VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DATOVÁ VRSTVA WORKFLOW SYSTÉMU

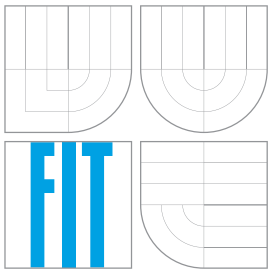
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

KRISTÍNA PROCHOCKÁ

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DATOVÁ VRSTVA WORKFLOW SYSTÉMU

WORKFLOW SYSTEM DATA LAYER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

KRISTÍNA PROCHOCKÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MILAN POSPÍŠIL

BRNO 2013

Abstrakt

Tato bakalářská práce se zabývá konkrétním návrhem datové vrstvy Workflow systému. Workflow systemy jsou užívané pro rozličné účely mnoha firmami. Výsledná práce navrhuje jednoduchou a flexibilni reprezentaci Workflow dat, implementuje – v jazyce PHP a částěčně v JavaScriptu – minimální jádro využívající tuto reprezentaci a poskytuje ukázkové procesy k demonstraci tohoto návrhu.

Abstract

This bachelor's thesis deals with a specific design for a data layer of a Workflow system. Workflow systems are used for various purposes by many companies. The resulting work proposes a simple and flexible Workflow data representation and implements – in the PHP language and partly JavaScript – a minimal kernel using this representation and provides sample processes showcasing the design.

Klíčová slova

Workflow, datová vrstva, PostgreSQL, task, proces, úkol, úloha

Keywords

Workflow, data layer, PostgreSQL, task, process

Citace

Kristína Prochocká: Datová vrstva Workflow systému, bakalářská práce, Brno, FIT VUT v Brně, 2013

Datová vrstva Workflow systému

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením Ing. Milana Pospíšila. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Kristína Prochocká
13. května 2013

Poděkování

Děkuji svému vedoucímu Ing. Milanu Pospíšilovi za odborné vedení při vypracování této práce a za cenné informace, které mi poskytl, jakož i za jeho velkou ochotu během průběžných konzultací.

Obsah

Kapitola 1

Úvod

Cílem této práce je navrhnout vhodnou datovou vrstvu. Prvním krokem byl tedy výběr databáze. O jejich výhodách a nevyhodách spolu s jejich rozdělením se uživatel seznámí v kapitole 2.

Workflow systém je počítačový systém, který řídí a definuje řadu úkolů v rámci firmy a produkuje konečný výsledek nebo výsledky. Umožňuje uživateli definovat různé pracovní postupy pro různé typy úloh a skládat je do procesů. Proces je soubor vzájemně propojených úkolů, které společně transformují vstupy na výstupy. Tyto úkoly mohou být prováděny lidmi, přírodou, nebo stroji s využitím zdrojů. V dnešní době jsou Workflow systémy velmi využívané, zejména ve vnitropodnikových a manažerských procesech. Tomuto a dalším výhodám takovýchto systémů se věnuje kapitola 3.

Systém by měl být spolehlivý, výkonný, neměl by mít problémy se změnou procesu za běhu či jeho přerušením. Takovéto a další požadavky na návrh Workflow systému jsou popsány v kapitole 4. Následný detailní popis návrhu nalezneme v kapitole 5.

Popis ukázkové implementace návrhu s ukázkovým procesem mergesort se uživatel dozví v kapitole 6.

Následující 7. kapitola se zabývá manažerskými procesy. Shrnutím celé této práce a nastínění jejího dalšího možného rozšíření naleznete v kapitole 8.

Kapitola 2

Základy databází a jejich rozdělení

Systém řízení báze dat (zkracováno na SŘBD či DBMS podle anglického výrazu database management system) je kolekce spolu souvisejících dat a množina programů a protokolů pro přístup k těmto datům. *Kolekce dat*, obvykle označovaná jako databáze, obsahuje zajímavé informace a metadata – informace o těchto informacích. SŘBD umožňuje sdílet data v databázi mezi více uživateli nebo aplikacemi.

Databáze jsou nezbytnou součástí dnešního podnikání. Skladují nejen určité druhy informací, které jsou společné pro většinu podniků, ale také informace, které jsou specifické pro dané podniky. V průběhu posledních čtyř dekád dvacátého století používání databází vzrostlo ve všech podnicích. V ranných začátcích, jen velmi málo uživatelů komunikovalo přímo s databázovými systémy, ale aniž si to uvědomili, byli ve styku s databázemi nepřímo – prostřednictvím tiskových sestav (výpisy z kreditní karty), nebo pomocí zástupců (bankovní úředníci a zástupci leteckých rezervací). Internetová revoluce v pozdních 90.letech minulého století prudce zvýšila přímý přístup uživatele k databázím. Organizace převáděli mnoho svých telefonních služeb do webového rozhraní uložených v databázi.

Databázové systémy jsou navrženy tak, aby spravovaly velké množství informací. Správa dat zahrnuje definující struktury pro ukládání informací a poskytování mechanismů pro manipulaci informací. Kromě toho databázový systém musí zajistit bezpečnost uložených dat proti výpadkům systému nebo neoprávněným pokusům o přístup. Pokud jsou data sdílena mezi několika uživateli, systém musí zabránit možným nezvyklým výsledkům.

V dnešní době patří společnost Oracle k největším softwarovým společnostem ve světě, ale je mnoho dalších známých firem věnujícím se databázím (Microsoft a IBM). Proto existuje mnoho různých typů databází, které mohou být rozděleny do různých kategorií podle počtu uživatelů, podle distribuovanosti a podle způsobu a intenzity použití [6].

2.1 Relační databáze

Relační databáze získaly své jméno, protože obsahují vztahy (tj. tabulky) a skládají se z kolekcí dat jednoduše uskupených ve formě tabulek. Popisuje ji tzv. relační model, jehož nedílnou součástí je normalizace. Ta zahrnuje soubor postupů, jejichž cílem je odstranění složených (neatomických) hodnot a redundanci (opakování) dat, což předchází anomáliím při práci s daty. *Relační model* je databázový model založený na predikátové logice prvního řádu, ve kterém jsou data reprezentována pomocí uspořádaných n-tic uskupených do relací. *Relace* se skládá z atributů a uspořádaných n-tic. Atributy popisují typy možných dat v každém sloupci relace a uspořádané n-tice jsou řádky této tabulky.

Relační model sestává z následujících tří částí [7]:

- **Struktura** – uniformní typ struktury jednotlivých dat zvaný relace
- **Manipulace** – množina operátorů, které transformují relace do jiných relací
- **Chování** – obecná pravidla integrity, která se starají o konzistenci každé databáze

2.1.1 Vlastnosti

Hodnoty jsou atomické, skalární. Každý sloupec v tabulce je jedna hodnota, nikoli množina hodnot. Tabulka musí splňovat první normální formu (1NF). Hodnoty ve sloupci jsou stejného typu. V relační terminologii toto znamená, že jsou všechny podmnožinou stejné domény. *Doména* je množina hodnot, kterých může sloupec nabývat. Jedinečnost je zajištěna primárními klíči. Toto zaručuje, že každý řádek může být identifikován pomocí hodnoty primárního klíče a není nepotřebný. Nezáleží na pořadí řádků ani sloupců v tabulce. Z databáze můžeme brát řádky či sloupce v libovolném pořadí. Každý sloupec je jednoznačně identifikován svým jménem. Protože nezáleží na pořadí sloupců, musíme se na sloupec odkazovat jménem, nikoli pozicí.

2.1.2 Výhody

- Nezávislost na struktuře – nezávisí na pořadí, nejsou žádné hierarchie
- Jednoduchost konceptu – umožňuje nám logický pohled na databázi, aniž bychom museli brát do úvahy, jak jsou data fyzicky uložena
- Jednoduchý návrh databáze, implementace, management a užití
- Deskriptivní dotazovací jazyk – SQL neříká databázi, co má dělat, ale co po ní chce [5]

2.1.3 Nevýhody

- Snadno nabádá k špatnému návrhu struktury databáze[5]
- Paradigma relačních databází nebylo navrženo pro distribuovanost

Mezi nejvýznamnější zástupce relačních databází patří MySQL od firmy Oracle.

2.2 Objektově-orientované databáze

Objektová databáze je databáze, ve které jsou data reprezentována ve formě objektů, jako je tomu v objektově-orientovaném programování. Základní myšlenka objektů je stejná jako u objektově-orientovaných jazyků. *Objekt* zapouzdřuje související data a metody pro manipulaci s těmito daty, má typ nazývající se třída. Objekty stejné třídy, mají stejné atributy a metody. Místo primárního klíče se k identifikaci objektu používá object ID (OID), které je unikátní pro tento objekt. OID je přiřazené objektu při jeho vzniku a nikdy se nemůže změnit.

2.2.1 Vlastnosti

Data jsou reprezentována jako množina objektů provázaných odkazy. Změny probíhají pomocí posílání zpráv konkrétním objektům. Objekty lépe vystihují prvky reálného světa. Zabezpečují identifikaci objektů vlastními systémovými prostředky. Ke spojování množin dochází v malé míře, jelikož objekty podporují polymorfismus.

2.2.2 Výhody

- Přístup ke komplexním datům může být snazší – join není většinou u této databáze potřeba (atributy objektu mohou být i komplexní typy nebo jiné objekty).
- Objektové vlastnosti – dědičnost objektů, polymorfismus, propojování objektů, zapouzdřenost
- Komplexní objekty nemusí být skládány z mnoha tabulek
- Možnost zamykání hierarchie objektu
- Snadnější aktualizace dat
- Součástí uložených objektů je také jejich chování

2.2.3 Nevýhody

- Ne všechny objektové databáze umí hledat podle komplikovaných dotazů, jako je tomu u relační databáze
- Nižší efektivita
- Méně dostupné nástroje pro čisté objektově-orientované databáze
- Složitý proces vývoje a návrhu modelu

Mezi zástupce objektově-orientovaných databází patří: Caché, GemStone, ITASCA, ObjectStore, Objectivity/DB.

2.3 Objektově-relační databáze

Objektově-relační databáze je SŘBD podobný relačním databázím ale s objektově-orientovaným modelem. Objekty, třídy i dědičnost jsou přímo podporovány ve schématech a v dotazovacím jazyku. Dále, stejně jako relační systémy, podporují rozšíření datového modelu uživatelskými typy a metodami. Spojují výhody relačních i objektových databází.

2.3.1 Vlastnosti

Objektově-relační databáze jsou objektově-orientované databáze postavené na relačním modelu, nebo databáze, které jsou postavené nad perzistentním programovacím jazykem, nebo objektově-relační mapování, které vytváří objektovou vrstvu nad relační databází. Systém objektově-relačního mapování umožňuje programátorům vytvořit aplikace používající objektový model, ačkoliv používají pro ukládání dat tradiční databázový systém, mají vysoké režijní nároky z konverze dat mezi objektovým a relačním modelem.

Objektově-relační databáze poskytují komplexní datové typy, silný dotazovací jazyk a konzistenci dat [6]. Tyto databáze vyžadují definování struktury dat před jejich použitím, ale neomezují ve způsobech použití těchto dat. Dokud je schéma navržené v normální formě, bez duplikace nebo ukládání spočitatelných hodnot, můžeme se dotazovat na cokoliv. Pokud jsou načteny správně moduly a správně nastavené indexy, výkonnost bude dobrá i pro objemy dat v řádu terabytů s malým nárokem na systémové zdroje. Pokud je bezpečnost dat prioritou, musí být zajištěna ACIDita transakce [4].

2.3.2 Výhody

- Objektově-relační databáze využívají výhod relačních databází:
  - 30letá praxe s relačními databázemi
  - Silný dotazovací jazyk
  - Dostupné nástroje na práci
- Vyhovují požadavkům pro objektový přístup
  - Volná rozšiřitelnost systému o uživatelsky definované datové typy
  - Možnost definovat funkce pracující s těmito typy

2.3.3 Nevýhody

- Nevýhodou objektově-relačních databází je předčasný návrh schématu
- Neefektivní zpracování komplexních datových struktur – jsou rozdělené do mnoha relačních tabulek

Mezi zástupce objektově-relačních databází patří PostgreSQL a DB2.

2.4 Dokumentové databáze

Dokumentové databáze jsou složeny z množiny samostatných dokumentů. To znamená, že všechna data pro daný dokument jsou uložena v tom samém dokumentu, v žádné tabulce, jak by tomu bylo v relačních databázích. V dokumentové databázi nejsou potřeba žádné tabulky, řádky, sloupce ani vztahy. Nemají žádná schémata, která by musela být dopředu definována. Pokud dokument potřebuje přidat nové pole, může toto pole přidat, aniž by to mělo vliv na jiné dokumenty. To také znamená, že dokumenty nemusí obsahovat prázdné hodnoty pro pole, pro která nemají hodnotu.

2.4.1 Vlastnosti

Nepřítomnost schémat je ideální k ukládání dynamických dat. Vzhledem k tomu, že data jsou ukládána v dokumentech, je možné získat všechna relevantní data v jednom dotazu. Dynamické entity, jako jsou uživatelsky přizpůsobitelné entity nebo entity s velkým množstvím nepovinných polí, mohou být uloženy bez problému se schématem. Nepotřebují funkci *view*, jako je tomu u relačních databází, a data mohou být uložena ve finální formě bez nutnosti dalšího počítání. Data v dokumentové databázi mohou být snadno rozdělena na více serverů, protože nejsou vzájemně závislá, pokud se nachází na jednom serveru, i tak se dají efektivně zpracovat.

2.4.2 Výhody

- Objekty mohou být ukládány jako dokumenty
- Dokumenty mohou být komplexní
- Dokumenty jsou na sobě nezávislé
- Otevřené formáty – data jsou ukládána v JSON nebo v XML
- Žádná schémata
- Vestavěné verzování

2.4.3 Nevýhody

- Není žádný standardní dotazovací jazyk
- Není mnoho nástrojů na práci s těmito databázemi
- Nejsou žádné standardy

Zástupci těchto databází jsou CouchDB, RavenDB, MongoDB a Terrastore.

2.5 Ostatní databáze

Grafové databáze využívají k reprezentaci a ukládání dat grafovou strukturu s uzly, hranami a vlastnostmi. Podle definice, grafová databáze je libovolná databáze, která poskytuje sousednost bez indexů. To znamená, že každý element obsahuje přímé ukazatele do sousedních elementů a není potřeba hledat v indexech. Obecné grafové databáze, které umí uložit libovolné grafy, jsou odlišné od specializovaných databází jako triplestores a síťové databáze.

Existuje ještě mnoho dalších databází, ale ty nejsou vhodné k práci s Workflow systémem.

Kapitola 3

Workflow a procesy

Workflow je obvykle označován za počítačové zjednodušení a automatizaci řízení procesů ve firmě. Skládá se z množiny aktivit, které jsou vykonávány kvůli dosažení konkrétního cíle. Cílem Workflow management systému je podpora přiřazování aktivit v organizaci tak, že práce je vykonána ve správný čas, správnou osobou a správným nástrojem. Zaměřuje se na strukturu procesu, nikoli na detaily konkrétních úloh. Podpora jednotlivých úloh je poskytována programy specifickými těmto úlohám. Workflow management propojuje osoby s konkrétními programy, aby mohly provést dané úkoly.

Workflow systém je informační systém založený na Workflow management systému (WFMS), který umožňuje vykonávat specifickou množinu procesů na základě specifikace těchto procesů. Workflow schéma (specifikace procesu) popisuje typ procesu, který může být interpretován jako šablona pro spuštění konkrétních Workflow instancí.[3]

Workflow systém můžeme rozdělit na čtyři typy:

1. **Administrativní Workflow** – Automatizuje rutinní procesy každodenní práce. Tyto procesy bývají velmi jednoduché, s pevnou strukturou. Příkladem takovéhoto typu workflow může být vystavení objednávky, sledování výdajů atd.

2. **Ad hoc Workflow** – Jedná se o náhodně objevující se procesy, které jsou většinou unikátní a nejsou předem specifikovány. Je nutné je definovat až v okamžiku jejich vzniku. Například odpověď na dotaz zákazníka, vyřízení nestandardní reklamace, apod.

3. **Kolaborativní Workflow** – Podporuje týmovou spolupráci, kde si účastníci vyměňují informace uložené v dokumentu, který je následně výsledkem jejich spolupráce. Jednotlivé úkoly se často opakují (iterativní zpracování) a jsou dynamické. Příkladem může být zpracování kupní smlouvy, tvorba dokumentace, aj.

4. **Produkční Workflow** – Převážně pracuje se statickými procesy, ale velmi komplikovanými. Definici definují specialisté a jednotlivé úkoly vykonávají řadoví pracovníci. Velmi často se používá u výrobních linek.

Cílem této práce bude navrhnout takový typ Workflow systému, který by podporoval všechny již zmíněné typy, tzn. navrhnutí rozšiřitelného jádra Workflow systému.

3.1 Prvky Workflow systému

3.1.1 Podnikový proces

Existuje mnoho různých druhů práce, jako je pečení chleba, stlaní postele, projektování domu nebo z nashromážděných výsledků průzkumu sestavit statistiku. Ve všech těchto příkladech můžeme nalézt jednu konkrétní „věc“, která je vyráběna nebo upravována: chléb, postel, dům, nebo statistika. Této „věci“ se říká případ. Případ nemusí být určitý objekt, ale může být také více abstraktní – řekněme jako soud nebo pojistná událost [1].

Každý případ má svůj začátek a konec a každý může být odlišný od všech ostatních případů. Případ zahrnuje hlavně plnění procesu. Proces se skládá z několika úkolů, které musí být provedeny, a množiny podmínek, které určí pořadí úkolů. Úkol je logická jednotka práce vykonávaná celá jedním prostředkem (obecné jméno pro osobu, stroj nebo skupinu lidí či strojů, kteří mohou vykonávat specifické úkoly) [3].

3.1.2 Definice a instance procesu

Reprezentuje podnikový proces ve formátu, který umožňuje automatické zpracování, jako modelování nebo spuštění pod Workflow management systémem. Definice procesu se skládá ze sítě úkolů a jejich vztahů, kritérií pro spuštění a ukončení procesu a informací o jednotlivých úkolech. Například o jejich účastnících, o souvisejících IT aplikacích, datech atd [3]. Způsob provedení určitých případů – to znamená, které úlohy je třeba provést – je popsán v příslušném procesu. Ukazuje také pořadí, ve kterém by se měly provést.

Instance procesu je základní jednotka vykonávání a běží samostatně, nezávisle na okolí. Pro každou definici procesu může být více nezávislých instancí, které mezi sebou nekomunikují.

3.1.3 Task

Task je atomická úloha vykonávaná za běhu procesu a logická jednotka práce. Je nedělitelná a proto se provede vždy celá. Skládá se z definice a její instance. Definice obsahuje informace o jejím propojení s ostatními tasky (například následující a předchozí task). V případě, že více tasků zahrnuje informaci o tom, zda se jedná o AND nebo OR směrování. Pokud se jedná o OR směrování, které si můžeme představit jako podmínku IF, pak se na základě podmínky definované v tasku směruje do jedné z větví. Směrování AND pokračuje ve všech úlohách zároveň.

Instance tasku je konkrétní výskyt úlohy běžícího procesu, tedy drží odkaz na task. Obsahuje lokální data, pokud tento task nějaká má. Instance může obsahovat i předdefinované chování platné pouze pro tuto instanci – více v sekci 4.2.2.

3.1.4 Subnet

Každý task může obsahovat další vrstvu podřízených úkolů – svoji podsíť. Tato podsíť slouží k implementaci úloh, které se skládají z více, již definovaných úloh. Používáme ji tedy stejně jako vnořené funkce v programovacích jazycích. Subnet může obsahovat vlastní definice použité jen pro úkoly v této podsíti. Informace, zda obsahuje task subnet, uchovává jeho instance.

3.1.5 Workflow model

Workflow model (nebo Workflow schéma) specifikuje všechny aspekty postupu, které jsou relevantní pro jeho provedení. Každé Workflow schéma definuje Workflow typ. Pro každý tento typ může být vytvořeno mnoho instancí (provedení workflow). Workflow schéma obsahuje informace o úkolech, které mají být provedeny, vztahy mezi úkoly a podmínkami.

Workflow modelování obvykle začíná s modelem podkladového obchodního procesu, který je pak upraven a vyladěn tak, aby odpovídal specifickým požadavkům Workflow managementu. Obvykle však není vhodné překládat procesní model přímo do Workflow modelu. Obecně platí, že je účinnější přeorganizovat celý proces s cílem plně využít potenciál procesní automatizace a počítačové podpory.

3.2 Výhody Workflow systému

Workflow má řadu důležitých výhod:

- Umožňuje nám dosáhnout jednotné správy funkčnosti a izolovat ji od zbytku systému (tato funkce byla rozšířena pro celý systém). Takto je možné využívat stejné funkce ve více úlohách.
- Aplikace již nevyžadují žádné funkce správy, a proto jsou jednodušší a zcela nezávisl0 na prostředí nebo místě v procesu. Díky tomu je možné změnit uspořádání podnikové procesy v pozdější fázi.
- Řídící vrstva umožňuje integrovat rozsáhlé aplikace. Tímto způsobem, je dokonce možné integrovat nové aplikace se staršími systémy.
- Podnikový proces, na úrovni řízení, je identifikovatelný a jeho stav, do kterého se dostane, je snadné zjistit. Tento proces je tedy více sledovatelný. Vzhledem k tomu, že je jasné, které úkoly mají být provedeny, je snadné určit, který by měl v konkrétním případě následovat. Proces je ovladatelnější a s pokrokem se dá snadněji průběžně kontrolovat.

Dále se podíváme na výhody z hlediska programátorského a vývojářského či manažerského.

3.2.1 Programátorský a vývojářský pohled

- **Workflow jako framework** – Workflow tvoří framework pro tvorbu obchodních procesů. Zahrnuje základní funkčnost, která by se jinak musela psát ručně.
- **Rapid application development** – Workflow je deklarativní, poskytuje nástroje pro rychlou tvorbu formulářů a procesů, neomezuje uživatele ve vytváření vlastních formulářů.
- **Analýza a vývoj systému** – Struktura procesu není definovaná v kódu, nýbrž deklarativně. Je ji možno vizualizovat pomocí různých nástrojů tak, aby ji pochopil i neprogramátor (např. formou Petriho sítě). Zároveň se může použít jako dokumentace, protože systém eviduje i změny procesů a verze.
- **Využití v nepodnikatelské oblasti** – Workflow má využití u složitých výpočetních systémů – například u úloh umělé inteligence a dataminingu. Hlavní logika běžícího

kódu je řízena Workflow, běh může být pozastaven, pozměněn dle potřeby a zase obnoven. Workflow se pochopitelně nepoužije na celou logiku kódu, ale může volat atomické výpočty asynchronně a sbírat výsledky.

3.2.2 Manažerský pohled

- **Formalizovaný podnikový proces** – Díky tomu, že je podnikový proces formalizován, je udržován ve společnosti větší pořádek. Snižuje se riziko opomenutí některého kroku a v případě odchodu klíčových zaměstnanců ze společnosti jsou jejich znalosti stále k dispozici. Další důležitá vlastnost je, že podnik má své definice centralizované, což je výhodné.
- **Zvyšuje efektivitu práce** – Pokud je Workflow správně navrženo a používáno, zvyšuje efektivitu práce. V administrativě se více používají elektronické dokumenty místo papírových a některé operace mohou probíhat paralelně.
- **Analýza** – Workflow produkuje historická data, která můžeme analyzovat a najít v nich prostor pro zlepšení.
- **Řízení** – Je-li proces dostatečně opakovatelný a predikovatelný (například výroba), můžeme ho pomocí těchto systémů i řídit – procesy mohou být plánovány s ohledem na zdroje a materiál. Na základě historických dat můžeme například kontrolovat některá rozhodnutí, pokud neodpovídají, je potřeba provést audit.

Kapitola 4

Požadavky na Workflow systém

4.1 Datová vrstva Workflow systému

První úlohou bylo nalezení vhodné databáze pro Workflow systém, která by ukládala data ve formátu JSON nebo XML a splňovala požadavky na vhodnost modelování, výkonnost a spolehlivost. Níže budou popsány přesněji jednotlivé požadavky.

4.1.1 Ukládání dat

Při ukládání je třeba uložit složitá strukturovaná data reprezentující definice a průběžné stavy procesů a tasků. Součástí těchto definic může být i kód ve zvoleném programovacím jazyce. Požadovaný formát ukládání dat je JSON nebo XML, rozhodnutí o výběru bylo na mně a bude popsán v kapitole 5.

4.1.2 Modelování

Uživatel (neprogramátor) musí být schopen nadefinovat vlastní proces z existujících komponent. Na toto se používají různé grafické nadstavby. Samotné modelování je však mimo rozsah této práce.

4.1.3 Výkonnost

Databáze musí být dostatečně škálovatelná, aby bylo možné spouštět procesy velkého podniku s mnoha přístupy do databáze. V takovém podniku může běžet i několik stovek procesů zároveň a tyto procesy se mohou skládat z velkého množství tasků.

4.1.4 Spolehlivost

Databáze musí podporovat ACID transakce a měla by být dostatečně zavedená a důvěryhodná. V okamžiku, kdy proces dosáhl nějakého stěžejního stavu, musí být všechna data uložena, a to i v případě výskytu hardwarové chyby, výpadku proudu… Zároveň při paralelní interakci mezi více tasky v rámci procesu nesmí nastat případ, že by task načetl nekonzistentní data, tzn. data uprostřed ukládání.

4.2 Jádro Workflow systému

Workflow systém musí být plnohodnotný programovací nástroj, umožňující znovupoužití předem definovaných úkolů. Měl by být lehce rozšiřitelný pomocí definic jednotlivých úkolů. Zároveň musí splňovat další požadavky popsané níže.

4.2.1 Přerušení za běhu a jeho pokračování

Protože je Workflow uloženo ve formě dat a ne jako programový kód (až na atomické jednotky vykonávání, které jsou psané běžným kódem), může být Workflow pozastaveno, uloženo a poté znova spuštěno. Stejně tak může být obnoveno v případě neočekávaného ukončení.

4.2.2 Změna procesu za běhu

Díky tomu, že je Workflow definováno mimo kód, je možné měnit provádění za běhu. To může být užitečné pro adaptabilní Workflow. Můžeme například zastavit systém, změnit mu logiku a poté zase spustit. Zároveň si jednotlivé úkoly mohou vytvářet za běhu nové podúkoly.

4.2.3 Monitorování

Manažer musí být schopen monitorovat běžící procesy a jejich stav. Například kdo zrovna pracuje na jakém úkolu. Nástroje na monitorování nejsou součástí této práce, návrh databáze je však zohledňuje.

Současně musí být k dispozici historická data již skončených procesů, takže je možno monitorovat proces i zpětně a v případě výskytu problému jej jednoduše nalézt. Tato data přitom také slouží k analýze historických trendů a statistik.

4.2.4 Verzování

Chceme zajistit, aby dříve spuštěné procesy mohly dokončit svůj běh s pomocí staré verze definice, a to i když byly přerušeny. Nově spuštěné procesy již poběží pod novou verzí definice. Každá instance si při spuštění uloží odkaz na správnou verzi definice. Pro jednoduchost verzujeme definice pomocí data.

4.2.5 Paralelismus

Workflow procesy mohou být déletrvající, musíme zamezit tomu, aby jeden proces blokoval jiné procesy aniž by tyto byly přímo závislé na jeho výsledku. Příkladem je proces, který čeká na vstup od uživatele. Sdílení dat musí probíhat pouze přes databázovou vrstvu, která musí zajistit konzistenci těchto dat.

Kapitola 5

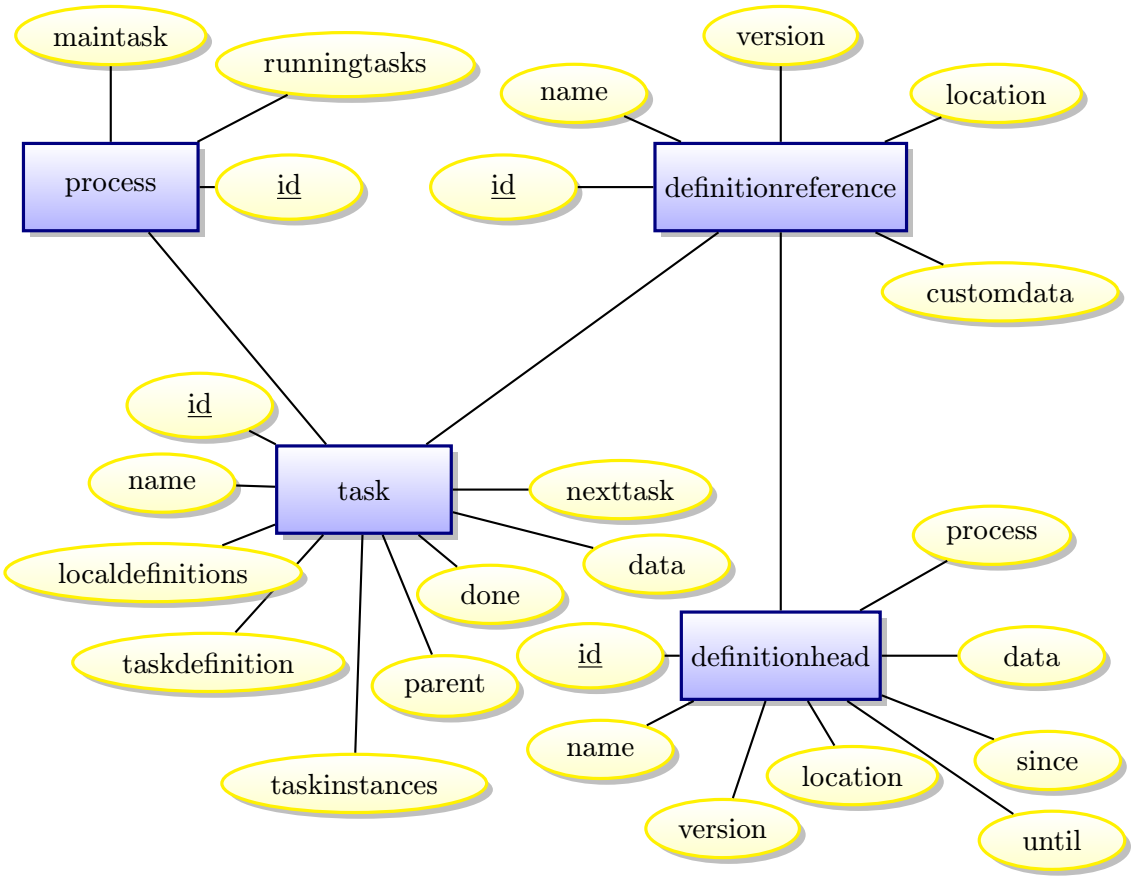
Návrh Workflow systému

Zaměřím se na jeden konkrétní návrh datové části Workflow systému a jádra postaveného nad touto datovou částí. Návrh zohledňuje popsané požadavky v kapitole 4 a zůstává obecný pro všechny typy Workflow. Tato práce je míněna jako jádro systému , které může být dále rozšířeno podle potřeby, nikoli jako kompletní Workflow systém.

5.1 Datová vrstva

Stávající návrh Workflow systému je velmi flexibilní a tedy v něm můžeme použit téměř jakoukoliv spolehlivou databázi, která umožňuje ukládání většího množství nestrukturovaných dat. Po shrnutí výhod a nevýhod zmíněných databází z kapitoly 2, jsem došla k následujícím závěrům:

- Použití standardního návrhu *relační databáze* nebylo vhodné z důvodu příliš komplikovaných vztahů mezi tabulkami a vysokého počtu tabulek. Je potřeba reprezentovat velké množství různých složitých entit, ale hledání probíhá jen podle primárních klíčů nebo jednoduchých položek.
- *Objektově-orientované databáze* nemají dostatečnou podporu ani nejsou tolik používané a neposkytují žádné zásadní výhody, které by tento nedostatek převážily. Jejich problémem je, že nelze jednoduše reprezentovat objekty různých typů v jedné kolekci.
- *Objektově-relační databáze* neposkytují dostatečnou flexibilitu na to, aby bylo možné uložit různé druhy definic do stejné tabulky s využitím jejich objektového modelu, avšak se velmi přibližují požadovanému standardu.
- *Dokumentové databáze* by byly vhodné, ale současné verze nejsou dostatečně vyspělé, aby zároveň poskytovaly spolehlivost a výkonnost – existující databáze poskytují pouze jednu možnost z těchto dvou anebo jsou příliš nové a tedy nedůvěryhodné z hlediska velkých firem. Současný návrh je velmi kompatibilní s datovým modelem těchto databází.
- *Ostatní databáze* jsou zpravidla stavěny pro specifické účely, které se neshodují s požadavky na reprezentaci dat ve Worflow systémech a zároveň nejsou ani výrazně používané.



Obrázek 5.1: Entity-relationship model pomocí Chenovy notace

Na obrázku 5.1 je ER model databáze. V databázi jsou potřeba pouze 4 tabulky, což znamená jednodušší dotazy. Veškeré selecty jsou jen pro hledání podle `definitionreference` nebo podle `id`. Definice vytvořené v kódu se automaticky ukládají do databáze. Je možné za běhu vytvářet nové definice, které částečně či úplně přetěžují existující definice (dědičnost). Jednotlivé sloupce v tabulkách odpovídají názvům proměnných v kódu (více v sekci 5.2).

Tabulka `definitionhead` obsahuje data jednotlivých definic. Sloupce mimo `data` a `process` se používají pro nalezení správné verze definice. Ve sloupci `data` je samotné tělo definice uloženo ve formátu JSON, který přímo odpovídá struktuře objektu v jádru systému. JSON navíc obsahuje položku `_class`, která označuje třídu daného objektu. Sloupec `process` je příznak, zda se jedná o definici procesu či nikoli. Tabulka `definitionreference` se odkazuje na `definitionhead` a může obsahovat vlastní definici ve sloupci `customdata`.

Instance tasků a procesů (jak běžící, tak již ukončené) jsou uloženy v tabulkách `process` a `task`. Sloupce těchto tabulek odpovídají proměnným ve třídách `TaskInstance` a `ProcessInstance`. Odkazy na ostatní entity (procesy, tasky, definice) jsou uloženy jako cizí klíče, které odkazují do příslušných tabulek. Systém se bude starat o automatické promítání změn v objektech do databáze, aby mohl kdykoliv skončit v konzistentním stavu.

Datová vrstva by měla umožnit jak vytvoření několika různých tasků podle stejné definice, tak nalezení již vytvořeného tasku podle jeho definice. Tato část návrhu je v konfliktu sama se sebou, neboť ne vždy je jasné, zda je daná reference míněna jako reference na nový objekt nebo na již existující.

5.2 Jádro systému

Jádro zajišťuje načítání a ukládání objektů do databáze a jejich postupné změny, spouštění/obnovování a také rozhraní pro vyvolání procesů. Stará se o převody dat mezi aplikací a databází, spojování definic, změny struktury za běhu i paralelní spouštění.

5.2.1 Definice objektu

Samotné definice jsou uloženy v třídách `ProcessDefinition`, `TaskDefinition`, `TaskBehavior`, `Subnet`, každá z nich má položku `Head`, která odpovídá záznamu v `DefinitionHead`. `ProcessDefinition` obsahuje odkaz na hlavní úlohu procesu a může obsahovat i lokální definice. `TaskDefinition` řeší úlohy na stejné úrovni jako task, ke které patří (vstupy a výstupy). Také obsahuje informaci, zda se jedná o první nebo poslední úlohu a v případě vícero vstupů či výstupů určuje `AND` nebo `OR` směrování. `TaskBehavior` obsahuje definici podsítě dané úlohy a nebo její kód. `Subnet` může obsahovat další lokální definice a také hlavní úlohu podprocesu.

Třída `DefinitionReference` se používá k nalezení správné definice. Může obsahovat `CustomDefinition`, která se vždy použije jako první, pokud nějaká existuje. Další definice se hledají podle kritérií vyplněných v této třídě – jméno, nepovinné verze, `location` a omezení data. V případě omezení na verzi jsou nalezeny jen definice se stejnou verzí, obdobně je tomu tak pro ostatní položky. Nalezené definice jsou poté spojeny do jedné, která obsahuje položky z nejspecifičtějšího místa, kde jsou definovány.

5.2.2 Spuštění procesů

Samotný proces obsahuje odkaz na hlavní task. Jednotlivé tasky jsou uspořádány ve stromu grafů – tasky na jedné úrovni jsou uspořádány do grafu – každá úloha může mít libovolné množství předků a následníků, každý task však může obsahovat `Subnet`, který představuje další úroveň nezávislou na ostatních tascích.

Workflow umožňuje spustit nový proces na základě jeho definice nebo obnovit již existující proces uložený v databázi. Uložený proces lze nalézt podle jeho definice nebo `id`. Každý proces má svůj `MainTask`, který řídí běh celého procesu. `MainTask` nemůže mít žádné potomky na stejné úrovni, protože by je neměl kdo spravovat. Spuštění `MainTasku` typicky vyvolá nějaký kód nebo jeho podtasky. Úlohy jsou uspořádány do stromové struktury, každá úroveň má nějakou vstupní a výstupní úlohu. Po skončení výstupního tasku je její nadřazený task označen za dokončený a pokračuje se v provádění jeho následníků.

Pro znovupuštění procesů je nutné znovu vytvořit instance procesu a hlavního tasku na základě dat v tabulkách (`process` a `task`) databáze. Poté dojde ke spuštění `MainTasku` a jednotlivé tasky jsou načítány či vytvářeny podle potřeby.

Možným rozšířením by bylo vyhledávání ve stromové struktuře pomocí relativních cest a též změna definice takto nalezených tasků. K implemetaci by bylo vhodné užití B+ stromů či jiných stromových algoritmů a změna ve třídě `TaskInstance`, která by musela mít unikátní jméno v rámci `Subnetu`, podle kterého by se hledalo. Avšak toto rozšíření nebudu implementovat ve své práci, místo toho naimplementuji jednodušší variantu, kdy každá úloha má odkaz na svého předka i pole potomků a takovýto přístup je možné řešit programově.

5.2.3 Směrování

Definice směrování pro Workflow určuje, jaké úkoly musí být provedeny a v jakém pořadí. Mezi základní směrování patří sekvenční, výběr, iterace a paralelního zpracování:

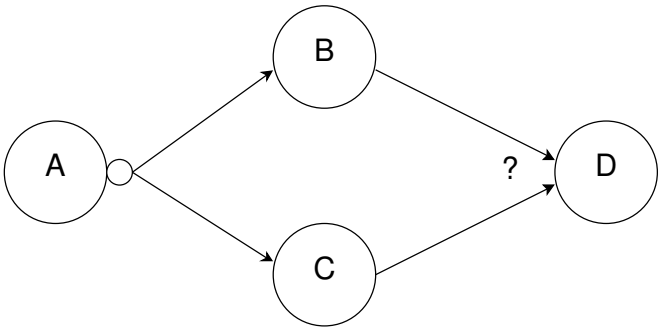
- **Sekvenční provádění úkolů** – Úkoly jsou prováděny jeden po druhém. Dva úkoly A a B musí být provedeny v pořadí, pokud například, do B přichází výstupní směrování A.
- **Výběr mezi úkoly** – Podmnožina daného souboru úkolů je zvolena tak, aby byly provedeny. Výběr je nutný, pokud například dvě činnosti potřebují přístup ke stejnému zdroji, avšak zdroj může sloužit pouze jedné činnosti. Vznikají dvě alternativní větve procesu, které jsou opět integrovány do jedné tzv. OR směrování.
- **Iterace úkolů** – Konkrétní úkol je proveden několikrát, a to buď na předem definovaný počet iterací nebo se opakuje, dokud není podmínka splněna.
- **Paralelní provádění úkolů** – Dva či více úkolů se provádějí nezávisle navzájem tzn. paralelně. To nemusí nutně znamenat, že úkoly jsou prováděny současně. Paralelní větvení procesů je vyjádřena tzv. AND směrováním.

V této práci budou využita výše zmíněná směrování. Vykonávání tasků na jedné úrovni začíná spuštěním startovacího tasku (`MainTask` podsítě nebo procesu). Po dokončení tasku je nutné se podívat na jeho výstupy a spustit některý z nich. Jak již bylo zmíněno dříve (5.2.1), task může mít dva druhy výstupního směrování a dva druhy vstupního – AND, OR. V případě výstupního OR směrování musí task během svého spuštění nastavit položku `NextTask`, která bude sloužit k identifikaci volby následující výstupní úlohy. V případě AND směrování jsou spuštěny všechny tasky, kvaziparalelně, aby vybraný task byl skutečně spuštěn, musí splňovat další podmínky:

- nesmí mít nevypršený timer
- musí být vybrán přes výstupní směrování nebo označen jako startovní
- všechny jeho vstupní závislosti musí být splněny

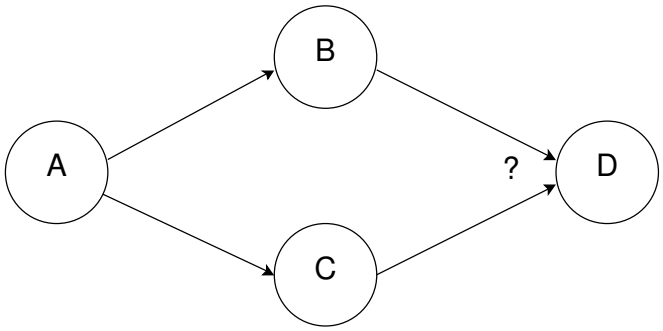
Řešení vstupních závislostí probíhá opět na základě AND/OR volby. AND znamená, že všechny vstupní tasky musí být dokončeny, OR, že alespoň jeden musí být dokončen.

Na obrázku 5.2 je zobrazena ukázka AND výstupního směrování. Po dokončení tasku A jsou souběžně prováděny tasky B a C. Mohou nastat dvě situace, což označuje otazník u vstupního směrování tasku D. Při AND vstupním směrování je po dokončení B i C spuštěn task D. Pro OR je task D také spuštěn, může však být spuštěn i dříve, než jeden z B nebo C.



Obrázek 5.2: Ukázka AND výstupního směrování

Obrázek 5.3 reprezentuje OR výstupní směrování. Task A musí vybrat mezi B nebo C, a ten který je vybrán se spustí po dokončení A. V případě OR vstupního směrování u tasku D se D spustí hned po dokončení vybraného tasku. Pro AND směrování se D buď nespustí vůbec (chybí mu závislost) nebo se spustí ta (pokud je nevybraná úloha spustitelná sama o sobě – například má atribut start) a potom D.



Obrázek 5.3: Ukázka OR výstupního směrování

5.2.4 Změny za běhu

Užitečnou vlastností je schopnost procesu či tasku modifikovat svoji definici za běhu nebo dynamicky vytvářet nové úlohy a předávat jim kontrolu. Existuje několik způsobů změny definice za běhu. První možnost předefinování se nachází v `Subnet` jako atribut `LocalDefinitions`. Standardně nebude obsahovat žádnou definici, ale pokud ano, budou použity ke změně globální definice tasků daného `Subnetu`. Toto předefinování může obsahovat prakticky cokoliv (definice směrování, chování tasku).

Je možné předefinovat chování tasků přímo v procesní instanci. V `TaskInstance` se nachází pole `LocalDefinitions`, které obsahují odkazy, kde je buď předefinována stávající definice nebo odkaz na jinou definici, která se opět hledá rekurzivně, tzn. nejdříve v procesní instanci a poté v globálních definicích.

Pro řešení chybových stavů je k dispozici ekvivalent `goto` schopný předat ovládání úplně jiné úloze, mimo definovaný běh.

5.2.5 Datová vrstva a polymorfismus

K zajištění snadného propojení databáze s Workflow systémem jsou klíčové entity v databázi přímo reprezentovány pomocí JSONu. Jádro zabezpečuje nalezení správných objektů v databázi, jejich automatickou instanciaci a perzistenci. Také řeší Workflow ekvivalent dědičnosti – lokální definice, které částečně upravují globální definice a také vlastní definice v kódu.

Hledání definice v aktuálním tasku probíhá stromově. Tzn. nejdříve hledáme definici v samotné referenci, poté v instanci vykonávaného tasku a v jeho předcích až ke kořeni. A teprve pokud nenalezneme ani v kořeni, podíváme se do globální definice.

Přitom platí, že hledáme všechny dostupné definice v daném pořadí a pro každou položku definice použijeme její nejspecifičtější variantu. Definice nemusí předefinovávat všechny položky. Ty, které nejsou zmíněné, budou zděděny z globálnější definice. Tedy můžeme předefinovat např. jen směrování, ale chování tasků zůstane stejné, takže bude platit implicitní chování. U směrování můžeme předefinovat také jen část – například vstupy, výstupy nebo jen příznaky `start`, `final`, apod.

5.2.6 Paralelismus

Veškeré akce budou atomické – není potřeba používat transakce, změna jedné položky v řádku je atomická sama o sobě. Paralelismus v tomto případě budeme realizovat pomocí vláken, díky kterým můžeme lokální data procesu udržovat v paměti bez ztráty konzistence – jednotlivé tasky by se prováděly paralelně ve více vláknech, hlavní smyčka programu může běžet v hlavním vlákně. Různé Workflow procesy mohou běžet v oddělených systémových procesech. Komunikaci mezi procesy může probíhat přístupem pouze pro čtení do dat instance procesu v databázi (kde jsou vždy aktuální), obousměrná komunikace může být implementována vzájemným čtením či přidáním konceptu posílání zpráv mezi procesy. Při dodržení těchto pravidel není třeba uzamykat tabulky v databázi, což vede k urychlení a odstranění potenciálních problémů s deadlocky.

V rámci implementace v jazyku PHP, který nepodporuje tvorbu vláken, jsem si tuto část návrhu zjednodušila tak, že v rámci jednoho procesu paralelismus není. Načítání z databáze by se muselo provádět při každém přístupu k datům znovu. Hlavní smyčka programu spouští vždy nějaký task připravený ke spuštění, ale blokuje běh dokud tento task neskončí (ať již dočasně – timer, nebo trvale). Popsaný způsob paralelního spuštění procesů však podporován je.

Timer je jednoduchý mechanizmus pozdějšího spuštění. Task může při svém spuštění rozhodnout, že chce být spuštěn až za n sekund. Workflow proces poté musí pokračovat v dalších spustitelných tascích nezávisle na tomto pozastaveném tasku. Task pak v danou dobu (nebo později) musí být spuštěn, aby zbytek procesu mohl pokračovat.

Kapitola 6

Implementace

Po přezkoumání výčtu databází z kapitoly 2 jsem se rozhodla modelovat datovou vrstvu nad PostgreSQL 9.2. Tato databáze podporuje objektově-relační model, avšak v této práci je využita jen její relační část spolu s dalšími funkcemi, zejména podporou typu JSON a polí. Všechny sloupce, podle kterých hledáme, jsou reprezentovány jako relační položky v tabulkách, ale všechna ostatní data jsou uložena ve formátu JSON, čímž se vyhneme proliferaci tabulek.

JSON byl vybrán místo XML hlavně proto, že byl přímo navržen k reprezentaci JavaScriptových objektů, a tedy je převod oběma směry triviální. Další výhodou je čitelnost a možnost zápisu bez použití specializovaných nástrojů. Jak již bylo zmíněno v podsekci 5.1, properties objektu jsou uloženy v JSONu přímo, jeho metody jsou definované v kódu a v JSONu je navíc jméno třídy, která je implementuje. Příklad takovéhoto objektu naleznete na obrázku 6.1. V nové verzi databáze PostgreSQL 9.3 bude možné pracovat s JSON daty přímo, bude možno vybírat i filtrovat podle JSON polí a to i včetně těch vnořených. Zatím jsem toto vyřešila přidáním všech sloupců, podle kterých se filtruje či vyhledává.

```
{
    "_class": "TaskDefinition",
    "inputs": [],
    "outputs": [],
    "inputs_route": null,
    "outputs_route": null,
    "final": true,
    "start": true,
    "TaskBehavior": {
        "_class": "DefinitionReference",
        "Name": "countedBehavior",
        "Version": null,
        "Location": null,
        "CustomDefinition": null
    }
}
```

Obrázek 6.1: Příklad definice JSON kódu

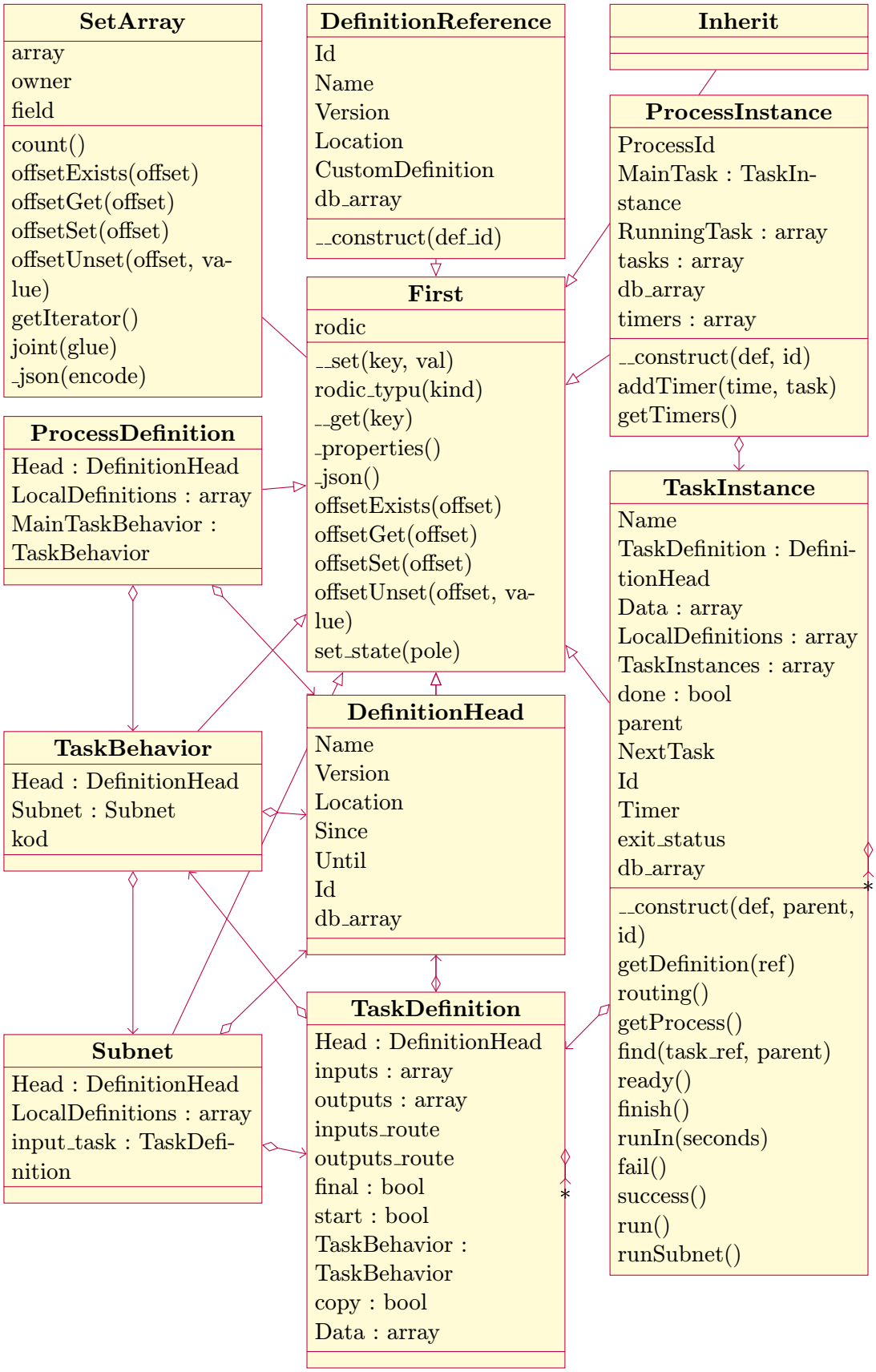
Při dotazech do databáze nedochází k zamykání tabulek, které by vedlo ke zpomalení systému a menší spolehlivosti této databáze. S databází komunikuje pouze jádro, jednotlivé procesy přistupují k databázi přes jádro.

Jádro Workflow systému bylo implementováno v PHP z důvodu jeho dynamičnosti a obeznámenosti autora s tímto jazykem. Propojení mezi PHP kódem a databází zprostředkovává vrstva Dibi, databázová vrstva s podporou bezpečného skládání dotazů a různých databází, včetně PostgreSQL. Perzistenci řeší třída `First` viz obrázek 6.2, ze které dědí všechny ostatní třídy, poskytuje obecný getter a setter, které se používají pro všechny properties a průběžně ukládají data do databáze. Properties, které jsou typu pole, musí být obaleny do objektu třídy `SetArray`, který vyvolá příslušný setter při změně pole. Kontruktory jednotlivých tříd umí objekt vytvořit v databázi, pokud už v ní není uložen.

Třída `First` mnohdy potřebuje informace o tom, jak je daný objekt uložen v databázi, k tomu slouží property `db_array` (definuje jméno a strukturu tabulky). Stará se také o převody do JSONu a udržování informací o stromové struktuře. Níže bude následovat popis nejdůležitějších části této implementace.

V rámci této práce byly vytvořeny příklady tasků na ukázku správně pracujícího jádra Workflow systému. Jedná se o výpočetní procesy, jednoduchou matematickou funkci a třídící algoritmus.

- **IF** – Definice tohoto tasku obsahuje větve `if` a `else`. Předdefinovanou podmínkou je výraz $5 > 4$ (je možné ji změnit v `CustomDefinition`) tato podmínka by měla být splněna a na výstup se vypíše *You can go to school*, což znamená správné ukončení tasku.
- **FOR** – Jedná se o ukázku for cyklu, který postupně vypíše předdefinované pole *[1,2,3,4,5]*.
- **WHILE PHP** – Ukázka provedení 500x while cyklu a následnému výpisu počtu provedení.
- **WHILE** – Na rozdíl od tasku WHILE PHP, tento WHILE je implementován ve Workflow systému pomocí tasků s podsítěmi. Změnou parametru je možno změnit podmínku i tělo tohoto cyklu.
- **EUKLID** – Euklidův algoritmus – nalezení jednoho čísla probíhá v jednom tasku
- **COUNTED** – operace s čísly, v definici je použit operátor pro sčítání, je však možno ho změnit na libovolný jiný operátor
- **CALL** – volání podpogramu v procesech
- **MERGESORT** – o detailnějším postupu tohoto algoritmu přes Workflow systém více v sekci 6.6
- **TIMERTASK** – tento task má ve své definici nastavený timer, který se při spuštění tohoto tasku zaktivuje, mezitím proběhnou následující tasky, a po vypršení nastaveného času se spustí



Obrázek 6.2: Diagram tříd

6.1 PHP kód v databázi

Pokud `TaskBehavior` obsahuje kód, je spuštěn při spuštění tasků a musí se starat o případné spuštění `Subnetu`. Pokud obsahuje pouze `Subnet`, je spuštěna jeho hlavní úloha a pokračuje se spouštěním jejích následníků. Kód je uložen jako řetězec v JSONu, který nepodporuje neescapované nové řádky uvnitř stringu, je tedy potřeba kód důsledně escapovat před uložením do databáze a je spouštěn v kontextu běžící `TaskInstance`, jeho `$this` tedy obsahuje odkaz na běžící úlohu.

Spouštění probíhá pomocí funkce `eval()`, která interpretuje libovolný řetězec jako kód. Pokud uživatelům v rámci přidávání definic poskytneme příliš velkou svobodu, může to vést k narušení bezpečnosti Workflow systému, tzn. uživatel může vložit libovolný kód. Řešením by bylo omezení kódu na volání schválených funkcí a metod, což však snižuje flexibilitu. Při vyšším počtu uživatelů a procesů může dojít k zahlcení systému a následnému pádu. Řešením tohoto problému je rozdělit server na databázi a několik nezávislých serverů, kde jádro používá stejnou databázi a spouští různé procesy.

6.2 Načítání

Načítání definic z databáze řeší funkce `getDefinition()`, která najde všechny platné definice odpovídající dané referenci a spojí je do jedné instance definice. Nevyplněná políčka v konkrétnějších definicích nebo ta, která jsou vyplněná instancí třídy `Inherit`, jsou nahrazena políčky z obecnějších definic. Načítání instancí procesů obstarává konstruktor `ProcessInstance`, instancí tasků metoda `TaskInstance::find()`, která hledá jen tasky v daném procesu. V obou případech jsou načtená JSON data předána funkci `getToObj()`, která se postará o instanciaci včetně rekurzivního zpracování vnořených objektů, ale nepřekládá vnořené `DefinitionReference`, ty jsou přeloženy až při pokusu o přístup k nim. Další popis načítání definic je níže v sekci 6.3.

6.3 Změny za běhu

Příkladem změny za běhu v mé práci je algoritmus mergesort, který má definici uloženou v databázi a každý fragment pole je zpracovaván ve zvláštním tasku, vytvořeném za běhu podle potřeby. Funkce `getToObj()` dostane JSON popis definice a vytvoří instanci příslušné definice. Toho se dá využít pro dynamické vytváření tasků a jejich zapojení do existující sítě. Z pohledu jádra Workflow systému se dynamicky vytvořené definice nijak neliší od definic již uložených v databázi, při vytvoření jsou do databáze uloženy a od té doby se s nimi pracuje stejně jako s načtenými definicemi. Zapojení nové instance do procesu probíhá pomocí nastavení `inputs` a `outputs` příslušné `TaskInstance`, aby se task spustil, musí být v `outputs` nějakého již připojeného tasku nebo jako `MainTask` existujícího `Subnetu`.

Primitivní formou změny za běhu je i funkce `go_to`, která předá řízení z aktuálně vykonávané úlohy do úlohy libovolné jiné, v případě potřeby tuto úlohu vytvoří. Tato funkce se dá použít k jednoduché implementaci vyjímek s jednou konkrétní úlohou, která řeší `catch`.

Dalším druhem změny za běhu jsou změny v objektech – změny hodnot položek, například uložení dat do proměnné `Data`. Tyto změny řeší třída `First` metodami `__set` a `__get`, kdy při jakékoliv změně v položkách dojde buď k vytvoření kopie definice a následnému uložení do databáze, nebo pouze k aktualizaci již uložené položky. Rozhodnutí vytvořit kopii místo aktualizovat se provádí na základě globálnosti dané definice.

6.4 Monitorování a verzování

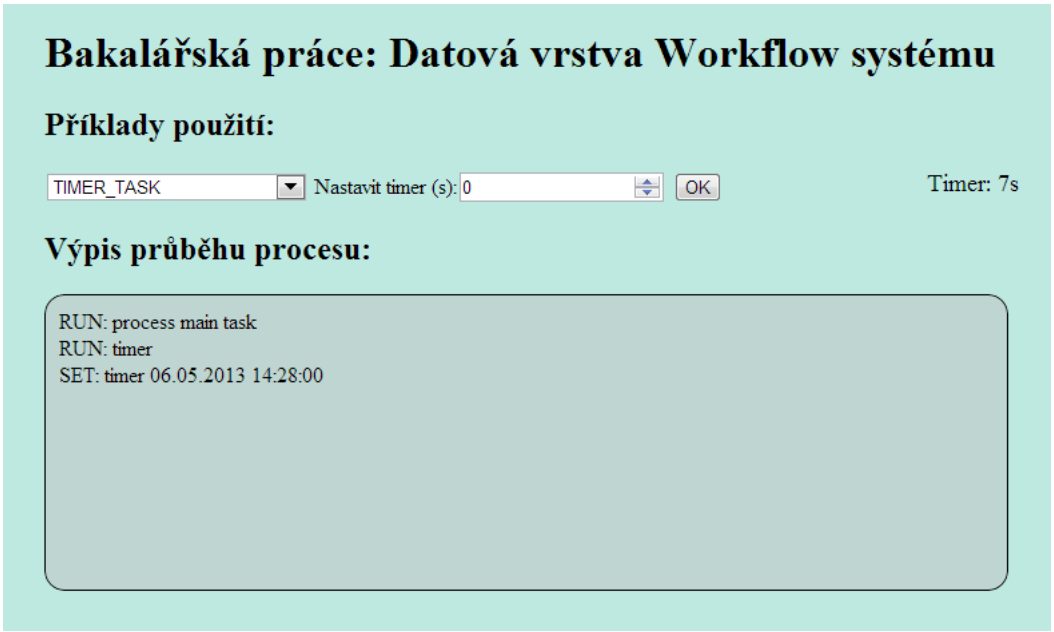
Vzhledem k nutnosti uchování historie již dokončených procesů a tasků se bude rychle zvyšovat počet záznamů v databázi, což může také vést ke snížení výkonu systému. Tomuto by šlo předejít odkládáním starých záznamů do jiné tabulky. Každopádně historické záznamy existují v tabulkách `task` a `process` spolu s lokálními definicemi v tabulce `definitionhead`, ale pro účely monitorování by bylo vhodné historii rozšířit i o postupné popisy změn za běhu. Například by stačilo přidat tabulku `log`, kam by metoda `__set` ukládala každé své volání s relevantními daty. Monitorování jsem však více neřešila, je to nad rámec této práce.

Verzování je zohledněno ve funkci `getDefinition()`. Pokud je v referenci zmíněna verze, použije se příslušná verze z databáze. Pro plnohodnotné využití by bylo vhodné přidat omezení na verzi větší než, či menší než zadanou a dořešit interakci verzování s kopírováním definic při zápisu.

6.5 GUI a timer

Aby bylo možné jednoduše spouštět procesy, vytvořila jsem k tomuto jednoduché GUI viz. obrázek 6.3, kde je možné vidět i použití timeru. Na stránce si uživatel může vybrat z předdefinovaných procesů, spustit si je a sledovat, jak proces cestuje Workflow systémem. Součástí GUI jsou timery, které zapříčiňují uspávání a probouzení procesů. Z důvodu omezení PHP jazyka, bylo nutné timery implementovat v JavaScriptu.

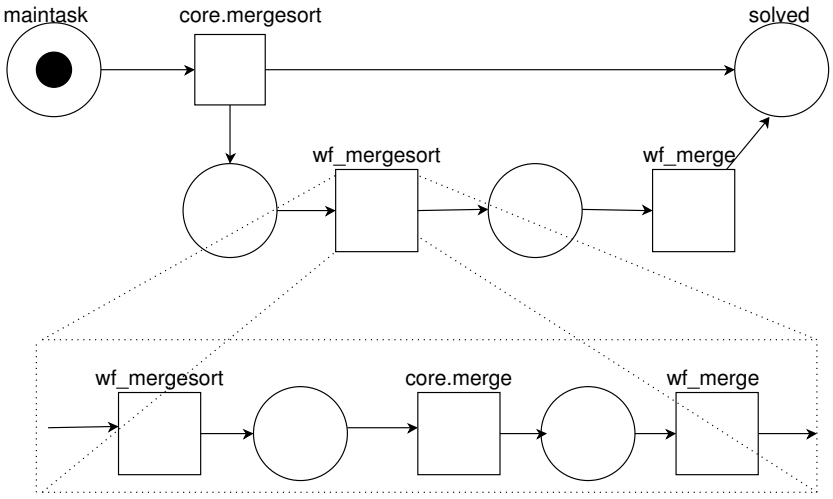
V okamžiku, kdy task zavolá metodu `timer()`, uloží se `id` tasku spolu s časem spuštění do seznamu v procesu. Tento task skončí a pokračuje se následujícími, metoda `ready` bude nadále vracet hodnotu `false` dokud nepřijde čas spuštění. GUI dostane pole timerů (čas, id procesu, id tasku) v JSON formátu a vytvoří si podle něj JavaScriptové timery (`setTimeout`), které poté daný proces spustí. Ruční spustění timeru je možné nastavit u všech předdefinovaných tasků.



Obrázek 6.3: Ukázka GUI

6.6 Ukázkový proces merge sort

Na obrázku 6.4 je znázorněn jednoduchý třídící algoritmus merge sort a jeho průchod Workflow systémem. Nejprve se vytvoří a spustí proces `maintask`, který je hlavním rodičem všech ostatních tasků. Následně je spuštěn task `core.mergesort`, který zavolá funkci `wf_mergesort`, ta si vytvoří podsíť, ve které probíhá samotná práce tohoto algoritmu. Task `core.mergesort` rozděluje pole na dvě poloviny, obě si rekurzivně předá v podsíti a výsledky spojí zavoláním tasku `core.merge`, který si vyvolá funkci `wf_merge`. Po dokončení spojování setříděných polí v této podsíti, se algoritmus ukončí a vypíše na výstup konečné setříděné pole.



Obrázek 6.4: Příklad algoritmu mergesort ve Workflow systému

6.7 Shrnutí implementace

Implementace vytvořená v rámci této práce slouží primárně k ilustraci návrhu datové vrstvy Workflow systému a průchodu procesu tímto systémem. Vztahují se na ni veškerá omezení vyplývající ze zvoleného programovacího jazyka (například nemožnost komunikace s již běžícím procesem).

Finální aplikace podporuje dynamickou změnu za běhu, pseudoparalelismus, lokální definice včetně spojování definic, timery, samostatné spouštění PHP kódů z databáze, modelování víceúrovňových procesů.

Objekty vytvořené za běhu jsou automaticky ukládány do databáze, a proto není náročné udělat vlastní definice za běhu či měnit již existující. Pro vytvoření není nutné přímo interagovat s databázovou vrstvou, protože konstruktoru můžeme předat JSON, případně nový objekt měnit.

Data tasků i definice jsou v jednom poli, veškeré zápisy jsou atomické, nemůže tedy nastat případ, že by si jiný task přečetl data v nekonzistentním stavu.

Všechny definice se nacházejí v jedné tabulce, není možné provádět typovou kontrolu na úrovni databáze. Pokud tělo nějaké definice obsahuje neznámou položku, tato položka se nepoužije, ale uživatel nedostane žádné varování. Typová kontrola by mohla probíhat v kódu, při použití nějakého nástroje pro tvorbu definic problém nenastane.

Zejména v případě dat tasku může existovat legitimní potřeba měnit nezávislé položky dat paralelně z různých tasků. Toto v našem návrhu není možné. Tasky by měly při pokusu o změnu dat kontrolovat, že mění verzi, ze které vycházely, případně zamykat daný řádek v tabulce už při čtení, což může negativně ovlivnit výkon Workflow systému.

Vytvořená implementace jádra Workflow systému není vhodná pro reálné použití. Takovýto systém by měl být schopen neustále běžet a spouštět procesy ve více vláknech. Skriptovací jazyky jsou vhodnými kandidáty vzhledem k jejich dynamičnosti. PHP je však navrženo primárně k jednorázovému obsloužení požadavku na serveru a skončení.

Kapitola 7

Manažerské procesy

Workflow nemusí být nutně použito pouze pro výpočetní procesy, ale často se používá pro manažerské procesy, které modelují chod nějaké organizace (tj. její procesy). Tyto procesy simulují skutečné procesy ve firmách, například administrativu, ale může se jednat i o nějaké výrobní procesy. Součástí manažerských procesů bývá:

7.1 Formuláře pro zadávání dat

Uživatelé procesu pomocí těchto formulářů zadávají data do procesu. Může se jednat o automaticky generované formuláře (specializovanou komponentou Workflow systému), poloautomaticky generované (s úpravou od tvůrců pocesu) nebo čistě naprogramované pomocí aplikace nezávislé na Workflow (tato aplikace ale s Workflow pochopitelně komunikuje) – toto už nejsou klasické Workflow formuláře, ale jedná se spíše o automatizované tasky – viz. sekce 7.2.

V jistých krocích procesu může být zpřístupněno zadání některých dat, v jiných krocích může toto být vyžadováno (může tomu tak být v tom stejném tasku nebo se může čekat na data až po nějaké delší době v jiném tasku).

Data mohou být pochopitelně zadána i nějakou externí aplikací automaticky – uživatelské rozhraní pro generování objednávky spustí proces objednávky a předá mu data, která v sobě mají relevantní informace – samotná objednávka bude pochopitelně v aplikační databázi, ale workflow bude obsahovat logiku procesu její obsluhy, kdežto databáze může mít v sobě pouze základní informace o stavech (např. je ve stavu posuzování, vyřizování, výroby, dopravy . . .), zapsané Workflow systém a spřístupněné pro jiné aplikace.

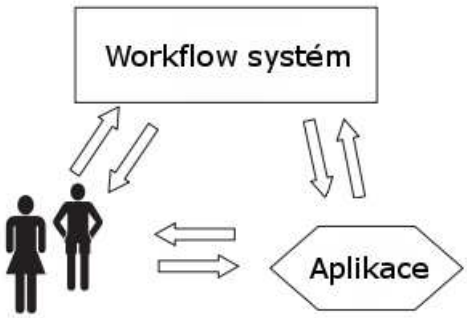
7.2 Automatizované tasky

Některé typy Workflow mohou umožňovat jen zadávání dat podle šablony procesu. Ty složitější v sobě obsahují i kódy, které mohou být různých typů:

- *řídící* – Starají se o řízení procesu – mohou rozhodovat o následujících krocích, mohou manipulovat s daty, měnit lokální definici procesu apod. V manažerských procesech tyto úlohy automatizují rozhodnutí manažera (musí se pochopitelně jednat o relativně jednoduchá předdefinovaná rozhodnutí nebo se rozhodují na základě dat, které jim někdo poskytnul). Například v procesu vyřízení objednávky se může jednat o to, zda bude objednávka zamítnuta nebo přijata na základě poskytnutých dat v procesu.

- *neřídící* – Tyto typy kódů nemají přístup k procesu, mohou pouze požadovat data na vstupu a na výstupu něco vracet. Mohou tedy být vykonány asynchronně a to nejen v rámci Workflow systému, ale jako např. webová služba. Tyto úlohy obvykle používají databázi jako společné uložiště dat. K aplikační databázi standardně nemá jádro Workflow přístup, může pouze volat kódy, které se o tento přístup starají. Lokální Workflow data by tedy neměla zrcadlit databázi kvůli konzistenci, ale měla by obsahovat procesní data, která se pravděpodobně v databázi nacházet nebudou a ostatní spíše ve formě cizích klíčů nebo ta, která se velmi pravděpodobně nebudou v průběhu procesu měnit – například můžeme evidovat rodné číslo, jméno a datum narození jako součást procesu, přestože je zrcadlíme z databáze.

Takže klasický manažerský proces bude obsahovat formuláře pro zadávání dat. Automatické až plně naprogramované tasky, které volají kódy vykonávající operace nad databází (či webové služby) a řídící tasky vykonávající proces – různé timery, rozhodování a změny procesů. Na obrázku 7.1 je zobrazeno propojení mezi uživateli Workflow a aplikací v pořadí, jak se plní jednotlivé úlohy. Příkladem může být timer, který po dokončení okamžitě pozastavuje proces a běh předává části procesu obsluhující timeout průběhu procesu. Pokud byla část procesu vykonána v dané lhůtě, timer se zruší.



Obrázek 7.1: Ukázka propojení Workflow systému mezi uživateli a aplikací

7.3 Lokální data

Lokální data procesu mohou náležet jednomu tasku nebo být sdílená v rámci podsítě, či celého procesu – to vše umí tento framework využít. Data mohou být také posílána společně s během vykonání mezi tasky nebo i mimo běh vykonání – to mohou zajistit řídící kódy. Pro manažerské procesy jsou využity všechny typy dat (jak sdílená, tak putující), simuluje se posílání dokumentů po procesu.

7.4 Uživatelé Workflow

Uživatelé jsou součástí manažerského Workflow systému, jejich implementace do jádra by byla již nad rámec práce. Uživatelé mohou mít různá práva k zadání dat, které zpřístupňuje task – např. zadání dat pro proces a automatizované tasky nebo dat potřebných k rozhodnutí (OR Split).

Wokflow systém se nemusí chovat k uživatelům jen pasivně, ale může jim pomocí GUI zobrazovat seznam procesů a tasků, které mohou vykonat. Může také pomoci naplánovat a rozdělit práci mezi lidi tak, aby byli pracovníci optimálně vytíženi.

Uživatel může taktéž mít právo přeskočit vykonávání nějakého tasku. Mohou existovat tasky, které čekají pouze na pokyn uživatele (budou čekat, dokud je manažer neschválí) nebo se může jednat o AND split a Join (všichni manažeři to musí schválit) či sekvenční spracování (to podléhá schválení) atd. V neposlední řadě eviduje, kdo jaký task vykonal. To je vhodné pro bezpečnostní audit, ale i pro monitorování a analýzu výkonnosti.

7.5 Zobrazování procesů a procesy obsluhy výjimek

Standardní běh firemního procesu může být jen malá část celého procesu. Velkou část může tvořit řešení nestandardních výjímek . Překročení časové lhůty, rollback procesu je taktéž složitá věc, která musí být provedena explicitně u každého procesu. Proto systém může při zobrazování umožňovat nastavování zobrazovací vrstvy, kdy každá část procesu obsahuje název vrstvy a uživatel si vybere, které vrstvy chce vidět. Jádro má pro toto podporu, ale implementace uživatelského rozhraní by byla nad rámec bakalářské práce. Nicméně je to důležitá součást Workflow, protože každý si proces zobrazí, jak potřebuje. Někteří uživatelé dokonce mohou mít v procesu nastavené, co vše mohou vidět. Uživatelé pochopitelně mají práva nastavená standardně pomocí rolí (mohou patřit až do několika rolí).

Kapitola 8

Závěr

Cílem práce byl návrh vhodné datové reprezentace Workflow systému. Vzhledem k variantám a požadavkům různých Workflow systémů bylo těžké najít nějaký společný základ. Důraz byl kladen spíše na jednoduchost, rozšiřitelnost a obecnou použitelnost návrhu, než na specifický Workflow systém. Z těchto důvodů je můj návrh relativně jednoduchý, ale vynahrazuje si to rozšiřitelností a přenositelností.

Databázové schéma je možno snadno převést do jiných databází. Můžeme použít libovolnou relační databázi včetně těch, které nepodporují datový typ JSON. Tato podpora je výhodou, nikoli podmínkou. Je možno použít datový typ text. Při převodu do dokumentových databází stačí přidat indexy na položky v nejvyšší úrovni.

V takto navržené databázi je velmi jednoduché změnit či přidat datové typy použité ve Workflow systému. Například rozhodneme-li se přidat položku timeout do `TaskInstance`, nemusíme nijak měnit databázovou vrstvu, stačí jen upravit příslušné řádky v databázi. V případě potřeby může být celá definice procesu uložena v jednom řádku tabulky nebo také libovolně rozdělená na jednotlivé definice.

Naprogramování samotné části jádra Workflow systému bylo nejsložitější a vyskytlo se zde mnoho problémů. Brzy se ukázalo, že by bylo vhodné udržovat data vždy jen v jednom formátu. Střídání odkazů, JSON dat, polí a objektů bylo sice třeba, ale mohlo být více izolované. Možnost měnit definice za běhu spolu s děděním definic vede k nejasnosti, kterou definici měnit. Obnovování procesu s takto změněnou definicí je implementováno velmi křehce. Dalším úskalím bylo rozhodnout se, kdy použít již existující instanci podle reference a kdy vytvořit novou (přestože již existuje).

Ve spojení se systémy pro tvorbu procesů a generování uživatelského rozhraní pro Workflow (viz [2]), by byl takovýto systém jednoduše použitelný. Není však stavěn na přímé psaní kódu v jednotlivých tascích. Toto má podporu, ale je vhodnější používat již hotové funkce.

Workflow systémy jsou průběžně nasazovány v různých firmách, ať již k řízení výroby či firemních procesů. Zatím není dostupný žádný univerzální systém, který by pokryl všechny požadavky a byl by dostatečně rozšiřitelný, nabízí se tu prostor pro růst. V průběhu tvorby bakalářské práce se naskytla příležitost použít tento návrh v praxi. Jedná se o systém správy a zabezpečení rodinného domu, který umožňuje vlastníkovi definovat chování pro případ nenadálé události. V takovém případě informuje majitele v reálném čase a umožňuje mu podniknout jinou akci. Nad implementovaným jádrem bychom museli udělat vrstvu pro práci se senzory a aktuátory nacházejícími se v domě. Takovýto systém bude pravděpodobně implementován v JavaScriptu.

Literatura

[1] Aalst van der, W., Hee van, K.: *Workflow Management: Models, Methods, and Systems*. The MIT Press Cambridge, Massachusetts London, England, 2002, ISBN 0-262-01189-1.

[2] Kateřina Šímová: *Generování uživatelského rozhraní pro Workflow systém*. Bakalářská práce, Vysoké učení technické v Brně, 2013, [bude obhájena v červnu 2013].

[3] Marlon, D., Aalst van der, W., Hoftstede, A.: *Process-Aware Information Systems : Bridging People and Software through Process Technology*. Wiley-Interscience, 2005, ISBN 978-0-471-66306-5.

[4] Redmond Eric, W. J. R.: *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf Book, 2012, ISBN 978-1-93435-692-0.

[5] Rob Peter, Coronel Carlos: *Database Systems - Design, Implementation, and Management*. Thomson, 2004, ISBN 978-0-619-21372-5.

[6] Silberschatz A., Korth H. and Sudarshan S.: *Database System Concepts*. McGraw-Hill, 2005, ISBN 978-0-07-295886-7.

[7] Stanczyk, S.; Champion, B.; Leyton, R.: *Theory and Practice of Relational Databases*. Taylor & Francis Group, 2001, ISBN 978-0-415-24702-3.

Příloha A

Obsah CD

- **bakalarka.sql** – databázové schéma, data definic
- **bakalarka.pdf** – technická zpráva v elektronické podobě
- **latex/*** – zdrojové kódy technické zprávy
- **kod/*** – zdrojové kódy implementace detailněji níže
  - **algoritmus.php** – algoritmus euklid hledání největšího společného dělitele
  - **app.js** – javascriptový timer
  - **composer.json** – popis externích závislostí
  - **def_beh.php** – kód pro testovací data – `eval`
  - **definice.php** – jádro Workflow systému
  - **dibi.php** – externí knihovna pro práci s databází
  - **if_while.php** – kód pro testovací data – `if, for, while`
  - **index.php** – jednoduché GUI Workflow systému
  - **jquery-1.9.1.min.js** – externí knihovna jquery
  - **mergesort.php** – algoritmus mergesort ve Workflow a v PHP podobě
  - **podprogram.php** – kód pro testovací data – `call`
  - **style.css** – kaskádové styly pro GUI
  - **vendor/*** – zdrojové kódy externích knihoven

Příloha B

Postup instalace

Ke spuštění této práce je potřeba následující software: PostgreSQL 9.2+, PHP 5.4+ a Apache od verze 1.2.

Postup je následující:

1. Naimportování databáze ze souboru *bakalarka.sql*
2. Nastavení adresáře *kód* jako DocumentRoot
3. Upravit soubor *dibi.php* – přístupy k databázi

Příloha C

Popis tříd

- **class SetArray** – Pomocná třída obalující pole, spolu s First usnadňuje automatickou aktualizaci databáze při změně. Jedná se o referenci na pole s informací o jeho vlastníkovi.
- **class First** – Předek všech následujících tříd. Poskytuje defaultní gettery a settery, které aktualizují databázi při změně. Defaultní getter také provádí automatickou dereferenci hodnot typu DefinitionReference. Udržuje informace o stromové hierarchii.
- **class Inherit** – Prázdná třída reprezentující pole děděná z obecnějších definic.
- **class DefinitionHead** – Hlavička definice – společná data pro všechny definice.
- **class DefinitionReference** – Třída poskytuje pojmenované odkazy na libovolné definice, včetně omezení na verzi a platnost definice. Také umožňuje nalezené definice částečně modifikovat.
- **class ProcessDefinition** – Definice procesu. Obsahuje pouze procesu specifické definice a odkaz na behaviour hlavního tasku.
- **class TaskDefinition** – Definice jednoho tasku, obsahuje informace o vztahu tasku k ostatním taskům na stejné úrovni, typu směrování, data procesu a odkaz na jeho chování.
- **class TaskBehavior** – Samostatné chování procesu. Obsahuje buď (PHP) kód, nebo definici podsítě. V případě, že obsahuje oboje, kód má vždy přednost před podsítí a musí tuto podsíť explicitně spouštět.
- **class Subnet** – Definice podsítě. Stejně jako proces, obsahuje definice specifické pro podsíť a odkaz na hlavní task. Na rozdíl od ProcessDefinition však odkazuje na TaskDefinition, nikoli TaskBehavior.
- **class ProcessInstance** – Instance již běžícího procesu. Udržuje informace specifické pro celý strom procesu – definice, spuštěné úlohy, i metody pro práci s timery.
- **class TaskInstance** – Instance jedné konkrétní úlohy. Obsahuje odkazy na definice i data instance, stará se o vytváření či načítání úlohy z databáze, implementuje metody pro spouštění tasků, hledání běžících tasků podle definice i postupné spouštění jeho následníků.