevard, Helplinka), které určují, co uživatelé mohou nebo co vidí.

3. Návrhové vzory

Návrhové vzory představují soubor modelů, které napomáhají v řešení obecných problémů. Patří zde analytické vzory, vzory pro design, architektonické vzory a mnoho dalších. Primárním účelem vzorů je ulehčit rozbor situace a rozdělení systému na menší části, provést dekompozici. Pojem návrhové vzory pochází ze šedesátých let z architektury, kde jej použil Christopher Alexandr. V oblasti počítačů byly návrhové vzory poprvé použity v roce 1987 na konferenci OOPSLA. Tyto vzory byly určeny pro začátečníky v programovacím jazyce Smalltalk[3]. K většímu rozšíření návrhových vzorů přispěl v devadesátých letech vznik skupiny GoF[4] (Gang of Four), která sesbírala obecné zásady z tehdejšího vývoje softwaru a vytvořila několik vzorů používaných dodnes (např. Observer). Milníkem v této oblasti však bylo vydání knih Patterns: Elements of Reusable Object-Oriented Software od Martina Flowera [1] a Design Patterns: Elements of Reusable Object-Oriented Software od autorů skupiny GoF [2].

[1] Jako backend se označuje část webové aplikace, která slouží k administraci webu a ke zpracování dat.
[2] Pojem frontend slouží k označení části webu viditelné běžným návštěvníkům.
[3] Smalltalk je interpretovaný, dynamicky a striktně typovaný, čistě objektový programovací jazyk.
[4] GoF je pseudonym pro skupinu čtyř autorů: Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides.

3.1 Analytické vzory

Pro lepší pochopení pojmu *analytický návrhový vzor* uvedu příklad ze sportu. „Vezměme v úvahu bowling. Můžeme hru simulovat stovkami diagramů, které reflektují úhly, rychlosti, rotaci koule, ale to nám dá pouze povrchově nahlédnout do problematiky bowlingu. Lepší je prozkoumat zákony, kterými se pohyb na herní ploše řídí. Když pochopíme, proč dodání rotace kouli shodí více kuželek, můžeme se na problém podívat z jiného úhlu pohledu. Jde zde o zvolení správné míry abstrakce." [1, cit. 04.12.2013] Dělat důsledný rozbor a vzít v úvahu například všechny možné hody a váhy koulí je časově náročné a nepraktické. Naopak přílišná míra abstrakce však může systém znepřehlednit. Důsledné studium problému přináší velké množství informací. Zpřehlednění a zobecnění je v takovém případě více než na místě. K tomu nám pomáhají právě analytické návrhové vzory.

Obecně je návrhových vzorů velké množství, protože za vzor lze považovat téměř každé řešení, které bylo použito více než jednou v odlišných situacích. Martin Flower ve své knize rozdělil vzory do dvou kategorií. [1] První obsahuje analytické vzory a druhá pak vzory podpůrné, které lze použít při aplikaci analytických vzorů. Tato práce se zaměřuje na vzory z první části knihy. V knize je uvedeno 65 vzorů členěných do 9 oblastí (domén). Výčet domén je následující:

- Odpovědnost
- Dohled a měření
- Dohled nad podnikovými financemi
- Odkazy na objekty
- Účetnictví a evidence
- Plánování
- Obchodování
- Obchodování s deriváty
- Granularita velkých systémů

Jak domény, tak samotné vzory, autor knihy označuje spíše jako doporučení. Řešení určitého problému není nikdy jednoznačné a jen výjimečně se nachází jen v jedné doméně, pokud vůbec v nějaké. Dále autor uvádí, že vzory by měly sloužit jako návod k pochopení problému. K zachycení situace bychom si je měli libovolně přizpůsobit, pokud to považujeme za vhodné.

Analýza a sestavení analytického diagramu tříd by měl být výsledek kooperace softwarového inženýra a doménového odborníka. Odborníkem je zde myšlen člověk, který se pohybuje několik let v oblasti, do které bude software zasazen. Softwarový inženýr nikdy nemůže znát detailně problematiku a proto je tato spolupráce klíčová. Například lékař zná nemocniční prostředí jako nikdo jiný a nejlépe ví, co by měl software v lékařském prostředí obsahovat.

4. Analytické vzory podle domén

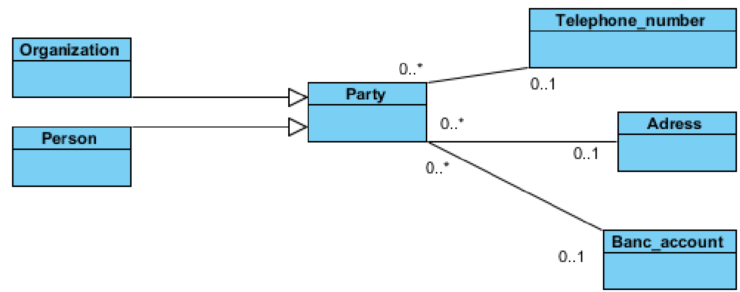
Tato kapitola je věnována popisu domén a stručnému představení některých vzorů. Většina textu je parafrází z knihy Martina Flowera, jelikož se jedná o teoretický úvod pro pochopení struktury jednotlivých vzorů. [1] Stejně tak diagramy demonstrující strukturu vzorů jsou překresleny z knihy, vyjma vzoru *Heterogenní seznam*, který v této knize není obsažen.

4.1 Odpovědnost

Odpovědnost (angl. Accountability) je koncept, kdy člověk nebo sdružení odpovídá za jiné. Je to velice obecná záležitost, použitelná v mnoha směrech. Lze zde zahrnout organizační struktury, kontrakty nebo například vztah zaměstnavatel zaměstnanec.

4.1.1 Vzor Skupina

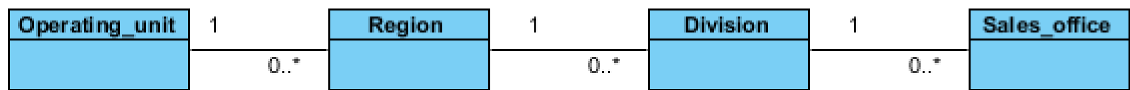
S výhodou lze vzor použít v případech, kdy dvě jinak odlišné entity vystupují v rovnocenné pozici. Vezměme například v úvahu fyzickou a právnickou osobu (společnost). V telefonu je stejně uložené číslo na konkrétního taxikáře i na taxikářskou ústřednu. Potřebujeme s číslem dělat stejné operace, ať patří společnosti nebo jednomu člověku. Platby na bankovní účty jsou dalším příkladem výhodnosti skrytí toho, jestli je na pozadí člověk nebo společnost. V takových případech je dobré používat pro operace nadtyp společnosti a člověka neboli *Skupina (angl. Party)*. I když není v aplikaci *ShiftPlanner* použit, jde o poměrně často využívaný vzor, proto jsem se rozhodl jej uvést. Jako jeden z prvních vzorů je uveden v případové studii eshopu [3] nebo v projektu „Aplikace návrhových vzorů“ Ostravské univerzity, který je součástí projektu Evropské unie [4].



Obrázek 4.1: Schéma vzoru *Skupina*

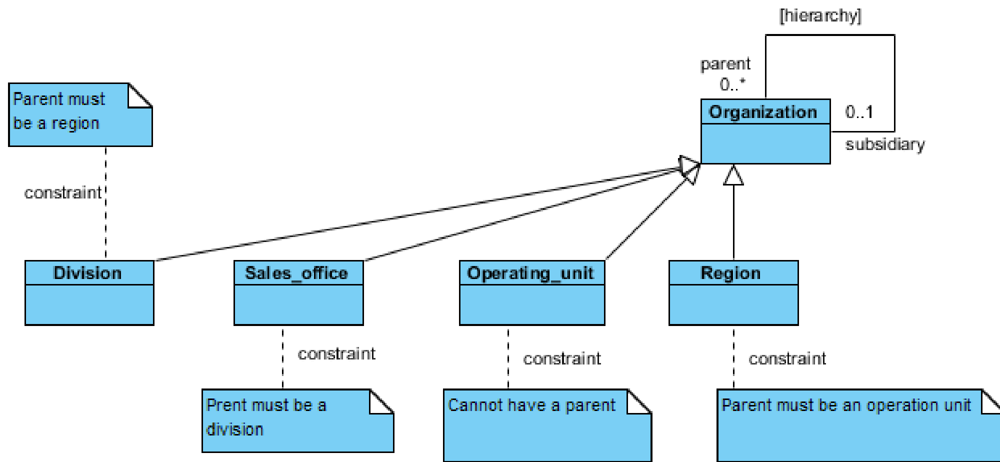
4.1.2 Vzor Hierarchie organizace

Vytvoření hierarchické struktury často vypadá tak, jak je znázorněno na obrázku 4.1.2. Takto utvořená struktura je velice náchylná k chybám při výpadku některých elementů. Pokud by byl region vyřazen, bylo by třeba přebudovat diagram.



Obrázek 4.1.2: Schéma před aplikací vzoru *Hierarchie organizace*.

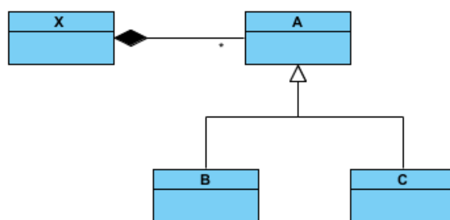
Lepší řešení je ukázáno na obrázku 4.1.3. Definování obecného předka umožňuje navrhnout hierarchii tak, že při změnách elementů není třeba předělat model. U tohoto návrhu je však třeba dbát na dodržení omezujících podmínek pro jednotlivé úrovně hierarchie, aby nedocházelo ke špatnému začlenění uzlu do hierarchie.



Obrázek 4.1.3: Schéma vzoru *Hierarchie organizace*. Každý z potomků má omezení definující jeho pozici v hierarchii.

4.1.3 Vzor Heterogenní seznam

Tento vzor není součástí knihy Martina Flowera. [1] Je popsán v knize „Analytické modelování IS pomocí UML v praxi“. [8] Vzor je kombinací dvou vztahů: *Kompozice 1: N* a *Generalizace (Dědičnost)*.



Obrázek 4.1.4: Schéma vzoru *Heterogenní seznam* s obecnými třídami. Obrázek převzat z knihy. [8]

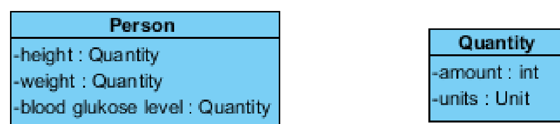
“V tomto případě třída typu X drží v seznamu prvky typu A, což v instanční rovině můžou být buď prvky typu B nebo C (pozn.: třída A na obrázku je abstraktní).“ [8, str. 73] Takto sestavený návrh má smysl pokud je potřeba v systému pohlížet na seznam prvků z pohledu obecného předka. Například pokud bychom měli třídu Zvíře, která by byla nadtypem pro Kočku a Psa, pak je využití generalizace na místě. Obecně totiž často mluvíme o zvířeti a je jedno jestli se konkrétně jedná o psa nebo kočku.

4.2 Dohled a měření

Objekty z reálného světa mají často svůj obraz ve světě virtuálním. Nejpřímočařejší cesta k uchování údajů o jejich vlastnostech je přidávat je jako atributy k entitám, které reprezentují objekty reálného světa. Vzory v doméně *Dohled a měření* ukazují jak vlastnosti objektů zachytit lépe a efektivněji než seznamem atributů. Také poskytují způsoby jak se získanými daty pracovat.

4.2.1 Vzor Veličina

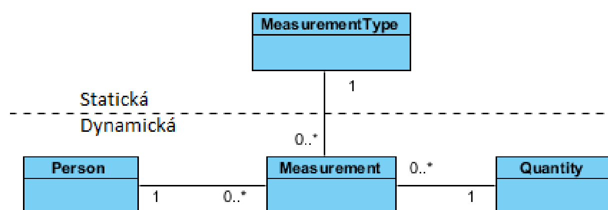
Jak už bylo řečeno, zjištěné hodnoty můžeme uchovávat přímo jako atributy objektů. Pokud se však jedná o komplexnější hodnotu jako například hmotnost, je vhodné zavést novou tabulku, ve které si k hodnotě přidáme I její veličinu, protože samotná atribut neudává nic o jednotce uložené hodnoty. Tato na první pohled triviální operace zajistí velkou míru abstrakce, protože můžeme jednoduše udávat hmotnost v kilogramech, librách nebo v jakékoli další jednotce. Také můžeme zavést elementární operace pro každý typ veličiny. Tento koncept popisuje vzor *Veličina (angl. Quantity)*.



Obrázek 4.2.1: Schéma vzoru *Veličina*.

4.2.2 Vzor Měření

Modelování veličin jakožto atributů (použití vzoru *Veličina*) je užitečné pokud máme menší množství veličin. Například v nemocničním prostředí, kde jeden pacient může ve své kartě mít tisíce údajů, pak je seznam atributů příliš dlouhý. Použitím vzoru *Měření* zredukujeme tento počet na pouhý jeden. Ve vzoru jsou zavedeny dvě úrovně. Jedna reprezentuje oblast, která se příliš nemění, tedy seznam druhů měření (*MeasurementType*). Druhá pak znázorňuje každodenní akci (*Measurement* a *Quantity*). Rozdělení vzoru dovoluje pohlédnout na problematiku z jiného úhlu pohledu. Toto rozdělení se často využívá u složitých návrhů.



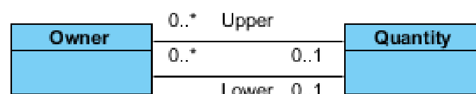
Obrázek 4.2.2: Schéma vzoru *Měření*.

4.3 Dohled nad podnikovými financemi

Obsahuje soubor vzorů používaných pro záznam faktů s důrazem na firemní finanční analýzu. Zejména pro větší společnosti platí, že pracují s velkými objemy dat a ta mohou zkomplikovat práci komukoli, kdo se je snaží analyzovat. Pro doplnění bych uvedl, že tato sada vzorů primárně popisuje rozdělení zkoumané oblasti do segmentů, pohledů a rozměrů. Ty se pak zpracovávají odděleně nebo společně podle toho jak to vyžaduje situace.

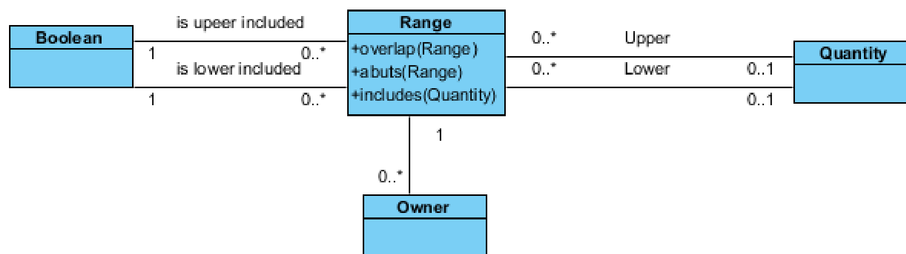
4.3.1 Vzor Rozsah

Poměrně často se setkáváme se situací, kdy je potřeba evidovat rozsah hodnot. Ať už se jedná o obyčejná čísla (1..10), data (1.1.2013..1.2.2013) nebo složené výrazy (10..15kg). Obvykle je v aplikacích rozsah hodnot určen pouze horní a spodní hranicí, jak je ukázáno na obrázku 4.3.



Obrázek 4.3: Obvyklé použití rozsahu

V takovém řešení nastává problém s určováním zda-li je konkrétní hodnota uvnitř nebo vně rozsahu, případně jestli se dva intervaly překrývají apod. Řešením je zavést třídu *Rozsah (angl. Range)*, která bude pokrývat veškerou zodpovědnost při porovnávání hodnot a intervalů. Obsahuje horní a dolní mez a informaci o tom jestli jsou tyto hodnoty součástí intervalu, či nikoli. Také definuje metody pro práci s intervaly. Schéma vzoru *Rozsah* je na obrázku 4.4.



Obrázek 4.4: Schéma vzoru *Rozsah*

4.4 Odkazy na objekty

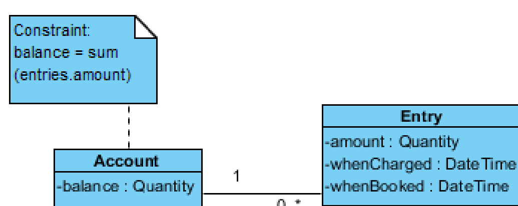
V objektově orientovaném návrhu aplikací má každý objekt unikátní ID, podle kterého jej lze snadno identifikovat. Tato ID neboli primární klíče spolu s cizími klíči, které jsou také hojně využívány, tvoří základní stavební kameny tradičního datového modelu. Někdy jsou ovšem zapotřebí i další způsoby identifikace objektů. Těm se věnuje doména vzorů *Odkazy na objekty (angl. Referring to Objects)*.

4.5 Účetnictví a evidence

Vzory z této domény slouží k monitoringu pohybu věcí a peněz mezi sklady. Skladem můžeme rozumět cokoli, kde je komodita, kterou evidujeme, uložena. Bankovním účtem počínaje, kartotékou u lékaře konče. Pohyb komodit má množství úskalí, které je potřeba zaznamenat. Například jak evidovat, že zboží bylo vyexpedováno, ale nebylo doručeno (vzor *Transakce*) nebo historii příjmů a výdajů (vzor *Účet*).

4.5.1 Vzor Účet

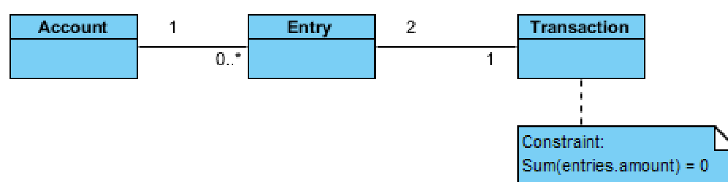
V mnoha případech je důležité ukládat nejen aktuální hodnotu, ale i veškeré změny, které jí ovlivnily. Typickým příkladem je bankovní účet (*Account*), kde je nutné evidovat všechny výběry i vklady. Podle obrázku je patrné, že třída *Entry* reprezentuje jednotlivé změny na účtu. Podmínka, která je na atributu *balance* je nutnou podmínkou. Neznamená to však, že by se při každé editaci *Entry* musela suma znovu přepočítat. Podrobnější popis této situace je uveden v kapitole 5.3. Účtem zde nemusí být myšlen jen bankovní účet. Může jít o cokoli, kde je něco uloženo. Například sklad, kde je obilí po tunách.



Obrázek 4.5.1: Schéma vzoru *Účet*.

4.5.2 Vzor Transakce

Použití vzoru *Účet* umožní uchovávání historie změn na účtu. Tyto změny ovšem obvykle znamenají, že se komodita na účet musela vzít z nějakého jiného účtu. Transakce explicitním spojením výběru z jednoho účtu a vkladu na jiný účet zaznamenají tuto akci. Důležitá je v tomto případě konzistentnost a pečlivá kontrola podmínky na nulový součet. Komodita se nesmí vytvořit ani ztratit, je pouze přesunuta z jednoho místa na druhé.



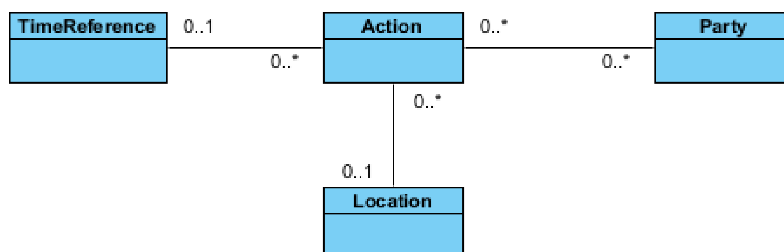
Obrázek 4.5.1: Schéma vzoru *Transakce*.

4.6 Plánování

Plánování je důležitou součástí každého velkého úkolu. Mnozí manažeři tráví většinu svého času vytvářením a sledováním plánů. Tato kapitola obsahuje některé základní vzory pro plánování.

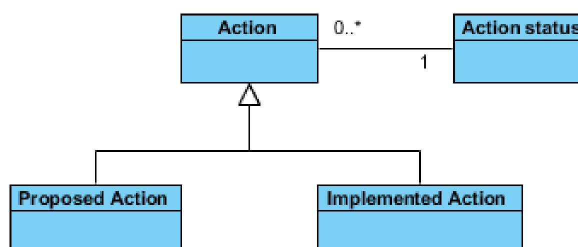
4.6.1 Vzor Plánovaná a realizovaná akce

Základem každého fungujícího systému je akce. Od základních jako je reakce na odeslání formuláře uživatelem po ty složité, například synchronizace dat nebo počítání statistik. Pojem „akce“ je velice široký. Může zahrnovat množství vlastností, založených na tom kdo, kdy a kde. S takto rozmanitými a mnohdy nesourodými vlastnostmi lze těžko pracovat. Proto se nejjednodušší schéma akce znázorňuje tak, jak je vidět na obrázku 4.6.1.



Obrázek 4.6.1: Přehled vlastností třídy *Action*.

Akce se většinou skládají do nějakých plánů. Ty mohou být plánovány, doplňovány o další akce. Mají někdy začátek a konec. Je obtížné relevantně zaznamenávat průběh takovýchto plánů, natož monitorovat jak se mění stav jednotlivých akcí v průběhu vykonávání. Proto když je akce vytvořena, spadá do tzv. naplánovaných akcí (*Proposed action*). Jakmile jednou začne, už patří mezi akce realizované (*Implemented action*). Nejde jen o změnu stavu, ale o vytvoření úplně nového objektu. Tak lze jednoznačně rozlišit plán od realizace. Objekt může být doplněn atributem odrážející jeho stav. Jde o zjednodušení v případě složitějších struktur, kde by bylo potřeba aktuální stav složitěji dohledávat.

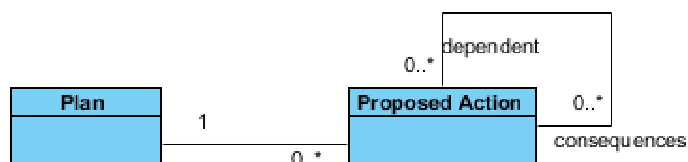


Obrázek 4.6.2: Schéma vzoru *Plánovaná a realizovaná akce*.

4.6.2 Vzor Plán

Ve své nejjednodušší podobě je plán kolekcí naplánovaných akcí, pospojovaných do určité posloupnosti. Nejčastěji je posloupnost reprezentována závislostmi – označením, že jedna

akce nemůže začít, pokud jiná neskončí. V diagramu tříd je u plánů vhodné zachytit také vztahy mezi třídami. Mimo to je doporučeno znázornit průběh plánu některým z interakčních diagramů (sekvenční, komunikační, …). Naopak, pokud je plán akcí hodně složitý, jsou interakční diagramy považovány spíše za zbytečnou práci navíc.[1]



Obrázek 4.6.3: Schéma vzoru *Plán*. Znázornění závislostí mezi plánovanými akcemi.

V následující kapitole se budu věnovat implementaci vzorů, jejich použití v reálném systému a vytvoření analytického diagramu tříd.

5. Nasazení vzorů do modelu

Tato kapitola obsahuje hlavní část práce, tedy samotné použití vzorů. Na začátku jsou uvedeny metodiky aplikace vzorů. Zbytek kapitoly je věnován jednotlivým vzorům. Vždy je uveden popis situace, případně doplněn o schéma tříd. Vzor je zhodnocen z hlediska přínosu pro systém.

5.1 Aplikace vzorů

Existuje nespočet popsaných vzorů, návrhových, analytických, softwarových, atd. Jejich použití se liší podle jejich podstaty, ale způsob jejich aplikace je v podstatě podobný. Vždy máme nějaký problém a snažíme se jej, pokud možno pomocí vhodného vzoru, vyřešit. Postupovat je možné několika způsoby. Těmto způsobům se věnuje kniha *Pattern-oriented Analysis and Design.* [5] Níže jsou popsány nejčastější z nich:

5.1.1 Náhodná aplikace vzorů

Použití vzoru je jednorázové a slouží k vyřešení konkrétního problému. Software, který je vyvíjen s tím, že bude s obměnami použitelný i v budoucnu, je často postaven tak, že základem aplikačního modelu jsou vzory. *Náhodná aplikace vzorů* není pro tento postup většinou vhodná.

[1] Bylo řečeno RNDR. Jaroslavem Ráčkem PhD. na přednášce předmětu *Softwarové inženýrství 2* na Fakultě informatiky MU. Semestr *podzim 2013.*

5.1.2 Systematická aplikace vzorů

Tato technika je postavená na zavedení postupů jak vzory aplikovat. Postupy mohou být dvojího druhu.

- Katalog vzorů (Pattern Language)

Představuje seznam vzorů rozdělených do domén. Výběrem domény se zúží spektrum vzorů a lze pak snadněji identifikovat vhodné vzory. Také katalog často uvádí vztahy mezi vzory. Typickým příkladem takového katalogu je kniha Martina Flowera, ze které vychází tato případová studie. Většinou jsou vzory aplikovány metodou „rozšíření modelu".

- Vývojový proces (Development Process)

Definuje postup pro systematickou aplikaci vzorů. Vzory jsou skládány jako základ aplikačního modelu. Tento proces je detailně popsán v knize. [5] V knize je uvedena metoda *Pattern-Oriented Analysis and Design* (POAD), jejíž základní princip je uveden níže.

Metoda rozšíření modelu

Ze specifikace požadavků se běžnými postupy vytvoří diagram tříd. Podle domény se pak vybere vzor nebo seznam vzorů, který by mohl být použit. Procházením diagramu tříd se hledá nejvhodnější místo pro použití vzoru. Dalo by se také říci, že se hledá taková část modelu, kde se vyskytuje problém, který lze vzorem vyřešit. Tato metoda je opakem k *náhodné aplikaci vzoru,* kde nejdříve máme problém až poté vzor.

Metoda POAD

Nejdříve je potřeba určit s jakými vzory metoda počítá a k čemu je určena. Tedy hlavním úkolem metody POAD je vytvoření návrhu aplikace sestaveného výlučně ze vzorů. To znamená, že se nevěnuje pouze analýze, ale i dalším aspektům vývoje, jako je například *návrh*. Používá tzv. *konstrukční vzory*. Jedná se o vzory, které zapouzdřují určitou oblast s daty a vystavují navenek pouze metody pro práci s daty nebo aplikační logikou. To znamená, že každá jednotka (vzor) obsahuje aplikační rozhraní (interface).

Metoda POAD navrhuje rozdělit vývoj do tří fází:

- *Analytická fáze*

  Dochází na základě analýzy požadavků k určení vhodných vzorů.

- *Návrh na vyšší úrovni*

Během návrhu na vyšší úrovni se vzájemně propojují instance vzorů, které byly vybrány v analytické fázi. Spojování vzorů je v knize určeno termínem „gluing", neboli lepení. Probíhá formalizovaným postupem s podporou propojovacích vzorů.

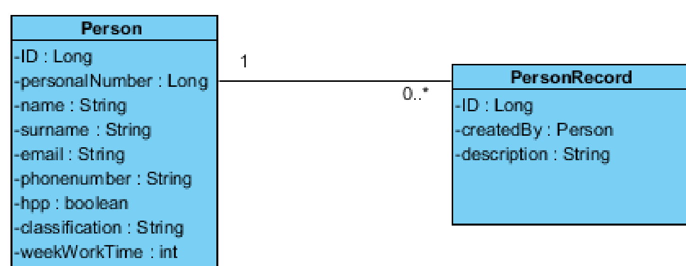
- *Zpřesnění návrhu*

  Zpřesnění návrhu doplňuje aplikační model z předchozích fází o detaily. Je možná také optimalizace vazeb, pokud je to potřeba.

V systému pro plánování směn byly použity jak systematická, tak nesystematická aplikace vzorů. I když je systém specifický a určitě jeho jádro nebude znovu použito pro výstavbu nějakého nového produktu, převládala systematická aplikace vzorů. Přesněji *metoda rozšíření modelu*. Většinou nelze při řešení situace poznat, jestli ji náhodou nepokrývá nějaký vzor, bez toho aby architekt znal všechny vzory nazpaměť. Lepší je vybrat seznam vzorů a hledat pro ně uplatnění v diagramu tříd. Nesystematickou aplikací byl použit jen vzor *Měření* (5.2), protože jsem znal jeho specifikaci a jakmile byl seznam požadavků hotov, situace ihned vybízela k použití vzoru. Zbytek vzorů byl aplikován již zmíněnou metodou *rozšíření modelu.* Konkrétně se jedná o vzory *Rozsah, Účet, Hierarchie organizace, Plán, Heterogenní seznam a Akce.*

5.2 Měření

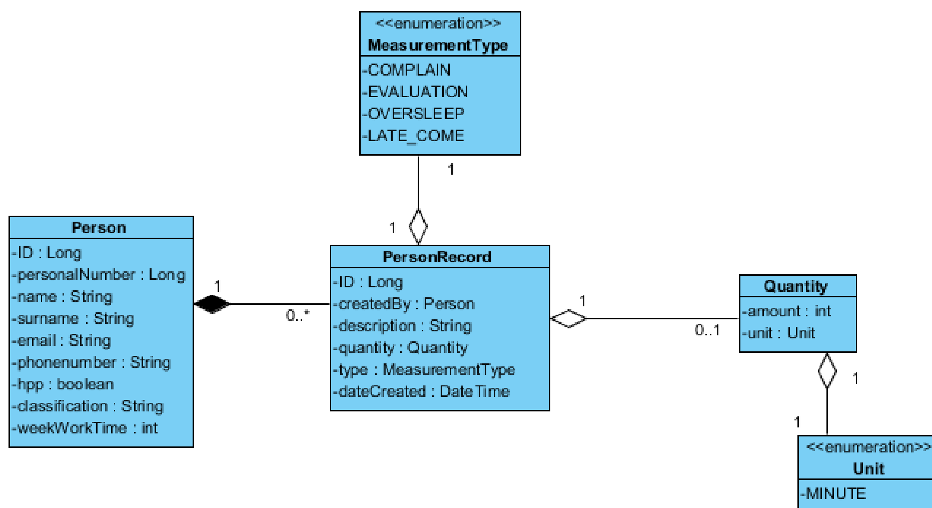
Jednotliví vedoucí si k stevardům přidávají poznámky o jejich aktivitě. Jde pouze o textové poznámky sloužící k pozdějšímu ohodnocení odvedené práce. Z toho důvodu byla pro zachycení poznámek vytvořena jednoduchá třída jak je vidět na obrázku 5.1.



Obrázek 5.1: Původní návrh záznamu informací o stevardech.

Po konzultacích s koordinátory se požadavky na systém rozšířily o vizi do budoucnosti. Konkrétně se jedná o funkcionalitu, která by měla generovat podklady pro mzdy, to znamená počty odjetých hodin na určitých linkách, počty hodin, ze kterých se vypočítá stravné apod. Také přibyla potřeba auditovat některé akce pro pozdější dohledání. Použití

vzoru *Měření* dodává systému dostatečné možnosti pro záznam různých typů údajů. Schéma rozšíření modelu vzorem *Měření* je znázorněno na obrázku 5.2.



Obrázek 5.2: Schéma nasazení vzoru *Měření*.

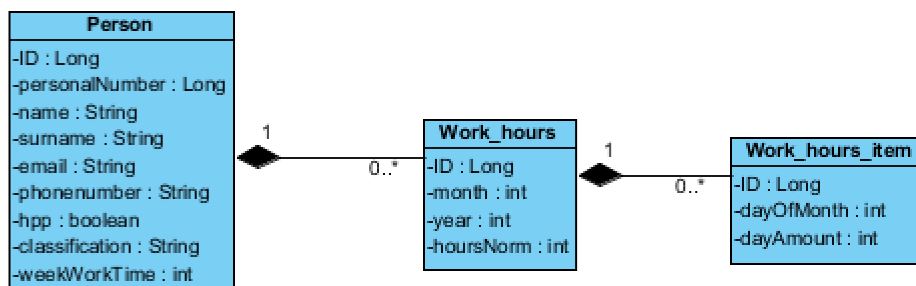
V současné době slouží pouze pro textové poznámky a záznam počtu minut, o které se člověk zpozdil při příchodu na směnu. Avšak přidáním dvou hodnot do výčtových typů *MeasurementType* a *Unit* lze velice snadno docílit nového druhu evidence. Například počtu stravenek, které zaměstnanci náleží. *MeasurementType* zde představuje tu část, která zůstává konstantní při každém měření. Je to klíč pro vyhledávání mezi záznamy. Už při prvním pohledu na výčtový typ získáváme představu o tom, k čemu celý vzor slouží. To je účelem pomyslného rozdělení diagramu na dvě části, jak je zmíněno v sekci 4.2.2.

Zhodnocení vzoru:

Použitím vzoru vzrostla flexibilita systému. Je snadné přidat, téměř jakoukoli veličinu, která má číselnou hodnotu. Při implementaci vzoru jsem uvítal zastoupení výčtových typů, protože přispívají k snadnější čitelnosti kódu. I když je to méně obvyklé, třída *Quantity* by mohla být doplněna o jiný než číselný údaj. Například pokud by bylo potřeba měřit čas v různých jednotkách. Pak je ale potřeba dávat větší pozor na numerické operace, které musí být přizpůsobeny více typům, které se budou nacházet v rozšířené třídě.

5.3 Účet

Při návrhu modelu byly pro záznam odpracovaných hodin použity dvě entity. Konkrétně *Work_hours* a *Work_hours_item*, kde *Work_hours* představuje měsíční záznam a *Work_hours_item* denní záznam. Každý stevard má týdenní pracovní normu. Každý měsíc je jinak dlouhý, proto se týdenní norma přepočítá na měsíční a je uložena v atributu *hoursNorm*.



Obrázek 5.3: Použití vzoru *Účet*. Třída *Person* znázorňuje pouze kontext.

Při hledání vhodných kandidátů pro aplikaci vzoru *Účet* se nejen strukturou entit, ale i sémantikou nabízely právě tyto dvě entity. Vzor do struktury, která je na obrázku, přidal atribut *monthAmount* (v kapitole 4.5.1 pojmenován jako *balance*). Ten reprezentuje aktuální počet odpracovaných hodin. Vyřešil se tak problém, který by časem nastal, s nutností neustálého přepočítávání denních záznamů. Při každém přidávání odpracovaných hodin by bylo potřeba přepočítat všechny záznamy v daném měsíci, aby se zjistilo, jestli není překročen měsíční limit. V aplikaci se zobrazuje aktuální počet odjetých hodin často, toto zobrazení by také mohlo trvat zbytečně dlouho, pokud by se pokaždé musela suma vypočítat.

Výhoda v tom mít záznam uložený v jediném atributu je však podmíněna nutností tento atribut mít stále aktuální. Existuje i druhá možnost. Hodnota v atributu nemusí být vždy aktuální. Vezměme situaci, kdy často potřebujeme záznam odpracovaných hodin za celý měsíc až do aktuálního dne. Pak by se vždy večer spouštěla akce, která by uplynulý den připočetla. Ovšem při plánování následujících dnů bychom potřebovali naopak vědět sumu za celý měsíc, aby nedošlo k překročení limitu. To se děje velice často, proto by musel být celý měsíc záznamů (*work_hours_item*) pro každého stevarda držen v paměti. Tento způsob je však pro aplikaci ShiftPlanner zbytečně složitý a nepřináší žádnou výhodu, navíc by zbytečně zvyšoval nároky na paměť.

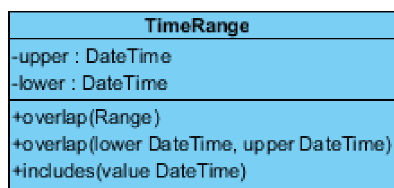
Zhodnocení vzoru:

Kromě výše zmíněné výhody, nasazení vzoru přineslo možnost použití navazujících vzorů. Ve stavu, kdy se jednalo jen o dvě entity, byla tato část diagramu „nepojmenovaná“. Díky aplikaci vzoru lze snadněji mapovat další vzory. Nasnadě by mohlo být použití vzoru *Transakce*, jehož princip je popsaný v kapitole 4.5.2. Modelová situace by mohla být taková, že při zadávání absence je potřeba odečíst z odpracovaných hodin určitou hodnotu, pokud absence pokrývá už naplánované jízdy. Tento záznam by mohlo být potřeba mít uložený. Avšak nevyužil by se naplno potenciál vzoru *Transakce*, protože jde o transakci pouze mezi jedním účtem. Pravděpodobně lepší řešení je přidat další hodnotu do výčtového typu *MeasurementType* a udělat nový záznam jak je popsáno v kapitole 5.2.

5.4 Rozsah

Je víc než jasné, že aplikace bude při svém běhu často porovnávat časové rozsahy. Autobusy mají příjezd a odjezd, dovolená má začátek a konec nebo časové možnosti stevardů mají také začátek a konec. To jsou všechno případy, kdy jsou mezi sebou potřeba porovnávat časové intervaly. Logické by bylo přidat vždy jeden atribut typu *Range* k entitám, které obsahují dva časové údaje. Vzhledem k tomu, že se často pracuje jen s jedním atributem (například odjezd autobusu) a také by se typ *Range* musel pokaždé do databáze mapovat s jiným názvem, rozhodl jsem se vzor použít jen pro zjednodušení operací s časovými intervaly.

Schéma vzoru představené v kapitole 4.3.1 je pro takové použití v aplikaci zbytečně složité. Nepředpokládá se, že by někdy bylo potřeba zjišťovat, jestli je spodní nebo horní hranice součástí intervalu. Také vzor nezávisí na žádné entitě, proto jsem vzor zjednodušil do podoby, která je vidět na obrázku. Deklaroval jsem metody pro určení, jestli se dva intervaly překrývají a pro zjištění jestli je hodnota uvnitř intervalu.

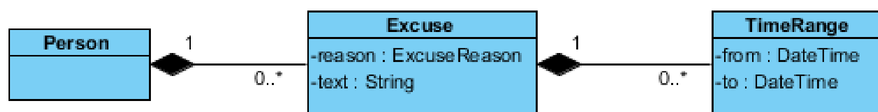


Obrázek 5.4: Vzor *Rozsah* jak je použit v aplikaci

Zhodnocení vzoru:

Přestože je vzor využit pouze z části, je velkým přínosem při implementaci aplikace. Zpřehledňuje kód, protože použití metod, které jsou součástí vzoru, je naprosto intuitivní. Po-

kud by byla v budoucnu potřeba u nějakých časových údajů evidovat skutečnost, zdali patří do nějakého intervalu nebo ne, pak je po přidání dvou atributů (*upperIncluded, lowerIncluded*) třída *Range* připravena k použití. Ještě přidám znázornění, jak by mohl být vzor použit, kdyby byl součástí některého z objektů mapovaných do databáze (obrázek 5.5).



Obrázek 5.5: Vzor *Rozsah* jako součást entity *Exuse*, kde tvoří rozsah omluvenky (od-do).

5.5 Hierarchie organizace

Jediná zřetelná hierarchie je patrná u uživatelských rolí. Při analýze požadavků jsem narazil na otázku, co všechno by mělo být přístupné jednotlivým uživatelům. V situaci, kdy byly uživatelské role definovány jen jako seznam, nabízelo se použití vzoru *Hierarchie organizace* (kapitola 4.1.2). Rozhovor s koordinátory kvůli upřesnění požadavků tuto aplikaci potvrdil. Než aby se pracně u každého elementu (v GUI) vyjmenovávaly všechny role, které k němu mají přístup, jednoduše se specifikuje nejnižší možná. Díky hierarchii rolí, která je uspořádána do N-árního je pak jednoduché určit, jestli uživatel s danou rolí má k objektu přístup, či nikoli.

Uvedu příklad. Stevard nesmí odesílat z aplikace e-maily. Proto tato sekce má nejnižší oprávnění pro přístup nastaveno na *koordinátor*. Z obrázku (5.8) je pak vidět, že k této sekci mají přístup pouze role „koordinátor (Coordinator)“ a „vedoucí (Supervisor)“.

Struktura vzoru popsána v kapitole 4.1.2 je založená na objektech a na omezeních určujících pozici v hierarchii. Takto navrhnutá struktura byla použita i v aplikaci ShiftPlanner. Kromě identifikace o jakou roli se jedná, nejsou u instancí objektů potřeba žádná další data. Také se s objekty relativně těžce pracovalo, kvůli potřebě stále zjišťovat o jaký typ objektu se jedná.

Proto jsem se rozhodl nahradit objektovou strukturu v tomto případě za elegantnější řešení (obrázek 5.7). Definoval jsem výčtový typ, který obsahuje konstruktor a díky tomu lze zároveň definovat role spolu s jejich umístěním v hierarchii. Přidáním metod pro vrácení rodiče a potomků jsem docílil stejného efektu jako u objektového přístupu se ziskem lepší přehlednosti kódu.

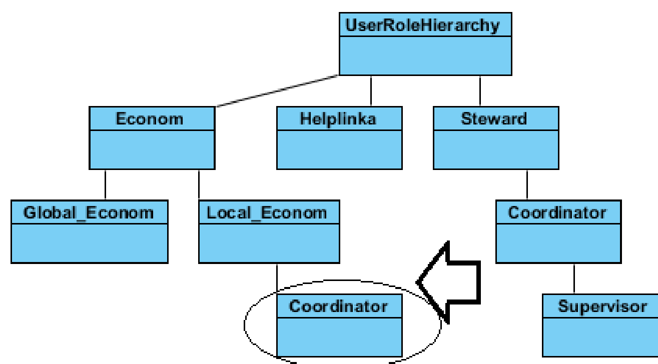
```java
public enum UserRoleHierarchy {

  Helplinka(null),
  Steward(null),
    Coordinator(Steward),
      Supervisor(Coordinator);

  private UserRoleHierarchy(UserRoleHierarchy parent){
    this.parent = parent;
    if(this.parent != null)
      this.parent.children.add(this);
  }
```

Obrázek 5.7: Výňatek kódu demonstrující použití vzoru Hierarchie organizace „neobjektovým“ způsobem.

Zhodnocení vzoru:

V průběhu analýzy se se vzorem dobře pracuje. Má jednoznačně definovanou strukturu, o které není potřeba příliš přemýšlet a je na první pohled jasné k čemu slouží. Problematičtější je implementační fáze. Je třeba pečlivě implementovat omezující podmínky, aby nedošlo k chybám. Jednoznačně však aplikace vzoru zvýšila rozšiřitelnost aplikace. Uživatelské role jsou zvlášť kritické místo, protože nový typ uživatele může přibýt téměř kdykoli.

U řádově větších řešení než jaké představuje ShiftPlanner by relativně lehce mohlo dojít k situaci, kde použití vzoru *Hierarchie organizace* narazí na problémy. Vezměme případ, kdy se hierarchie rolí rozroste do šířky a najednou bude potřeba, aby koordinátor měl zároveň práva z jiné větve stromu (obrázek 5.8). Lze to vyřešit přidáním další omezující podmínky. Avšak toto přináší velkou míru nepřehlednosti a použití vzoru se stává kontraproduktivní. Tuto variantu v ShiftPlanneru nepřipouštíme.

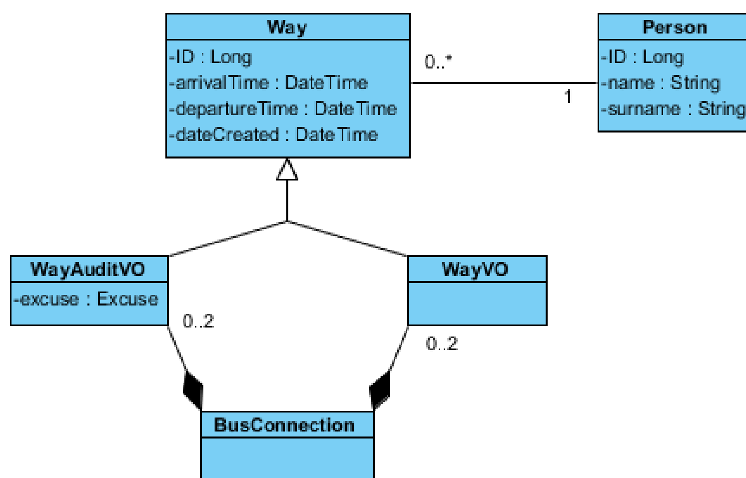


Obrázek 5.8: Větev nejvíce vlevo je záměrně přidaná jako modelová situace, kdy vzor přestává být dobře využitelný. Prostřední a pravá větev jsou pak znázorněním hierarchie, tak jak je použitá v ShiftPlanneru.

5.6 Akce

Tato kapitola vychází ze vzoru „Plánovaná a realizovaná akce“, který je popsaný v kapitole 4.6.1. Spíše než striktní rozdělení na naplánovanou a realizovanou akci byla v ShiftPlanneru použita drobná modifikace, která se svým charakterem odchyluje od konceptu „naplánovaná a realizovaná“. Z toho důvodu je tato kapitola pojmenována pouze jako *Akce*. Kandidáti pro nasazení vzoru jsou v diagramu hned dva.

První je na první pohled zřejmý. Jedná se o entitu *Way*. Struktura vlastností je shodná se strukturou, která je znázorněna na obrázku 4.6.1. Jen časová známka je zastoupena pouze atributem (*departureTime*), nikoli cizím klíčem, jak je popsáno v originálním schématu vzoru. Třídy *WayVO* a *WayAuditVO* jsou potomky *Way* a v nasazení vzoru reprezentují naplánovanou (*WayVO*) a realizovanou (*WayAuditVO*) akci. Modifikace vzoru spočívá ve vazbě mezi potomky *Way*. Není přímá, ale je reprezentována třídou *BusConnection*. Další změnou je již zmiňovaná odchylka v sémantice. V případě, že stevard má sestavený pracovní plán a je mu nahlášena absence, dojde k vytvoření instance *WayAuditVO*, která obsahuje odkaz na absenci. Vzhledem k absenci jde o změnu stavu.

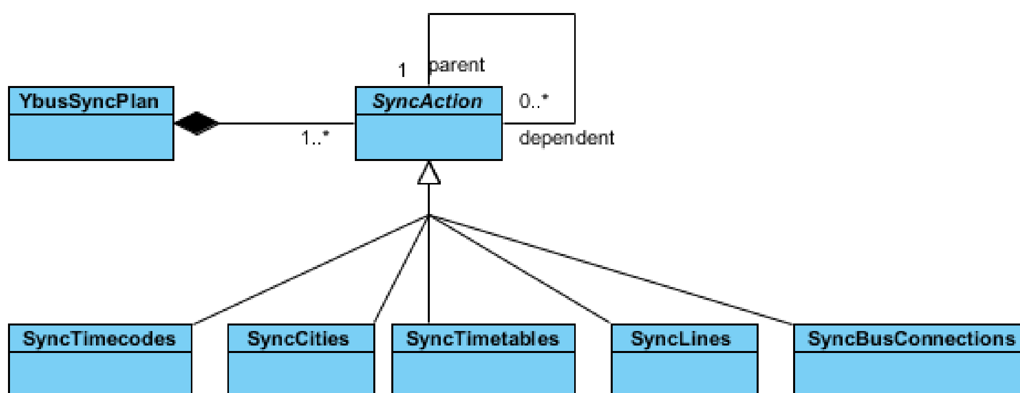


Obrázek 5.6: Znázornění vzoru *Plánovaná a realizovaná akce* v diagramu tříd.

Druhým kandidátem je abstraktní třída „*SyncAction*“ a její potomci. V tomto případě nejde až tak o akci ve smyslu vzoru, spíš se jedná o součást většího celku, který je vysvětlen v následujícím odstavci.

5.7 Plán a Heterogenní seznam

Synchronizační modul je pravděpodobně nejkritičtější oblast celé aplikace. To z důvodu, že se přenáší velké množství dat a pokud dojde k chybě v jejich formátu, je těžké chybu napravit. Použil jsem zde primárně dva vzory s tím, že další, třetí, vzor je pouze doplňující. Těma dvěma jsou *Plán* a *Heterogenní seznam*, doplňujícím je pak *Akce*. Třída *YbusSyncPlan* vytvoří závislosti mezi akcemi, které se mají zpracovat, také iniciuje start. Detailní popis průběhu synchronizace je uveden v následující kapitole. Vytvoření závislostí mezi akcemi je část, kterou do schématu přináší vzor *Plán*. *Heterogenní seznam* má v tomto případě význam kvůli abstraktní třídě (*SyncAction*), ze které dědí konkrétní synchronizační třídy. Plán nahlíží na seznam akcí jako na homogenní, což přináší výhodu v podobě jednoduchého přidávání dalších tříd.



Obrázek 5.6: Nasazení vzoru *Plán* a *Heterogenní seznam* do modelu.

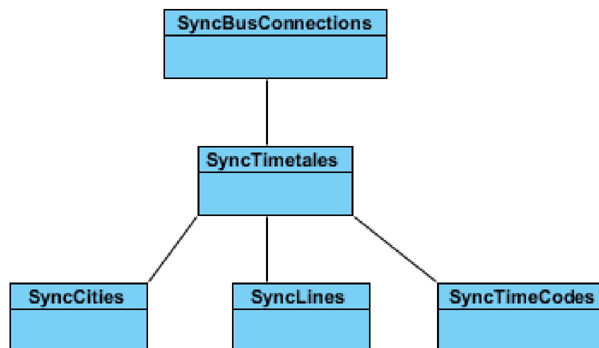
Nasazením vzorů v tomto případě získáváme relativně jednoduchou strukturu, která je přitom snadno rozšiřitelná o další synchronizační entity. Zároveň jde o robustní řešení, protože po definování závislostí je průběh plánu jednoznačný a neměnitelný.

5.7.1 Průběh synchronizace

Stručně uvedu popis jednotlivých tříd ze kterých je vystavěn strom synchronizace:

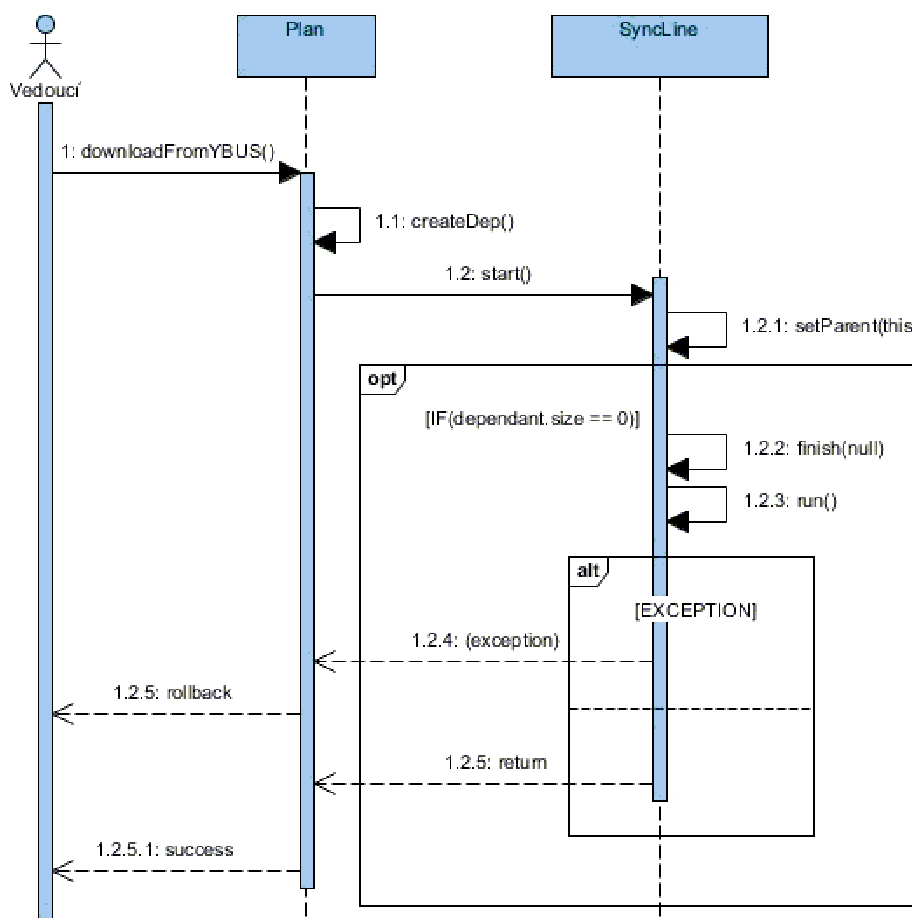
- SyncCities, SyncLines, SyncTimeCodes – Města, linky a časové značky jsou spolu s časem odjezdu a příjezdu základním stavebním kamenem každého jízdního řádu. Jsou nezávislé, tudíž tvoří listy stromu.
- SyncTimetables (jízdní řády) – Na jízdních řádech je postavena celá aplikace ShiftPlanner. Závisí na výše zmíněných záznamech, proto je ve stromě o úroveň výše.

- SyncBusConnections – Spoje jsou v YBUSu (externí systém) generovány z jízdních řádů. Spoj je údaj s konkrétním časem a datem odjezdu autobusu. Spoj vychází z jízdního řádu, proto je SyncBusConnections kořenem stromu.



Obrázek 5.7.2: Strom závislostí pro spuštění synchronizace je prozatím vytvářen staticky. Nevyskytla se potřeba, aby mohli uživatelé přímo zasahovat do synchronizace.

Po sestavení stromu je zavolána na kořeni metoda *start()*, která rekurzivně projde až k listům a na nich je zavolána metoda *run()*. Jakmile metoda run úspěšně skončí je předán běh programu do rodiče příslušného uzlu. Pokud má uzel více potomků, jako *SyncTimetable*, pak se čeká, až úspěšně doběhnou všechny metody potomků. Takto se běh programu dostane až ke kořenu stromu. Pokud dojde v kterékoli z úrovní k chybě, je zastaven průběh synchronizace a dojde k vrácení už provedených operací, tzv. „rollback“. Názorně je vše ukázáno na obrázku 5.7.3. V diagramu je zachycen pouze jeden list stromu. Celý diagram by byl velice nepřehledný, kvůli rekurzi.



Obrázek 5.7.3: Sekvenční diagram synchronizace linek. Jde o spuštění jen jedné akce, tudíž SyncLine je kořenem i listem stromu.

Zhodnocení vzorů:

Kombinace vzorů *Plán* a *Heterogenní seznam* vytváří seznam akcí, které mají být provedeny a pak je spustí v předem definovaném pořadí. Přináší do systému značnou abstrakci, díky abstraktnímu předkovi jednotlivých akcí. Je možné téměř neomezeně přidávat další potomky a vytvářet tak složitější strom. Samozřejmě je potřeba dodržovat určitou hierarchii. Je také možné jednoduše vytvářet různé druhy plánů. Jeden, který bude například spouštěn automaticky každý den a bude synchronizovat pouze některé třídy, zatímco jiný může aktualizovat celý strom vždy jednou týdně.

6. Použité technologie

Obsahem kapitoly je stručný přehled použitých technologií a architektury aplikace ShiftPlanner. Každá z technologií je pak samostatně popsána a významné části jsou znázorněny ukázkami kódu. Závěr kapitoly je věnován několika ukázkám z uživatelského rozhraní.

6.1 Java EE

Java Enterprise Edition (Java EE) je součást platformy Java. Je to standard, vyvíjený komunitou přes *Java Community Process*[1], primárně určený pro serverové a webové aplikace. Java EE dbá na bezpečnost, škálovatelnost a robustnost. K tomu napomáhá architektura, která je většinou rozdělena do následujících vrstev:

Datová vrstva – Poskytuje data aplikační vrstvě a zajišťuje jejich perzistenci. Zdrojem dat je většinou databáze. Použité technologie: PostgreSQL, Hibernate, JPA.

Aplikační vrstva – Zajišťuje vlastní funkcionalitu programu. Použité technologie: Spring, Dozer.

Prezentační vrstva – Je zodpovědná za zpřístupnění funkcionality aplikace uživateli. Použité technologie: Wicket, jQuery.

U každé vrstvy jsou vyjmenovány technologie, které jsou použity v ShiftPlanneru. Ještě před stručným rozborem použitých technologií bych uvedl několik hlavních vlastností Java EE:

- Java EE je postaven na platformě Java SE.
- Java EE umožňuje tvorbu webových aplikací s využitím technologií JavaServlets a Java Server Pages (*JSP*), případně Java Server Faces (*JSF*).
- Java EE poskytuje Java Persistence API pro zajištění persistence objektů prostřednictvím ORM mapování.
- Java EE poskytuje prostředky pro tvorbu komponent aplikační logiky prostřednictvím technologie Enterprise JavaBeans (*EJB*). [11]

6.2 Technologie datové vrstvy

V aplikaci jsou pro práci s daty použity tři technologie. Jsou to *Java Persistence Api (JPA), Hibernate a PotgreSQL.* Postupně jednotlivé technologie popíšu.

Java Persistence API je Java EE standard pro zajištění perzistence dat prostřednictvím objektově relačního mapování (ORM).

[1] Mechanizmus pro vyvíjení Java technologií. (http://www.jcp.org/en/home/index)

“Objekty reprezentující data jsou obyčejné POJO[1] objekty, které nepotřebují žádný kontejner a pracuje se s nimi jako s běžnými objekty. Tyto objekty jsou pomocí anotace doplněny o pomocné informace, které definují způsob jejich mapování na tabulky relační databáze”. [13, cit. 23.11.2013]

Hibernate je vývojový rámec umožňující objektově-relační mapování Java tříd do relační databáze. Princip je stejný jako u JPA, pouze anotace mohou být zaměněny za mapovací XML soubory. Na první pohled se zdá, že obě technologie jsou totožné, proto vysvětlím jejich použití odděleně a společně.

JPA je pouze specifikace[2]. Ať už jsou třídy anotovány pomocí JPA jakkoli, bez implementace se žádné mapování neprovede. Naproti tomu Hibernate lze použít samostatně, avšak omezujeme se tímto na jedinou technologii, která přináší jak specifikaci, tak implementaci.

Protože *Hibernate* obsahuje mimo jiné implementaci JPA anotací, nejvýhodnější je použití obou současně. Tedy třídy jsou anotovány anotacemi, které jsou za pomocí *Hibernate* zpracovány. Benefit, který tímto získáváme je ten, že můžeme téměř kdykoli zaměnit implementaci anotací, za jinou a použít jiný mapovací framework s tím, že stačí zaměnit pouze „importy“ v kódu u entit.

```
@Entity
@Table(name = "person_record")
public class PersonRecordVO extends AbstractVO {

  @ManyToOne(cascade = CascadeType.DETACH)
  @JoinColumn(name="createdby_id")
  private PersonVO createdBy;

  @Column(nullable = true)
  private String description;

. . .

public void createRecord(PersonRecordVO rec) {
    if (rec == null)
      throw new IllegalArgumentException("record");
    if (rec.getId() != null)
      throw new IllegalArgumentException("Already persisted");
    getSession().save(rec);
}
```

Obrázek 6.2: Použití JPA anotací.

[1] Plain Old Java Object (POJO) představují „jednoduché“ objekty. Obvykle obsahují pouze atributy, *get* a *set* metody a bezparametrický konstruktor.
[2] Standard JPA implementují například Hibernate, OpenJPA nebo Toplink.

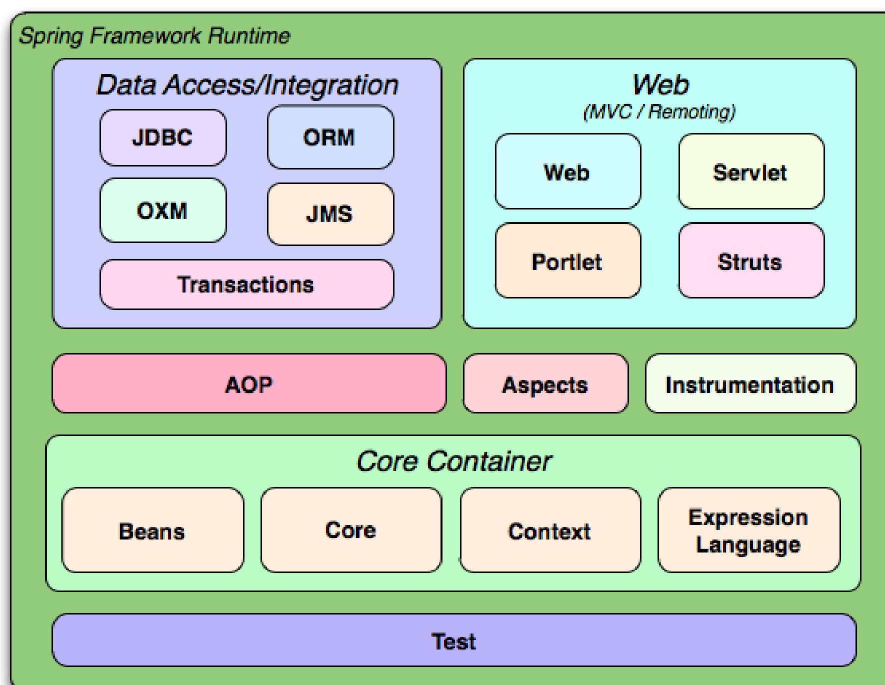
Ukázka demonstruje použití JPA anotací u třídy a jednoduché volání metody *save()*, která je už z *Hibernate* (obrázek 6.2).

PostgreSQL velice používaný, volně dostupný databázový systém. Obsahuje jak relační, tak objektově orientovaný přístup. Je primárně vyvíjen pro unixové systémy. [12] Hlavním důvodem proč ShiftPlanner běží právě na databázi PostgreSQL je ten, že je ve společnosti Student Agency k.s. používána jako primární databázový systém.

6.3 Spring

Spring Framework je open source aplikační rámec pro vývoj aplikací na platformě Java. První verze frameworku byla představena v roce 2002 Rodem Johnsonem. Postupně se stal velice oblíbeným jako alternativa k Enterprise JavaBean (EJB).
Spring framework nabízí alternativu k *Java EE,* aby vývojář mohl zajistit transakce, práci s daty, webové služby, posílání zpráv a další. Spring framework nabízí integrační vrstvu, díky které může aplikace pracovat v různých běhových prostředích (aplikační server, servletový kontejner, integrační test).

Spring nabízí množství modulů. Obsahuje podporu objektově- relačního mapování, JDBC, transakcí, webových aplikací založených na MVC, autorizace a autentizace nebo testování. [15]



Obrázek 6.3.1: Struktura modulů frameworku Spring. Převzato z webu. [17]

V ShiftPlanneru jsou z rámce Spring použity dva moduly. Prvním je „vkládání závislostí", anglicky „dependency injection", což je speciální případ návrhového vzoru Inversion of Control (IoC). Vkládání závislostí odbourává těsné vazby mezi vrstvami aplikace. V principu jde o to, že si třída nevytváří instance dalších tříd sama, ale jsou jí dány kontejnerem. Obvykle jsou závislosti mezi třídami popsány v xml souboru. Vkládané instance se nazývají *beans*.

```
XML:
<bean id="cityService" class="cz.sa.shiftPlanner.servicesImpl.CityServiceImpl">
	<property name="cityDAO" ref="cityDAO" />
</bean>
<bean id="cityDAO" class="cz.sa.shiftPlanner.daoImpl.CityDAOImpl"></bean>

Code:

@Service
public class CityServiceImpl implements CityService {

	private CityDAO cityDAO;

	public void setCityDAO(CityDAO cityDAO) {
		this.cityDAO = cityDAO;
	}
...
}
```

Obrázek 6.3.2: Použití IoC kontejneru. Pozn.: Setter je nutný, protože jej kontejner používá k vložení instance. Třída CityDaoImpl implementuje rozhraní CityDao a musí být anotována, aby s ní kontejner mohl pracovat.

Dalším použitým modulem je transakční modul. Třídy nebo metody v aplikaci jsou doplněny o anotaci *@Transactional.* Anotace je „parametrizovatelná", kde se určuje, jestli se vytvoří nová transakce nebo je vyžadováno, aby nějaká už existovala.

6.4 Wicket

Wicket je komponentový, webový aplikační rámec (angl. framework) založený na jazyce Java. V rámci Wicket je vytvořeno celé uživatelské rozhraní ShiftPlanneru, proto je rozebrán detailněji, než jiné použité technologie.

Na rozdíl od MVC[1] rámců je velice často v komponentových aplikačních rámcích striktně oddělena vykreslovací logika od HTML protokolu. To přináší nespornou výhodu v čitelnosti kódu. Druhou předností stavění stránek z komponent je jejich znovu

[1] Model-view-controller (MVC) je softwarová architektura, která rozděluje aplikaci do tří nezávislých komponent tak, že modifikace některé z nich má jen minimální vliv na ostatní.

použitelnost. Wicket také využívá v hojné míře dědičnost. Tím, že komponenty dědí hodně vlastností od rodičů, usnadňuje a zrychluje se vývoj aplikace. [16]

Mezi další podstatné vlastnosti rámce Wicket patří tyto:

- Možnost integrace s frameworky Hibernate, Spring nebo jQuery (od verze Wicket 6.0). Speciálně Spring má přímou podporu pro beany, které jsou kontejnerem vkládány přímo do stránek.
- Obsahuje množství připravených komponent. Například pro validaci uživatelských vstupů nebo vykreslování datových tabulek.
- Podporuje AJAX i klasické HTTP dotazy.
- Při psaní komerčních webů je důležitá možnost vytváření tzv. „clean URL“, neboli adres optimalizovaných pro vyhledávače.

Uživatelské rozhraní je poměrně rozsáhlé a do budoucna se plánuje jeho rozšíření. V ukázce níže uvádím jen malou část, na které demonstruji princip rámce Wicket.

Obrázek 6.4.1: Obrazovka po přihlášení uživatele.

Na obrázku 6.4.1 vysvětlím dědičnost. Tedy žlutá lišta nahoře je v kódu pojmenována jako *HomePage*, která dědí od třídy *WebPage* z balíku *org.apache.wicket.markup.html,* znázorňující webovou stránku. Menu na levém kraji je *MenuPage*, která dědí od *HomePage*. Díky tomu je možné vidět obě stránky naráz. Datová tabulka uprostřed je už komponenta, která však patří do stránky *LineView*, která dědí od *MenuPage.*

Téměř všechny prvky v rámci *Wicket* dědí od třídy *Component*, díky tomu je možné je libovolně skládat na stránku. To se dělá tak, že do základních HTML tagů přidáme atributy „wicket:id“, které odkazují na prvky v Java kódu. Například vypsání textu je ukázáno na obrázku 6.4.2.

```
Java:
add(new Label("message", "Seznam skupin linek"));
HTML:
<span wicket:id="message">Message goes here</span>
```

Obrázek 6.4.2: Ukázka použití rámce Wicket

Složitější „objekt“ na stránce pak může vypadat například, jak je ukázáno na obrázku 6.4.3.

BUS 144	...		-		-		-		-		...
			Brno	Praha	Praha	Brno	Brno	Praha	Praha	Brno	
			05 : 00	07 : 20	09 : 30	12 : 15	13 : 30	16 : 15	18 : 00	20 : 30	
			Eva		Eva		Eva		Eva		

Obrázek 6.4.3: Točka autobusu obsazená stevardem.

Celá „točka“ je tzv. Panel. Ten je pak opakovaně vložen na stránku. Jednotlivé cesty autobusu jsou tzv. fragmenty a políčko pro stevarda je opět fragmentem, který je doplněn o „klikatelné“ obrázky.

S vývojovým rámcem Wicket se mi pracuje velice dobře. Jedinou nevýhodou nebo slabším bodem bych viděl přenos dat z vygenerované stránky zpět do Java kódu. Většinou se lze obejít bez toho, ale při implementaci některých funkcí v jazyce JavaScript[1] je občas potřeba zpracovat nějaká data. Předání těchto dat z JavaScript kódu je relativně dobře realizovatelné formátem JSON[2]. Nebo je možné přidat data jako atribut k URL. Oba dva postupy však dělají kód velice nepřehledným, protože s daty se najednou pracuje na dvou místech za použití odlišných postupů.

6.5 Dozer

Dozer je knihovna sloužící k mapování JavaBean objektů. Rekurzivně kopíruje data z jednoho objektu do druhého. Typicky se jedná o různé typy objektů.

Má široké spektrum konfiguračních možností a umožňuje kopírovat téměř jakýkoli typ atributu. Počínaje jednoduchými atributy, dále kolekcemi, složenými typy, konče rekurzivním mapováním závislostí.

Použití je velice jednoduché. Je možné jej přidat do „pom.xml“ jako knihovnu pro Maven. Velikou výhodou Dozeru je, že mapuje atributy objektů podle jejich názvu, tedy při shodných názvech atributů nemusíme vůbec definovat mapovací soubor. Většinou se ale definici mapování některých tříd nevyhneme. Proto vytvoříme xml soubor, ve kterém specifikujeme, jak se třídy mají navzájem mapovat. Zde se právě projeví bohatost konfigurace, protože můžeme specifikovat, jestli se jedná o oboustranné mapování, jestli se má vytvářet nová instance nebo ne a spoustu dalších. Na obrázku 6.5.1 je vidět příklad jednoduchého obousměrného mapování tříd s jedním atributem.

[1] JavaScript je multiplatformní, objektově orientovaný jazyk používaný zejména při tvorbě www stránek.
[2] JavaScript Object Notation (JSON) je platformě nezávislý datový formát určený pro přenos dat.

```
<mappings>
  <mapping>
    <class-a>org.dozer.vo.TestObject</class-a>
    <class-b>org.dozervo.TestObjectPrime</class-b>
    <field>
      <a>one</a>
      <b>onePrime</b>
    </field>
  </mapping>
</mappings>
```

Obrázek 6.5.1: Ukázka konfiguračního souboru knihovny *Dozer.*

Pokud bychom například chtěli mapovat pouze třídu A na třídu B a naopak ne, uděláme jednoduchou úpravu jak je vidět na obrázku 6.5.2.

```
<mapping type="one-way">
...
```

Obrázek 6.5.2: Jednostranné mapování v konfiguračním souboru knihovny *Dozer*.

Nyní stačí mapovací soubor přidat do beany typu „org.dozer.spring.DozerBeanMapperFactoryBean“. Samotné použití je pak otázkou jednoho řádku kódu (obrázek 6.5.3).

```
private Mapper mapper;

public BusConnectionDTO copy(BusConnectionVO vo){
   return mapper.map(vo, BusConnectionDTO.class);
}
```

Obrázek 6.5.3: Použití knihovny *Dozer* v programu.

6.6 Maven

Apache Maven je nástroj pro správu, řízení a automatizaci sestavování aplikací (build). Maven sám nemá žádné uživatelské rozhraní a běží pouze na příkazové řádce. Jeho účelem je usnadnit práci vývojáři tím, že definuje jednotný proces sestavení. [14] Také definuje strukturu aplikace, protože jednotlivé typy souborů hledá v určitých balíčcích. Například spustitelné Java soubory by měly být v adresáři *src/main/java*.

Pro Maven je důležitý soubor „pom.xml“, ve kterém jsou uvedeny zásuvné moduly (pluginy), podle kterých Maven pozná, co má dělat. Také je zde seznam závislostí na

externí knihovny, které Maven dokáže stáhnout a tím velice ušetří práci vývojáři. Pro stažení externí knihovny z repositáře stačí uvést několik značek (obrázek 6.6.1).

```
<dependency>

<!-- upřesnění konkrétní knihovny -->

<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>

</dependency>
```

Obrázek 6.6.1: Ukázka z konfiguračního souboru *pom.xml.*

Při použití mavenu je sestavení programu otázkou jen jednoho příkazu (*mvn clean install*).

7. Závěr

Hlavním cílem práce bylo zhodnotit použití analytických návrhových vzorů při vývoji malého informačního systému. Tento cíl byl splněn a aplikace ShiftPlanner byla implementována v plném rozsahu specifikace požadavků.

První část práce byla věnována analýze. Obsahuje popis dosavadního nevyhovujícího stavu a diagram případů užití. Diagram obsahuje základní operace, které uživatelé provádějí v systému. Některé jsou detailněji popsány pomocí toku událostí.

Po analytické části následuje kapitola vymezující pojem *analytický návrhový vzor*. Je zasazen do kontextu a také je v kapitole uvedena hlavní literatura, ze které práce čerpá.

Čtvrtá kapitola se již věnuje analytickým vzorům. U každého vzoru je uvedena základní charakteristika a na jaké situace je primárně určen. Každý vzor je doplněn o malé schéma struktury vzoru.

Hlavní část práce, tedy pátá kapitola obsahuje popis nasazení vzorů do modelu. Nejdříve jsou uvedeny metodiky aplikace vzorů, pomocí kterých byly vzory aplikovány. Následuje seznam vzorů, které byly v ShiftPlanneru použity. Vždy je uveden popis situace, případně doplněn o schéma tříd. Vzor je zhodnocen z hlediska přínosu pro systém. U několika vzorů je také diskutováno rozšíření o další vzory. Některé vzory musely být pro použití v menším systému upraveny. Tyto úpravy jsou vždy zdůvodněny a jsou uvedeny alternativy. Příkladem takové modifikace je vzor *Range* (kapitola 5.4).

I když je tento informační systém doménově zaměřený na plánování, neznamená to, že se v něm nevyskytují jevy, které lze zachytit pomocí vzorů z jiných domén. Softwarová analýza posuzuje systém jako celek. Po zkušenosti, že mohu s výhodou využít vzory z oblastí mimo „Planning“, jsem dospěl k závěru, že použití vzorů jen striktně z

jedné domény je chybné. Přesto jsou vzory z této domény zastoupeny více než z kterékoli jiné a navíc se nacházejí v nejdůležitější části systému. Proto jsou rozebrány nejpodrobněji a jsou doplněny podpůrnými diagramy.

Závěrečná kapitola představuje použité technologie a architekturu systému. Hlavní použitá technologie pro datovou vrstvu je Hibernate, pro aplikační vrstvu Spring a uživatelské rozhraní je implementováno pomocí vývojového rámce Wicket.

Celkově bylo v práci představeno 10 vzorů, z toho bylo 7 úspěšně aplikováno. Myslím si, že analytické vzory nejvíce pomáhají v pochopení situace a i jen snaha o jejich použití odkrývá mnoho problémů, které nejsou na první pohled patrné. Samotné nasazení vzorů do modelu není jednoduchou záležitostí, ale když se to povede, předejde se mnoha implementačním chybám, protože vzory jsou autory dobře navrženy. Také vzory zvyšují přehlednost diagramů tříd.

Pro malé systémy jako je ShiftPlanner jsou analytické vzory stejným přínosem jako pro mohutné zdravotnické, finanční a jiné aplikace. Narazil jsem na jediné omezení v podobě objemných vzorů. V knize, ze které vychází tato práce, jsou vždy v závěru kapitol uvedeny komplexní vzory skládající se z mnoha tříd. Myslím si, že ty nejsou vhodné pro malé systémy, protože i kdyby se je povedlo implementovat, tak budou spíše komplikací navíc.

Aplikace ShiftPlanner je aktuálně nasazená na firemním testovacím serveru a běží zkušební provoz na lince s největším počtem spojů (Brno – Praha). V nejbližším období je plánovaný ostrý provoz na vnitrostátních linkách. Výhledově by se měly začlenit i zahraniční linky a systém by také měl sloužit pro generování podkladů pro mzdy. To znamená, že by měl vypočítat veškeré počty hodin, stravné a jiné.

Literatura

[1] FOWLER, Martin. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1996. ISBN 0-201-89542-0.

[2] GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. GANG OF FOUR. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2.

[3] BÜHNOVÁ, Barbora. *Analytické vzory* [online]. Brno [cit. 03.12.2003]. Dostupné z: <http://www.fi.muni.cz/~buhnova/PV167/Analyticke-vzory.pdf>. Výukový materiál. Masarykova univerzita.

[4] HUŇKA, František. *Aplikace návrhových vzorů* [online]. Ostrava, 2012 [cit. 03.12.2003]. Dostupné z: <http://www1.osu.cz/~hunka/vyuka/javaOOP/XOBJP.pdf>. Výukový materiál. Ostravská univerzita.

[5] YACOUB, Sherif a Hany AMMAR. *Pattern-oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley Professional, 2004. ISBN 02-017-7640-5.

[6] VONDRÁK, Ivo. *Úvod do softwarového inženýrství*. Ostrava, 2002 [cit. 03.12.2013]. Dostupné z: <http://vondrak.cs.vsb.cz/download/Uvod_do_softwaroveho_inzenyrstvi.pdf>. Výukový materiál. Vysoká škola báňská.

[7] Přispěvatelé Wikipedie, *Softwarové inženýrství* [online]. Wikipedie: otevřená encyklopedie, c2013. Datum poslední revize 03.12.2012 [cit. 10.11.2013]. Dostupné z: <http://cs.wikipedia.org/wiki/Softwarové_inženýrství>.

[8] KRAVAL, Ilja. *Analytické modelování informačních systémů pomocí UML v praxi*. 2010. ISBN 978-80-254-6986-6.

[9] OESTEREICH, Bernd. *Developing software with UML: object-oriented analysis and design in practice*. 2. ed. London: Addison-Wesley, 2002. ISBN 978-020-1756-036.

[10] WIEGERS, Karl. *Software requirements: object-oriented analysis and design in practice*. 3rd edition. London: Addison-Wesley, 2002, ISBN 978-073-5679-665.

[11] ADÁMEK, Petr & kol. FI WIKI : *Java EE* [Online]. Brno, 2013. Datum poslední revize 21.02.2013 [cit. 10.11.2013]. Dostupné z: <http://kore.fi.muni.cz/wiki/index.php/Java_EE>.

[12] *PostgreSQL* [online]. [cit. 23.11.2013]. Dostupné z: <http://www.postgresql.org>

[13] Přispěvatelé Wikipedie FI MUNI. FI WIKI : *Java persistence API* [Online]. Brno, 2013. Datum poslední revize 23.04.2013 [cit. 23.11.2013]. Dostupné z: <http://kore.fi.muni.cz/wiki/index.php/JPA>.

[14] *Maven* [online]. [cit. 23.11.2013]. Dostupné z: < http://maven.apache.org>

[15] WALLS, Craig. *Spring in action*. 3rd ed. Shelter Island: Manning, c2011, ISBN 19-351-8235-8.

[16] DASHORST, Martijn a Eelco HILLENIUS. *Wicket in action*. 3rd ed. Greenwich, CT: Manning, c2009, ISBN 19-323-9498-2.

[17] *JPA 2.0 and Spring 3.0 with Maven*. [online]. [cit. 03.12.2013]. Dostupné z: <http://paulszulc.wordpress.com/2010/01/09/jpa-2-0-and-spring-3-0-with-maven/>

Rejstřík

A

Analytické vzory, 7, 8
Analýza, 2, 8

B

Backend, 6

D

Diagram užití, 3

F

Frontend, 6

G

Generalizace, 10
GUI, 21

I

Interakční diagram, 15

J

JSON, 32

M

MVC, 30

N

Návrhové vzory, 6

P

POAD, 16
Požadavky, 2

S

Softwarové inženýrství, 1
Spring bean, 30

W

Wicket, 6, 30

Ž

Životní cyklus softwaru, 2

Přílohy

Příloha A - Funkční požadavky

1. Umožňuje sestavit časový harmonogram pro stevardy, kteří informují koordinátora, že mají pevnou pracovní dobu.
2. Podle časového harmonogramu systém generuje plán směn, který může být podle potřeby doplňován a upravován koordinátory.
3. U veškerého plánování i sestavování točení autobusů je potřeba rozlišovat mezi sudým a lichým týdnem. U zahraničí je pak nutné rozlišit čtyři po sobě jdoucí týdny, protože autobusy jezdí ve čtyřtýdenních cyklech.
4. Velké procento stevardů je zaměstnáno na dohodu o pracovní činnosti a mají flexibilní pracovní dobu. Proto systém musí umožňovat evidovat časové možnosti u jednotlivých stevardů. S tím také souvisí potřeba evidovat jakoukoli pracovní neschopnost včetně důvodu pracovní neschopnosti.
5. Jedná se o systém soužící pro každodenní práci. Jsou tedy spíše než na hezký design kladeny nároky na praktičnost a intuitivní práci.
6. Kvůli minimalizaci lidské chyby systém obsahuje integritní omezení pro dané situace:
    1. Stevard překročí svůj měsíční hodinový limit specifikovaný v jeho smlouvě.
    2. Stevardům se kryjí časy odjezdů u různých odjezdů autobusů.
    3. Stevard nemá dostatečný odpočinkový čas mezi jízdami.
    4. Hlášení koordinátorům, když 24 hodin před odjezdem autobusu není obsazen stevardem.
    5. Stevard nejel už 14 dní víkendovou službu.
    6. Změna v systému ze strany stevarda je formou upozornění (nikoli mailem) hlášena koordinátorovi.
7. Data v systému, jako jsou například informace o stevardech nebo jízdní řády autobusů, jsou synchronizována s databází společnosti Student Agency, k.s. pomocí interních webových služeb.

Každý uživatel, který má přístup k systému má svou roli, která jej omezuje v jeho právech.

Koordinátor:

1. Koordinátor může zobrazit seznamy linek rozdělené do kategorií, kde přehledně uvidí obsazenost autobusů (stevardy).
2. Koordinátor má možnost vylistovat si seznam stevardů a editovat absenci nebo časové možnosti.
3. Koordinátor sestavuje točení autobusů, časový harmonogram jízd pro stálé stevardy a celkový plán směn.

Stevard:

1. Stevard může nahlížet do seznamu svých jízd (minulých i budoucích) a případně zobrazit počet odpracovaných hodin za měsíc.
2. Stevard se může přihlašovat k jízdám, které mu vyhovují, ale maximálně určitý čas dopředu před odjezdem autobusu, pak se pro něj přihlašování uzavře. V nabídce bude mít stevard jen linky týkající se jeho výjezdového města, aby se co nejvíce předešlo chybám.

Helplinka:

1. Má přístup ke všem datům v aplikaci, kromě poznámek na osobní kartě stevarda.
2. Nemůže nic editovat, mazat ani spouštět žádné akce.

Příloha B - Textová specifikace případů užití

Nahlašování časových možností – Časové možnosti stevardů jsou určeny zejména pro brigádníky. Je to informace pro koordinátory, kdy mají nepravidelně pracující stevardi čas a tudíž, kdy je mohou obsazovat do směn.

Primární aktéři: Stevard
Sekundární aktéři: Koordinator, Vedoucí
Vstupní podmínky: Uživatel musí být přihlášený.
Tok událostí:

1. Na kartě stevarda uživatel klikne na záložku "časovky".
   1.2. Uživatel v kalendáři vybere dny kdy mají být "časovky" nahlášeny.
      1.2.1. Uživatel kline na tlačítko "vytvořit".
      1.2.2. Systém uloží údaje do databáze a aktualizuje GUI.
   1.2. Uživatel v seznamu již vytvořených časovek některou smaže nebo upraví.

Editace karty stevarda - Na kartě stevarda jsou souhrnně uvedeny všechny důležité informace, které je možné přidávat, odebírat nebo měnit.

Primární aktéři: Koordinátor, Vedoucí
Vstupní podmínky: Uživatel musí být přihlášený.
Tok událostí:
1. Uživatel vybere ze seznamu lidí jednoho a klikne na "editovat".
2. Uživatel vybere jednu ze záložek.
IF Poznámky
2.1. Uživatel vyplní formulář a uloží poznámku.
2.1. Uživatel smaže nebo upraví již vytvořenou poznámku.
IF Absence
2.1. Uživatel vyplní formulář a uloží absenci.
2.1.1. Systém absenci zařadí rovnou mezi schválené absence a v rozsahu absence vymaže stevardovi jízdy.
2.1. Uživatel smaže nebo upraví již vytvořenou absenci.
IF Časovky
2.1. **include** (Nahlašování časových možností)

Přihlašování se k jízdám - Jízdy jsou hlavním obsahem aplikace. Jedná se přehled směn všech stevardů, na který je navázaná téměř veškerá agenda.

Primární aktéři: Stevard
Vstupní podmínky: Uživatel musí být přihlášený.
Tok událostí:
1. V přehledu linek uživatel vybere jednu a klikne na "harmonogram jízd".
2. Uživatel si zvolí časové rozmezí pro výběr jízd.
3. Systém přehledně vylistuje jízdy seřazené do karet po dnech. Jednotlivé jízdy jsou poskládány do toček. Viz. *Editace točení autobusů.*
4. Uživatel se závazně přihlásí k jízdám.
5. Systém aktualizuje přehled.

Podání žádosti o absenci - Uživatel v roli "stevard" může pouze požádat o absenci. Proto se při vyplnění formuláře absence uloží ve stavu "požádáno". Schvalování absencí je v kompetenci koordinátorů nebo vedoucích.

Primární aktéři: Stevard
Vstupní podmínky: Uživatel musí být přihlášený.
Tok událostí:
1. Na kartě stevarda uživatel klikne na záložku "absence".
2. Uživatel vyplní formulář.
3. Uživatel kline na tlačítko "vložit".
4. Systém uloží údaje do databáze a aktualizuje GUI.

Prohlížení jízdních řádů - Jízdní řády jsou základním údajem, ze kterého se se všechny ostatní odvozují.

Primární aktéři: Stevard, Helplinka, Koordinátor, Vedoucí
Vstupní podmínky: Uživatel musí být přihlášený.

Tok událostí:

1. Uživatel si v seznamu linek vybere jednu a klikne na "zobrazit jízdní řád".
2. Systém zobrazí jízdní řád na dané lince.

Prohlížení jízdních řádů - Harmonogram jízd je přehled veškerých směn na určité lince. Obsahuje také seznam posilových spojů.

Primární aktéři: Stevard, Helplinka, Koordinátor, Vedoucí
Vstupní podmínky: Uživatel musí být přihlášený.
Tok událostí:

1. Uživatel si v seznamu linek vybere jednu a klikne na "harmonogram jízd".
2. Systém zobrazí dvě vstupní pole pro zadání časového rozmezí.
3. Uživatel vyplní data od-do a klikne na "další".
4. Systém zobrazí harmonogram jízd seřazený po dnech. Seznam je jen pro čtení.

Zobrazení karty stevarda - Karta obsahuje veškeré důležité informace o určitém stevardovi: jeho směny, absence, časové možnosti.

Primární aktéři: Helplinka, Koordinátor, Vedoucí
Vstupní podmínky: Uživatel musí být přihlášený.
Tok událostí:

1. Uživatel si v seznamu lidí vybere jednoho a klikne na "editovat".
2. Systém zobrazí osobní kartu, členěnou na záložky.

Editace plánů pro generování - Po sestavení "točení" autobusů je možné vytvářet tzv. trvalé plány směn. Stevardi pracující na hlavní pracovní poměr mají obvykle stálé směny. Ty se do plánů zadají a mohou se pak opakovaně používat pro generování směn na konkrétní časové období. Ušetří se tak práce koordinátorů.

Primární aktéři: Koordinátor, Vedoucí
Vstupní podmínky: Uživatel musí být přihlášený. Musí být zadáno "točení" autobusů.
Tok událostí:

1. Uživatel si v seznamu linek vybere jednu a klikne na "plány".
2. Systém zobrazí volbu pro výběr období, které chce uživatel editovat.
   A) podle týdnů - plány jsou členěny do 4 týdenních cyklů, v každém týdnu je možné zvolit konkrétní den.
   B) konkrétní časový úsek - plány je možné nastavit na konkrétní časové období. Je to potřeba například kvůli změnám, které se týkají jen několika dnů nebo týdnů.
3. Systém zobrazí odjezdy autobusů seřazené podle točení autobusů, času a výjezdního města přehledně po jednotlivých dnech.
4. Uživatel přidává a odebírá stevardy.
5. UC končí výběrem jiné sekce z menu nebo zmáčknutím na tlačítko "finish".

Vytvoření upozornění - Součástí menu aplikace je i položka "Upozornění". Obsahuje přehledné členění různých oznámení a upozornění. Jednotlivé záložky jsou: "nejbližší neobsazený spoj", "stevard", "generátor", "kdo je online". Tato upozornění vytváří systém při různých výjimkách. Neočekávané výjimky typu *IllegalArgumentException* nebo *IndexOutOfArray* se nepočítají mezi upozornění.

Primární aktéři: Systém

Tok událostí:

1. Při běhu programu nastala situace, která vyžaduje vytvoření upozornění.
2. Systém vytvoří Log výjimky a vytvoří v databázi nový záznam obsahující text problému a jeho zařazení.
3. PU končí uložením záznamu do databáze.

Editace posil - Posilové spoje jsou nedílnou součástí harmonogramu jízd. Existují dva typy posilových spojů. V jednom (žlutý typ) musí být stevard celou cestu. Ve druhém (bílý typ) stevard pouze odbaví spoj a ten pak jede bez palubního personálu.

Primární aktéři: Koordinátor, Vedoucí

Vstupní podmínky: Uživatel musí být přihlášený. Musí být synchronizovány spoje. Musí být

sestaveno "točení" autobusů.

Tok událostí:

1. PU začíná, když uživatel klikne na tlačítko "trasy autobusu" v levém menu aplikace.
2. Uživatel pokračuje kliknutím na "harmonogram jízd" u určité skupiny linek.
3. Uživatel si vybere konkrétní časový úsek, ve kterém chce pracovat se spoji.
4. Systém zobrazí spoje sdružené podle točení autobusů po jednotlivých dnech. Pod seznamem je zobrazena tabulka s posilovými spoji na daný den.
5. Uživatel vybere ze seznamu stevarda a přiřadí jej k posilovému spoji.

OR

5. Uživatel smaže stevarda z posilového spoje.
6. PU končí výběrem jiné sekce z menu nebo kliknutím na tlačítko "finish".

Poskytuje data ke stažení - Na určité adrese má externí systém WSDL, ze kterého si aplikace vygeneruje potřebné kódy pro přístup k externím datům

Primární aktéři: YBUS, Helios

Tok událostí:

1. Externí systém čeká na požadavek od ShiftPlanneru.

Kontrola obsazenosti spojů - Je velice důležité aby všechny spoje byly obsazeny stevardem. Občas se v množství spojů stane, že se na některý zapomene a je na systému aby dal včas vědět, že taková situace nastala.

Primární aktéři: Systém

Tok událostí:

1. Systém spustí akci na kontrolu obsazenosti spojů na následující den.

IF [kontrola bez výsledků]

2. PU končí s tím, že jsou všechny spoje obsazeny stevardy.

ELSE

2. Systém vytvoří upozornění a PU končí.

Mazání starých dat - Jednou za měsíc se automaticky spouštějí některé akce na čistění databáze. Toto se například týká tzv. "časovek". Ty by měly být smazány, jakmile jsou koordinátorem použity, občas je koordinátoři nemažou nebo na to zapomenou. Proto se systém automaticky stará o čištění.

Primární aktéři: Systém

Tok událostí:

1. Systém spustí akci na promazání starých záznamů.
2. PU končí, jakmile jsou data úspěšně smazána.

Import externích dat - Stahování dat z externích systémů. Helios poskytuje informace o stevardech. YBUS poskytuje veškerá data o spojích a jízdních řádech.

Primární aktéři: Čas, Vedoucí

Tok událostí:

1. Uživatel spustí akci na synchronizaci dat.
2. Synchronizace probíhá podle plánu, který je podrobně rozebrán v kapitole 5.7.1 Průběh synchronizace).
3. PU končí.

Alternativní tok:

V bodu 2 došlo k výjimce.

3. Spustí se "rollback" a je vytvořeno upozornění.
4. PU končí.

Příloha C - Elektronické přílohy

Shift-planner/src – Zdrojové kódy celé aplikace.

Shift-planner/diag – Projekt aplikace Visual Paradigm obsahující diagramy.

Shift-planner/text – Text práce ve formátu PDF.

Shift-planner/img – Entitně relační diagram, diagram tříd a diagram užití ve formě obrázků.