**TECHNICKÁ UNIVERZITA V LIBERCI**
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 – Informační technologie
Studijní obor: 1802R007 – Informační technologie

# ePCBlab – server a databáze

# ePCBlab – server and database

**Bakalářská práce**

Autor: **Martin Pomezný**
Vedoucí práce: Ing. Leoš Petržílka
Konzultant: prof. Ing. Zdeněk Plíva, Ph.D.

V Liberci 11. 5. 2011

(Originál zadání práce)

## Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Děkuji Jakubu Petržílkovi, autorovi klientských částí ePCBLabu, za vstřícnou spolupráci při vývoji a návrhu aplikace. Děkuji také svému vedoucímu práce a zároveň vedoucímu PCB Laboratoře Ing. Leoši Petržílkovi za konzultace detailů a uvedení do problematiky.

## Abstrakt

Tento text popisuje řešení konkrétního systému na správu a evidenci zakázek v laboratoři pro výrobu plošných spojů při Technické univerzitě v Liberci. Práce zahrnuje zhodnocení různých řešení a popis použitých technologií a postupů. S jeho pomocí by mělo být možné na systém navázat další aplikací či rozšířením stávající funkcionality, ať už přímo na straně serveru, nebo ve formě klientské aplikace. Text obsahuje taktéž popis nově vytvořeného protokolu pro přenos datových balíčků ve formě XML dokumentů po síti internet, s možností specifikovat nad daty zámky a asynchronně informovat klienty o změnách. Výsledkem práce je aplikace napsaná v jazyce Java s implementací nového protokolu XOE.

Klíčová slova: XML, PCBlab, Hibernate, XOE, správa zakázek

## Abstract

Abstract

This text describes a solution of a specified system for managing and evidence of orders in a printed circuit laboratory on Technical university of Liberec. This paper contains the evaluation of different solutions and the description of used technologies and methods. With its help, it should be possible to continue the development with another application or by extending current functionality, possibly on server side or with another client application. This text also contains the description of a newly created protocol for transferring data packages in form of XML documents over the internet. It has the ability to specify lock on data and asynchronously notify clients about changes. The result of this work is an application written in Java with implementation of new protocol XOE.

Keywords: XML, PCBlab, Hibernate, XOE, order management

# Obsah

Prohlášení .......... 3
Abstrakt .......... 5
Abstract .......... 5
Obsah .......... 6
Seznam zkratek .......... 8
1 Úvod .......... 9
1.1 Volba protokolu .......... 9
1.2 Volba datového úložiště .......... 10
1.3 Komunikační kanál .......... 10
2 XOE .......... 11
2.1 Stavba zprávy .......... 11
2.1.1 Request .......... 11
2.1.2 Response .......... 11
2.1.3 Response status .......... 12
2.1.4 Notification .......... 12
2.1.5 Ukončení zprávy .......... 13
2.2 Popis spojení .......... 13
2.3 Udržení spojení .......... 14
2.4 Dialog .......... 14
2.5 Zámky .......... 14
2.6 Režie .......... 15
2.7 Implementace v Javě .......... 15
2.7.1 XML un/marshalling .......... 16
2.8 Ostatní implementace .......... 16
2.9 Využitelnost .......... 17
3 Datový model .......... 18
3.1 Analýza .......... 18
3.2 Popis tříd .......... 18
3.2.1 DataContainer .......... 18
3.2.2 PcbData .......... 18
3.2.3 Pattern třídy .......... 19
3.2.4 Výčtové typy .......... 19
3.2.5 ProvedenaOperace .......... 19
3.2.6 Uzivatel .......... 19
3.2.7 Zamek .......... 20
3.2.8 Konstanta .......... 20
3.2.9 Material .......... 20
3.2.10 Zakazka .......... 20

3.2.11 Zakaznik ........ 20
3.3 Cyklus zakázky ........ 21
3.4 Přenos dat ........ 22
3.5 Aktualizace dat ........ 22
3.6 Režie dat ........ 22
4 Aplikace ........ 23
4.1 Požadavky ........ 23
4.2 Keystore ........ 23
4.3 Log událostí ........ 24
4.4 Konfigurace a instalace ........ 25
4.5 Struktura aplikace ........ 26
4.6 Aplikační logika ........ 26
4.6.1 WatcherThread ........ 27
4.6.2 WorkerThread ........ 27
4.6.3 Helpers ........ 28
4.7 Vývojové prostředí ........ 28
5 Datové úložiště, Hibernate ........ 29
5.1 Hibernate ........ 29
5.2 Hibernate v praxi ........ 30
5.2.1 Anotace tříd ........ 30
5.2.2 Selekce dat ........ 30
5.3 MySQL ........ 31
6 Závěr ........ 32
6.1 Volba technologií ........ 32
6.2 Dosažené výsledky ........ 32
6.3 Možnost navázání ........ 33
Seznam použité literatury ........ 34

## Seznam zkratek

**XOE** XML Object Exchange

**XML** Extensible Markup Language

**PCB** Printed circuit board

**SOAP** Simple Object Access Protocol

**HTTP** Hypertext Transfer Protocol

**SQL** Structured Query Language

**ORM** Object-Relational Mapping

**TCP/IP** Transmission Control Protocol/Internet Protocol

**XSD** XML Schema Definition

**ASCII** American Standard Code for Information Interchange

**SSL** Secure Sockets Layer

**JAXB** Java Architecture for XML Binding

**JDBC** Java Database Connectivity

# 1 Úvod

Cílem této práce je vytvořit serverovou aplikaci v jazyce Java, která bude uchovávat informace o zakázkách PCBlab. Měla by specifikovat rozhraní pro přístup klientským aplikacím, logiku výpočtu cen a způsob uchování dat jako takových. V následujících kapitolách jsou vysvětleny základní funkční principy aplikace a zvolený protokol komunikace. Tato práce by měla sloužit jako dokumentace k vytvořenému protokolu nazvanému XML Object Exchange (XOE) a server softwaru který vznikl. Text zprostředkovává pohled do vnitřního fungování a kódu aplikace. Jako celek navazuje na bakalářský projekt, který se zabýval především analýzou problému, rozdělením na menší, lépe uchopitelné bloky, a návrhem datových struktur. Průběh zakázky v laboratoři je relativně složitý proces a kalkulace ceny také není triviální problém, avšak při návrhu a následující tvorbě byl kladen důraz na maximální jednoduchost a srozumitelnost použitých postupů a funkčních principů. Především kvůli možnosti navázat na tuto práci a rozšířit ji o další funkcionalitu.

## 1.1 Volba protokolu

Jako první bylo nutno stanovit komunikační kanál pro výměnu zpráv s klienty. V úvahu připadaly protokoly pro přenos dat přes počítačovou síť, které jsou otevřené a poskytují dostatečnou volnost pro přenášená data. Zvolený protokol by také měl umět přenášet zprávy asynchronně od serveru ke klientům, jako upozornění na nové události. Bylo by též vhodné zabezpečit přenášená data proti případnému útoku šifrováním.

Účelu by, po určitých úpravách, dobře posloužil například protokol SOAP, který přenáší zprávy založené na XML a využívá k tomu standardizovaný protokol HTTP. To s sebou přináší i řadu nevýhod. Bylo by třeba na serveru více programového vybavení a přenášení upozornění ke klientům by se řešilo zbytečně komplikovaně a mohlo by v budoucnu přinést problémy v podobě mírných implementačních odlišností v různých programovacích jazycích či platformách. Nakonec jsme však tuto možnost zavrhli a přiklonili se k vlastní implementaci, kterou jsme pojmenovali XML Object Exchange. Navzdory tomu, že návrh a implementace nového protokolu jsou časově náročnější než použití hotového, čas potřebný na studium a úpravu pro naše konkrétní potřeby tuto nevýhodu vyvážil a ve výsledku jsou oba způsoby víceméně stejně náročné. Navíc přesně víme, jak náš protokol reaguje a jak vnitřně funguje. Výsledkem je efektivní a jednoduchý protokol přesně na míru. XML je použito především pro jeho

univerzálnost a použitelnost napříč programovacími jazyky. Toho bychom například nativní serializací objektu v Javě dosahovali jen velmi těžko a klientské aplikace by tudíž musely být rovněž v Javě. Na návrhu tohoto protokolu jsme spolupracovali s Jakubem Petržílkou, který je tvůrcem klientské aplikace a webového rozhraní.

## 1.2 Volba datového úložiště

Při volbě serveru jsme potřebovali vybrat vhodnou relační databázi. Jiná řešení, jako například ukládání do binárních souborů nebo ukládání dat jako XML jsme nebrali v potaz, protože oproti relačním databázím se s nimi pracuje velmi složitě, dotazování nad daty by se řešilo obtížně a řešení by bylo paměťově náročné. MySQL databáze splňuje všechny požadavky a potřeby tohoto projektu. Navíc je dnes víceméně standardem na linuxovém serveru. Pro práci s databází jsme zvolili Framework Hibernate od společnosti JBoss. Hibernate umožňuje objektově relační mapování dat (ORM) a řeší otázky uchování dat i po vypnutí aplikace uložením do databáze. Poskytuje tak programátorovi komfortní práci s objekty, což výrazně zvyšuje efektivitu práce.

## 1.3 Komunikační kanál

Jako komunikační kanál mezi klienty slouží TCP/IP socketové spojení zabezpečené kryptografickým protokolem SSL. Server naslouchá na portu 6789 a čeká příchozí spojení od klienta. Ten si ověří certifikát serveru, a pokud jej uzná tak se připojí. Komunikace je obousměrná a šifrovaná, kanál tudíž může být považován za bezpečný.

# 2 XOE

XML Object Exchange je protokol vytvořený pro potřeby této práce, ale jeho využití by se dalo velmi snadno rozšířit a použít jej i k jiným projektům. XOE obaluje data a o jejich obsah a korektnost se stará aplikace nad ním. Protokol umožňuje asynchronně přenášet upozornění vyvolané jedním klientem ke všem ostatním a definuje zámky nad daty. Jeho stavba je podobná protokolu SOAP a chybové kódy jsou definovány čísly, obdobně jako například u HTTP [1].

## 2.1 Stavba zprávy

Stavba jednotlivých zpráv je definována XSD šablonami. XOE definuje 3 druhy zpráv a to: *Notification*, *Response* a *Request*. Zpráva začíná značkou *<?xml>,* v jejíchž atributech je definováno kódování a verze xml. Poté následuje značka, která již přímo odpovídá typu zprávy a atributem *xsi:noNamespaceSchemaLocation* specifikuje šablonu, podle které je má server validovat. Teprve pokud je zpráva uznána za validní, je zpracována a část *Data* je postoupena vyšší vrstvě.

### 2.1.1 Request

Zpráva, která přenáší data směrem od klienta k serveru a vyjadřuje tak požadavek klienta. *Request* má element Id, kterým si klient označuje zprávu, aby ji následně mohl správně spárovat s odpovědí. Pokud první (autentizační) požadavek obsahuje tag *Notify*, klient má zájem dostávat upozornění, v opačném případě mu nebudou zasílána. V autentizačním dotazu se také musí vyskytnout blok Identity, jež obsahuje *Username* – login uživatele a *PasswordHash* – SHA1 otisk uživatelova hesla. Požadavek má 7 typů:

Get: Požadavek na data ze serveru.

Put: Požadavek vložení dat na server.

Lock: Uzamčení dat.

Update: Aktualizace stávajících dat

Delete: Smazání.

Heartbeat: Klient periodicky udržuje spojení otevřené.

V neposlední řadě je zastoupen také tag *Data*, ve kterém jsou uložena přenášená data.

### 2.1.2 Response

Zpráva přenášející data od serveru ke klientovi, na základě přijatého požadavku. Obsahuje tag *Id*, který vyjadřuje, na který požadavek se odpovídá. V případě většího

množství zpráv a zdržení v síti klientovi velmi usnadňuje práci s odpověďmi. Jádro odpovědi tvoří stejně jako u požadavku tag *Data*, v němž je datová opověď serveru. První odeslaný response (odpověď na auth) obsahuje blok *Hello*, v němž server klientovi řekne verzi serveru – version a interval ve kterém má udržovat spojení – *Heartbeat*. Velmi důležitou částí je tag Status. Ten vyjadřuje, jak server požadavek zpracoval a s jakým výsledkem.

### 2.1.3 Response status

Odpověď může mít 12 stavů, každý definovaný trojciferným číslem. První číslo určuje skupinu:1 (Server OK statusy), 2 (data OK statusy), 3 (server FAIL statusy), 4 (data FAIL statusy). Následuje jejich výčet a vysvětlení.

100 – HEARTBEAT_CONFIRMATION – potvrzení udržení spojení
101 – AUTHENTICATION_OK – Heslo a jméno uživatele je správné
200 – QUERY_OK – Dotaz byl úspěšně proveden
210 – ITEM_LOCKED – Data uzamčena úspěšně
300 – BAD_REQUEST – Špatný požadavek
301 – AUTHENTICATION_FAILED – Heslo nebo jméno nesouhlasí
401 – QUERY_FAILED_INCORRECT_DATA – Selhání kvůli špatnému zadání
402 – QUERY_FAILED_LOCKED – Dotaz se neprovedl, protože položka je zamčená
403 – QUERY_NOT_PERMITTED – Dotaz se neprovedl. Není povolen nebo nemá uživatel dostatečná oprávnění
404 – QUERY_FAILED_NO_SUCH_DATA – Požadavku neodpovídají žádná data
413 – LOCK_FAILED_LOCKED_ALREADY – Položku nelze zamknout, protože už je uzamčená
414 – LOCK_FAILED_NO_SUCH_ITEM – položka neexistuje, nelze ji tak uzamknout

### 2.1.4 Notification

Zpráva, která se ke klientům přenáší na základě aktivit jiných klientů. Pokud některý z připojených klientů vyvolá akci, o které je nutné vyrozumět ostatní klienty, je vygenerována notifikace a ta se následně všem ostatním odešle. Ti jsou takto okamžitě, téměř v reálném čase, vyrozuměni o změně. Mezi tyto události patří uzamčení, aktualizace stávajícího záznamu, přidání nového a smazání záznamu. Tomu také odpovídají typy upozornění – *Locked*, *Updated*, *Added*, *Deleted* – v tagu *Type*.

Notifikace samozřejmě také obsahuje tag *Data*, ve kterém je obsah, kterého se upozornění týká.

### 2.1.5 Ukončení zprávy

Zprávy jsou kvůli snazšímu rozpoznání konce odděleny netisknutelným znakem číslo 4 v ASCII tabulce, znamenající konec přenosu (End of Transmission). Zpráva se sestavuje, dokud se nenarazí na tento znak. Ten je zahozen a celek je předán ke kontrole a dekódování. Znak 4 není v XML povolen [2] a tak lze ho lze považovat za bezpečný oddělovač, bez rizika kolize s obálkou XOE nebo přenášenými daty.

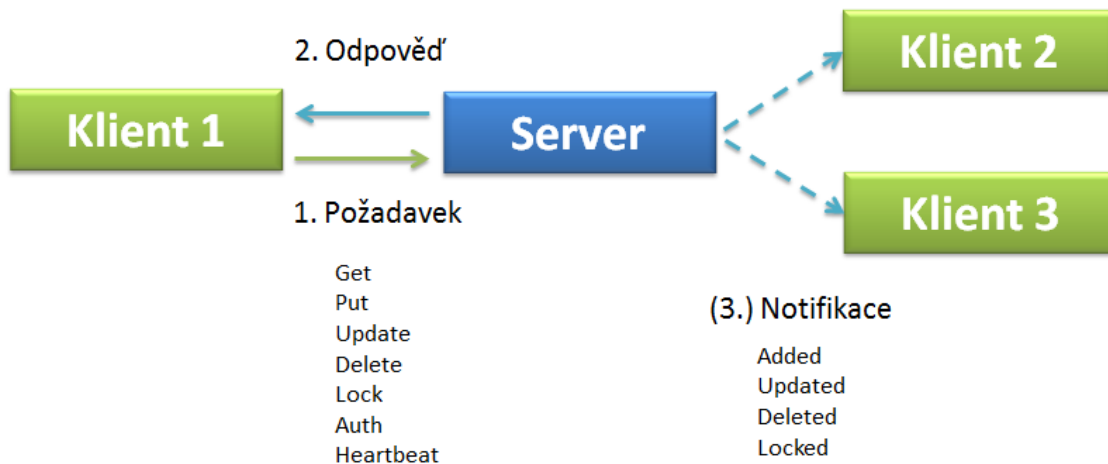
## 2.2 Popis spojení

Po spojení se musí klient co nejdříve autentizovat, mezním je časový úsek nastavený na serveru jako maximální prodleva mezi zprávami (heartbeat interval). Autentizací se rozumí odeslat první požadavek (*Request*) typu *Auth*, který musí obsahovat blok Identity, v něm klient odesílá své přihlašovací údaje – login a SHA1 otisk hesla. Server je ověří proti databázi, a pokud se neshodují, či v databázi nejsou, je odeslána odpověď se stavem 301 – autentizace selhala. Následně je socket ze strany serveru uzavřen. V opačném případě server odešle odpověď s kódem 101 a v bloku *Hello* heartbeat interval a verzi serveru. Také si zapamatuje, jestli tento klient v autentizačním požadavku uvedl tagem *Notify*, že chce po dobu svého spojení odebírat upozornění. Teprve potom je možné vyhovět klientovým požadavkům a zprostředkovat přístup k datům. Jak je z předchozího patrné, klient neví jaký je heartbeat interval a proto nemůže spojení udržovat (z bezpečnostních důvodů to ani není technicky možné). Programátor klienta by proto měl vzít tento fakt v potaz a nejprve si vyžádat od uživatele jméno a heslo a pak teprve otevřít spojení a vzápětí odeslat autentizaci. Vzhledem k tomu, že zabezpečení přenosu je šifrováno SSL a server se identifikuje certifikátem, může klient snadno ověřit, zdali je protistrana skutečně tím za koho se vydává. Neautorizovaným klientům nejsou z bezpečnostních důvodů odesílány notifikace, neboť by se data mohla dostat do ruky nepovolanému člověku, potažmo útočníkovi. Pokud chce klient využívat notifikací, měl by používat více vláken, aby na tyto události mohl reagovat dostatečně pružně a nezávisle na stavu zbytku aplikace, popř. využívat neblokujících funkcí.

## 2.3 Udržení spojení

Pro udržení aktivního spojení mezi serverem a klientem je nezbytné, aby se klient serveru pravidelně ozýval a ten tak měl jistotu, že je stále připojen. K tomuto účelu slouží typ požadavku *Heartbeat*. Pokud tak klient neučiní, server jej považuje za nečinného a spojení s ním je ukončeno.

## 2.4 Dialog




*Obr. 1 – Průběh XOE dialogu*

1. Klient nejprve odešle požadavek na server.
2. Server jej zpracuje a odešle příslušnou odpověď.
3. Pokud požadavek vyvolal změnu, o které mají vědět ostatní, odešle se jim notifikace

## 2.5 Zámky

Protokol umožňuje práci více uživatelů nad jednou množinou dat, a proto je nutné zavést systém zámků. Pokud by dva nezávislí uživatelé editovali jednu a tutéž položku mohlo by dojít ke kolizi a jeden z nich by přišel o své změny. Podobně pokud by jeden uživatel položku editoval a druhý ji v ten moment smazal. XOE proto umožňuje uživateli položku zamknout a ta je tak výhradně pod jeho kontrolou.

Zámek se váže na jednoznačný identifikátor (id) dat, jejich typ a uživatele, jenž je zamyká. Z toho plyne, že pokud se uživatel odpojí (například výpadkem sítě nebo přesunem notebooku) a znovu připojí, je položka stále uzamčena jen pro něj. Podle potřeb aplikace je nutné nastavit si maximální dobu platnosti zámku, aby nemohlo dojít k situaci, kdy je položka zamčena a není, kdo by ji odemknul. Pokud je zámek starší než

definovaná doba platnosti, je ignorován a položka se jeví jako nezamčená. Uzamčení položky klientem vyvolává notifikaci typu *Locked* a ta je tak po provedení akce okamžitě odeslána všem klientům spolu s daty která byla uzamčena. Notifikace se pochopitelně neodešle klientům, kteří si je nepřejí dostávat a tomu kdo tuto událost vyvolal. Odemčení položky je možné provést dvěma způsoby. Pokud se data změnila tak klient odesílá požadavek typu *Update*. Ten, pokud je úspěšný, odemkne položku a odešle ostatním klientům její aktualizovanou verzi. Pokud žádná data nemění a jen chce položku odemknout, odešle požadavek typu *Get*, server ji odemkne a stávající verzi odešle ostatním klientům na znamení odemčení. Odemykající požadavek musí obsahovat konkrétní identifikátor zamčené položky, jinak nebude odemčena. Toto opatření bylo zavedeno, aby nemohlo dojít k náhodnému odemčení při hromadném dotazu na více zakázek, mezi kterými by byla i zamčená položka.

## 2.6 Režie

Protokol má, díky své podstatě značkovacího jazyka jistou pevnou režii. Ta sestává z úvodního autentizačního požadavku a dále v každé zprávě z dat XOE, kterými jsou Id, typ požadavku/stavový kód odpovědi a v neposlední řadě tagy dělící tato data. Její velikost není závislá na velikosti přenášených dat, a proto ji nelze procentuálně vyjádřit. Čím větší je přenášený obsah tím se zastoupení režie ve zprávě zmenšuje. Z předchozího tedy jasně vyplívá, že XOE není příliš vhodný protokol pro přenos po malých množstvích, neboť ve výsledku by režie mohla tvořit i větší část než data.

## 2.7 Implementace v Javě

V samotném programu je XOE koncentrováno v balíčku (package) xoe. Ten obsahuje celkem šest souborů, z toho pět pro třídy a jeden pro definici výčtového datového typu pro stav odpovědi. Třídy *XoeResponse*, *XoeRequest* a *XoeNotification* reprezentují jednotlivé druhy zpráv. Ve své definici kopírují strukturu XML zprávy a XoeResponse a XoeNotification navíc obsahují statický Logger pro zaznamenávání událostí a chyb do logu aplikace a metodu toXML, která vrací instanci třídy serializovanou do XML jako String. Instanční proměnné jsou nastaveny jako private a přístup k nim umožňují getter a setter metody. Třída *XoeRequestFactory* slouží k deserializaci objektu z řetězce obsahujícího jeho XML reprezentaci, její úkol spočívá především v prvotní kontrole příchozích dat. Pokud data nelze korektně zpracovat je zapsán záznam do logu a je vrácen prázdný XoeRequest typu Bad. Okolí se tak může

spolehnout, že vždy obdrží požadavek a nikoli null protože na každý požadavek, byť špatný, by měla být odeslána odpověď. Třída je definována jako abstract, tudíž nelze vytvořit její instanci.

Třída *XoeResponseFactory* se stará a vytvoření odpovídajícího *XoeResponse* z XoeRequest. Fakticky už potřebuje znát strukturu, nebo alespoň rozhraní dat, která přenáší. De facto tak vytváří jakýsi most mezi čistým XOE a datovým modelem aplikace. Předaný požadavek zpracuje a podle jeho obsahu vytvoří odpověď a případně i notifikaci. V této aplikaci využívá rozhraní *PcbData* k získání klíčových informací pro zpracování požadavku.

### 2.7.1 XML un/marshalling

Marshalling zajišťuje standardní součást Javy a to Java Architecture for XML Binding (JAXB). Ta pomocí zadané třídy umožňuje velmi snadno zpracovat XML do objektu a naopak v podstatě bez asistence programátora. Tím značně zefektivňuje práci a lze předpokládat, že tato metoda bude rychlejší a spolehlivější, než vlastní, nově napsaná. Před samotným převedením se zkontroluje, zdali zpráva odpovídá šabloně (zdali je validní), a pokud ano, je vytvořen objekt XoeRequest. Vytvoří se JAXBContext pro příslušnou třídu a z té se vytvoří Unmarshaller. Ten pak sám převede příslušný vstup na odpovídající objekt. Marshalling probíhá obdobně, jen je z contextu vytvořen Marshaller. Tomu je nastaveno kódování utf-8 a xsd šablona. Předaný objekt je pak automaticky převeden na jeho XML reprezentaci.

Pro správnou funkčnost Marshalleru a Unmarshalleru je potřeba třídy upravit tzv. anotacemi. Jedná se o klíčová slova, jimž předchází znak „@“. Gettery instančních proměnných jsou opatřeny anotací @XmlElement. Té je nastaven atribut name, podle jména. V našem případě je nezbytný, protože XOE má atributy pojmenované s velkým počátečním písmenem, kdežto v jazyce Java je konvencí [3] zapisovat proměnné s malým počátečním písmenem a dále tzv. „velbloudí notací“. Důležitá je i anotace *@XmlRootElement*, která určuje kořenový element celého objektu. Loggery jsou anotovány jako *@XmlTransient*, to značí, že v XML se vůbec neprojeví a Marshaller je tak při zpracování vynechá.

## 2.8 Ostatní implementace

Xoe je implementován v primární klientské desktopové aplikaci pro platformu .NET napsaný v jazyce C#. Byla připravena také jeho odlehčená verze, která je napsaná

v Javě a slouží jako základ pro klientské webové rozhraní. Řešením těchto implementací se zabývá Jakub Petržílka ve své bakalářské práci. XOE je jen přenosový protokol a tudíž definuje jen formát a strukturu zpráv, jak na ně mají jednotlivé prvky reagovat, způsob autentizace a zabezpečení. Datové struktury pro uchování objektů, zámků a stavu aplikace nedefinuje a je tak čistě v režii programátora, který jej implementuje.

## 2.9 Využitelnost

XOE je velmi volně definovaný protokol pro přenos obecných zpráv a proto jej lze implementovat pro přenos libovolných dat po počítačové síti. V této práci je již částečně upraven pro konkrétní potřeby aplikace, avšak obecná část je použitelná do jiných projektů. Po vytvoření zcela obecného modelu jako samostatného balíčku by bylo možné jej vydat jako otevřené a všem dostupné řešení.

# 3 Datový model

Struktura dat je sdružena v balíčku *dataModel*. Ten obsahuje všechny nutné třídy, rozhraní a výčtové typy pro funkci programu a aplikační logiky. Pro své uchování využívá modul Hibernate a pro přenos XOE. Třídy jsou anotovány nad gettery XML anotacemi a nad instančními proměnnými Hibernate anotacemi.

## 3.1 Analýza

Datový model částečně čerpá z bakalářského projektu zpracovaného ve druhém ročníku společně s Jakubem Petržílkou. Projekt se zabýval strukturou databáze, protože v době jeho zpracování se počítalo se zpracováním programu jako celistvé webové aplikace postavené na jazyce PHP. Kvůli tomu bylo nutné jej částečně přepracovat, převést databázové entity na třídy a konvertovat datové typy proměnných. Analýza problému vycházela ze stávajících dokumentů programu Excel, který slouží jako evidence v současnosti. Dokumentace je velmi rozsáhlá a vzhledem k tomu, že obsahuje číselné hodnoty (konstanty) a postupy výpočtů, bylo nutné ji do datového modelu co nejvěrněji přenést. Při analýze a následné tvorbě modelu bylo také nutné nastudovat výrobní proces a model tomu náležitě uzpůsobit.

## 3.2 Popis tříd

Tato kapitola detailně popisuje účel hlavních tříd a rozhraní aplikace. Méně významné třídy jsou popsány stručně či sloučeny do jedné kapitoly.

### 3.2.1 DataContainer

Pokud je výsledkem dotazu více než jeden objekt, použije se *DataContainer* k obalení všech prvků. Obsahuje standardní JAXB anotace a konstruktor jímž vyplní své proměnné.

### 3.2.2 PcbData

Velmi důležité rozhraní pro datový model. Každý objekt, jenž chce být přenášen přes XOE musí toho rozhraní implementovat. Umožňuje získat nejnutnější údaje o objektu bez nutnosti znát jeho přesný datový typ a tím tak značně usnadňuje práci s přijatými daty. Objekt sám musí být schopen určit, zdali je platným a popř. se „validovat“ – doplnit stávající data, například spočítat cenu zakázky nebo přístupový kód. Rozhraní *PcbData* také specifikuje metodu getCriteria s její pomocí má objekt poskytnout kritéria podle své proměnné pattern. Postupně by tak měl projít její členské

proměnné, a pokud je nastavena, tak ji přidat mezi kritéria. Toto rozhraní implementují: *Konstanta*, *Material*, *Operace*, *Zakazka*, *Zakaznik*, *Zamek*.

### 3.2.3 Pattern třídy

Objekty, které budou klientovi poskytovány přes metodu get, by měly mít definovánu svou Pattern třídu a obsahovat ji jako instanční proměnnou. V té klient specifikuje, jaké objekty chce obdržet na principu kritérií, která musí platit pro daný objekt. Výsledek je definován jako konjunkce jednotlivých neprázdných parametrů. Pro řetězcové hodnoty je použito vyhodnocování bez ohledu na velikost písmen s výskytem kdekoli v položce. Pro číselné hodnoty je použito ostré porovnání. Množinové parametry (stav zakázky) jsou vyhodnoceny disjunkcí, kdy hodnota musí odpovídat alespoň jedné z podmínek. To umožňuje jedním dotazem sofistikovaněji určit množinu – například poptávky a probíhající zakázky. Výjimku tvoří datumOd a datumDo, kde atribut v objektu musí být větší nebo roven než datumOd a analogicky pro datumDo. Pattern třídy jsou: *PatternKonstanta*, *PatternMaterial*, *PatternOperace*, *PatternZakazka*, *PatternZakaznik*.

### 3.2.4 Výčtové typy

Uvedené datové typy umožňují logičtější a pro člověka srozumitelnější reprezentaci některých parametrů. Jejich implementace a použití je snazší než použití čísel či řetězců. V projektu jsou ty to výčtové typy: *DostupnostMaterialu* a *StavZakazky*.

### 3.2.5 ProvedenaOperace

Vzhledem k velké podobnosti tříd *ProvedenaOperace* a *ProvedenaOperaceVrtani* byla zavedena třída *ProvedenaOperaceAbstrakt*, která umožní Hibernate lepší uložení do jedné tabulky. Abstraktní třída definuje společné atributy a zbylé dvě ji rozšiřují.

### 3.2.6 Uzivatel

Třída reprezentující uživatele, který má přístup k datům, využívá ji XOE k autorizaci. V současné verzi obsahuje jen jméno a příjmení, login a otisk hesla. Do budoucna se však počítá s rozšířením o uživatelské role a oprávnění kde by tato třída byla rozšířena a její význam by značně vzrostl. To je také jeden z důvodů, proč je zařazena do balíčku dataModel a nikoli do xoe.

### 3.2.7 Zamek

Značí uzamčení položky, obsahuje její identifikátor, typ, čas uzamčení a uživatele, který ji zamkl. Staré zámky, i když zůstanou uloženy, nejsou brány v potaz a při první možné příležitosti jsou z databáze vyřazeny.

### 3.2.8 Konstanta

Konstanty jsou objekty, u kterých se počítá s použitím ve výpočtech a s jejich občasnou změnou (DPH, hodinová sazba), Každá konstanta má, kvůli historické důležitosti, stanovenou platnost dvěma daty. Pokud je platnost „do“ nenastavena, platí konstanta od určeného dále bez omezení. Toto pole, může klient využít i jako sdílené úložiště pro data programu, například obsah rolovacích nabídek.

### 3.2.9 Material

Reprezentuje materiál, který je možné použít na zakázku. Určuje jeho jednotky a cenu. S touto třídou úzce souvisí *PouzityMaterial*, který ji využívá a značí konkrétní množství a cenu použitého materiálu na konkrétní zakázku. Parametry desky plošného spoje představuje třída *ZakladniMaterial*, která rozšiřuje *Material*.

### 3.2.10 Zakazka

Zakázka je zpracována do této třídy. Obsahuje všechny nutné parametry a cenu si počítá pomocí metody validate. Využívá třídu *RozmerDesky*, která jen kvůli přehlednoti obaluje rozměry desky v milimetrech. Pro popis v jakém stavu se zakázka nachází je použita třída *ZakazkovaUdalost*, ta zajišťuje historické uchování, jak se s postupem času zakázka vyvíjela. Nejnovější stav zakázky je také zkopírován do instanční proměnné aktualniStav, čímž se za cenu malé redundance značně zjednoduší dotaz na zakázky podle jejich stavu. Třída *Operace* zaštiťuje požadované operace pro konkrétní zakázku. *ZvlastniPozadavek* symbolizuje zákazníkův specifický požadavek na zakázku, který může být také zvlášť zpoplatněn, podle domluvy se zákazníkem.

### 3.2.11 Zakaznik

Zákazník je na serveru reprezentován jako objekt typu *Zakaznik.* Uchovává všechny potřebné údaje o zákazníkovi a také reference na všechny jeho zakázky. Tato vlastnost není přes XML viditelná, ale možnost získat všechny zakázky zákazníka klient stále může.

## 3.3 Cyklus zakázky

*Obr. 2 – Vývojový diagram stavu zakázky*

Zakázka v průběhu svého zpracování prochází několika stavy a ty je nutné pro správnou funkčnost a jasnost definovat. Počáteční stav zakázky je závislý na jejím vzniku. Pokud zakázku poptá zákazník například z webového rozhraní, vstupuje na server jako *poptávka*. Tu personál zkontroluje a případně opraví či poupraví technologické parametry dle možností pracoviště. Zakázka tak přechází do stavu *nabídka*. Do tohoto stavu také může přejít přímo při vytvoření, pokud jej provedl personál. Pokud zákazník na nabídku nepřistoupí, zakázka je *stornována*, a případně posléze smazána. Může také přejít zpět do stavu *poptávka*, kdy zákazník upraví údaje a personál je reviduje a udělá opět nabídku. Pokud zákazník nabídku přijme, stává se z ní *objednávka*. V momentu zahájení práce na zakázce přejde do stavu *probíhající*, a v tomto setrvá až do svého dokončení – stav *dokončená*, nebo stornování – *stornovaná*. Byl implementován i stav *nedefinovaný*. Ten je však zaveden pro nestandardní případy a neměl by se využívat.

## 3.4 Přenos dat

Model je přenášen přes protokol XOE ve formě XML dokumentu, ten je specifikován definicí tříd na serveru, nikoli šablonami. Třídy jsou pro JAXB Marshaller anotovány stejně jako samotné XOE. Potřebný objekt je nejprve marshallován do XML pomocí třídy Helpers a poté předán jako data do objektu XoeResponse nebo XoeNotification. Ten data ošetří a znaky, které by narušovaly XML dokument opatří zpětným lomítkem, nebo je přeloží na html entity. Tím je zajištěno, že obsah datového modelu neovlivňuje XOE a je přenášen jako jakýkoli jiný řetězec. Při příjmu dat se z objekt XoeRequest unmasrhalluje a opět přeloží entity na příslušné znaky. Datový řetězec se poté opět unmarshalluje a výstupem je datový objekt příslušného požadavku.

## 3.5 Aktualizace dat

Každá entita je identifikována unikátním celočíselným identifikátorem. Pokud je tedy vkládána entita, která již má již určený identifikátor, aktualizuje se stávající záznam v databázi nově příchozím. Vzhledem k tomu, že identifikátor zůstane stejný, zůstanou zachovány i všechny odkazy na tento objekt. Nově příchozí, či ukládané objekty, identifikátor nemají přiřazen, až do doby kdy jsou vloženy do databáze, neboť přímo ta identifikátory vytváří.

## 3.6 Režie dat

Data se marshallují do XML, což je obdobně jako u samotného XOE. U samotných dat však tvoří režii jen „obalení“ datových hodnot jejich jménem. U některých vlastností tvoří dokonce více než samotný obsah. Například rozměr desky bude tvořit číslo od 1 do 280 (technologické omezení), což jsou 3 znaky pro každý ze dvou rozměrů, celkem tedy 6. Obalující prvek tvoří řetězec „<Rozmer></Rozmer>“. Už nyní je zřejmé, že režie u tohoto prvku je skutečně větší než samotná data. Kompenzovat tento neduh by šlo zkrácením jmen atributů datového modelu, tím by však zcela ztratil svou čitelnost přehlednost.

# 4 Aplikace

Program jako takový je aplikace běžící v konzoli a tudíž bez grafického rozhraní. Vhledem k předpokladu, že poběží na serveru s platformou Linux to lze považovat za pozitivum. Program je po sestavení distribuován jako soubor typu „jar“, což je standardní archiv Javy spustitelný na Java Virtual Machine spolu s potřebnými soubory ve složce data. Ta obsahuje schémata potřebná pro správný chod XOE, certifikáty, konfigurační soubor Hibernate a keystore (klíčenka). Při svém běhu nepotřebuje žádnou obsluhu ani průběžné nastavení. Před první spuštěním by však měla být provedena počáteční konfigurace.

## 4.1 Požadavky

Program potřebuje pro svůj běh a spuštění Java Runtime Environment v aktuální verzi, avšak je doporučena verze alespoň 6.0. Postačuje Standard Edition, protože software si nese všechny potřebné knihovny a třídy s sebou. Aplikace využívá TCP port 6789 a je nutné jej proto uvolnit a umožnit přijímat spojení, například přidáním výjimky do firewallu. Vzhledem k logování událostí je také vhodné, aby měl práva na zápis a přepis do složky „log“. K ukládání dat slouží relační databázový systém MySQL, ve které je vhodné vytvořit oddělenou databázi a uživatele, aby nemohlo dojít ke kolizi s jiným softwarem, pokud by náhodou používal stejná jména relací. V dané databázi je nezbytně nutné vlastnit oprávnění *SELECT*, *INSERT*, *UPDATE* a *DELETE*. Ze strukturních pak *CREATE*, *ALTER*, *INDEX* a *DROP*.

## 4.2 Keystore

Keystore je úložiště v souboru keystore.jks, kde si Java uchovává kryptografická tajemství, jako jsou klíče, certifikáty nebo identity. Aplikace je využívá k uchování certifikátu a privátního klíč k SSL zabezpečení. Běžně se používá sdílená keystore pro celý systém, ale vzhledem k přenositelnosti bylo zvoleno řešení, kdy je klíčenka nesena ve složce data spolu s ostatními důležitými soubory a nenarušuje tak stávající na počítači. Klíčenku není třeba nijak konfigurovat, jen případně vyměnit klíče při jejich vypršení nebo změně údajů. Heslo je nastaveno na „h3sl0PCB“ a je konfigurovatelné bohužel jen přímo ve zdrojovém kódu. Do jisté míry se tím chrání certifikát a klíč, neboť heslo není na první pohled viditelné v konfiguraci, ale komplikuje to případnou změnu hesla. Pro použití se SSL musí být shodné heslo pro certifikát i pro privátní klíč,

jinak SSLSocket vyvolá „java.security.UnrecoverableKeyException: Cannot recover key“. Klíče jsou uchovávány pod unikátními aliasy, v našem případě „importkey“.

Pro práci s keystore je nejjednodušší použít utilitu keytool, která je součástí utilit Javy. Pro změnu hesla do klíčenky slouží příkaz: *keytool -keystore keystore.jks –storepasswd.* Následuje výzva na zadání starého a nového hesla. Heslo ke klíči se mění pomocí: *keytool -keypasswd -alias importkey -keypass stare_hesl*o. Následuje výzva pro zadání nového hesla. Pokud je provedena jedna z těchto změn, měla by být provedena i druhá!

## 4.3 Log událostí

K záznamu událostí do souboru je použita utilita Log4J, kterou nyní vyvíjí Apache Software Foundation. Konfigurace se provádí v souboru log4j.properties v adresáři data. De facto není potřeba nic upravovat. Logger je konfigurován, aby záznam ukládal do maximálně 15 souborů s maximální velikostí 2MB. Soubory se ukládají do složky log, jako log.txt, popř. log.txt.X, kde X je pořadové číslo souboru od 1 do 14. Záznam probíhá v několika úrovních, při běžném provozu je vhodné ponechat nastavení *info*, kdy jsou zapisovány jen důležité události. Při ladění, nebo chybě, je možné přepnout do režimu *debug*, kdy se zapisuje mnohonásobně více informací. Z toho také plyne, že soubory budou narůstat v tomto režimu velmi rychle. V souboru je v komentáři i část, jenž zapne logování do konzole v reálném čase, pro snazší ladění.

Výstup do souboru i do konzole je formátovaný. První na řádku je čas s přesností na tisíciny, následován úrovní záznamu (*INFO*, *WARN*, *FATAL*), dále je zapsána třída, která provedla záznam a na kterém řádku. Za pomlčkou již následuje zpráva jako taková. Každá třída, která využívá logger a generuje tak zprávy jej má definovaný jako *public static final* proměnnou se jménem log. Každý záznam v logu má úroveň podle, která jednoznačně určuje jeho závažnost. Úroveň *info* je využita k uchování informativních záznamů jako je přihlášení uživatele nebo smazání zakázky. Úroveň *debug* slouží k výpisu ladících informací a detailů o databázovém spojení. *Warn* varuje před neočekávaným stavem, který ale ještě není považován za chybu. V našem případě například přijetí deformovaného požadavku. *Error* značí chyby, které jsou považovány za významné, ale lze se z nich ještě zotavit. Jako *fatal* jsou označeny chyby, které zásadním způsobem narušili běh programu. Po jejich vyvolání většinou nelze pokračovat. Příkladem jsou například chybějící konfigurační soubory.

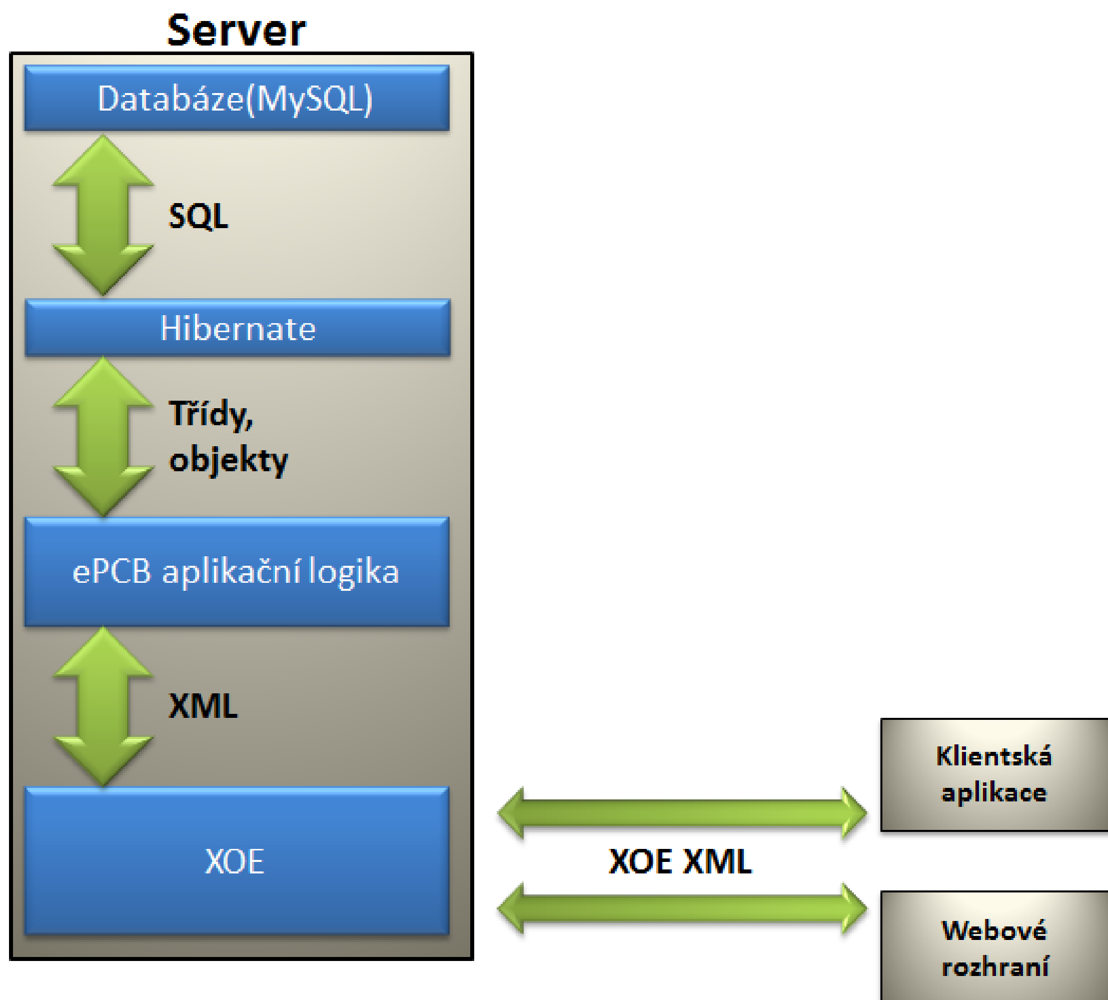
## 4.4 Konfigurace a instalace

Instalace spočívá v nakopírování aplikace do počítače, konfiguraci a spuštění. Koncepce je, vzhledem k univerzálnosti aplikační, tudíž se nejedná o službu. U systému s grafickým rozhraním (Windows, OS X…) stačí většinou pro spuštění poklepat na .jar soubor. Při spouštění z konzole, při nainstalované Javě stačí, například v Linuxu, příkaz *java –jar ePCBLab.jar*. Automatické spouštění na pozadí by však pro průměrně zdatného administrátora neměl být problém zajistit.

Po zkopírování složky s programem je vhodné před prvním spuštěním zkontrolovat a popřípadě upravit nastavení. Soubor konfigurace logu – log4j.properties nepotřebuje nastavení. Pokud by však správce serveru, nebo uživatel vyžadoval umístění logů do jednotného adresáře s ostatními aplikacemi, není problém přenastavit jej. Na unix systémech je to typicky /var/log. Změna umístění záznamů se provádí parametrem *log4j.appender.file.File.* Je také možné upravit velikost souborů se záznamem, případně jejich jméno.

Konfigurace podle použité databáze však vyžaduje konfigurační soubor hibernate.cfg.xml. V závislosti na umístění databázového systému a databáze je nutné nastavit parametr *hibernate.connection.url* podle vzoru:„jdbc:mysql://“ následovaného jménem nebo IP adresou serveru spolu s portem odděleným znakem „:“. Za lomítkem („/“) je nutné uvést jméno databáze. Příslušné jméno a heslo k databázovému systému je potřeba nastavit parametry *hibernate.connection.username* a *hibernate.connection.password*. Podle nastavení serveru je vhodné upravit i parametr *hibernate.c3p0.timeout* na hodnotu o cca 10% menší než je reálné nastavení parametru *wait_timeout* v MySQL.

## 4.5 Struktura aplikace




*Obr. 3 – Znázornění struktury aplikace*

Aplikace je koncipována do vrstev. V nejnižší vrstvě jsou uložena strukturovaná data přímo v databázi, ta spravuje, jak a kde jsou data fyzicky uložena. S databází komunikuje pomocí jazyka strukturovaných dotazů (SQL) framework Hibernate. Ten se stará o transformaci dat z SQL na objekty a obráceně. Také transformuje generická kritéria na SQL dotazy. Aplikační logika pracuje s objekty a předává je jako XML do XOE vrstvy. Ta je obalí a odešle přes socket klientům. Ti mají obdobnou strukturu. Přijmou data, z XOE si vyberou XML a to zpracují na objekty.

## 4.6 Aplikační logika

Hlavní výkonný kód je sdružen v balíčku pcbserver. Vstupním bodem je třída Main s metodou main. Ta jen vytvoří novou instanci třídy Main zavoláním stejnojmenného konstruktoru. Po jeho spuštění je nejprve konfigurováno SSL,

inicializovány společní kolekce socketů a je spuštěno nové vlákno *WatcherThread*. Poté je konfigurován Hibernate Framework načtením příslušného souboru. Hlavní smyčka pak jen naslouchá na portu 6789 a čeká příchozí spojení. Po připojení klienta je vytvořena nová instance vlákna *WorkerThread*, kterému je předán socket s otevřeným spojením s klientem, nová relace Hibernate a jako jméno socketu je nastaveno V + pořadové číslo spojení.

### 4.6.1 WatcherThread

Vlákno, které běží paralelně s hlavní smyčkou. Je odpovědné za rozesílání notifikací a integritu kolekce socketů. Vlákno ve své metodě *run* periodicky vybírá notifikace z FIFO fronty. Pro každou notifikaci zkontroluje celou kolekci, a pokud je vlákno již ukončené nebo klient neposlal delší dobu žádný požadavek, tak jej odstraní ze seznamu. V opačném případě zkontroluje, zdali vlákno (resp. klient) chce přijímat notifikace a zdali není jejím původcem. Pokud jsou tyto podmínky splněny, je přes výstupní proud vlákna odeslaná klientovi notifikace. Toto cyklické opakování probíhá, dokud vlákno není ukončeno z vnějšího kontextu zavoláním metody *endLooping*. Předchozí je nutné provést při zániku hlavního vlákna (vypnutí aplikace), jinak by se vlákno neukončilo. Metoda *addNotification* slouží přidání notifikace do fronty. Metody, které jsou volány z jiných vláken, by měly být definovány jako synchronized, aby nedošlo ke kolizi, nebo uváznutí. Synchronized zajistí, že metoda takto označená nebude ničím přerušena a provede se jako atomický celek.

### 4.6.2 WorkerThread

Každá instance této třídy je spustitelné vlákno. Představuje relaci právě jednoho klienta se serverem. Stará se o příjem zprávy od klienta, její rozdělení a prvotní zpracování. S každou příchozí zprávou se nastaví do proměnné *last_heartbeat* aktuální časové razítko v milisekundách. Metoda *messageHandler* obstarává autorizaci uživatele prvním požadavkem a zavolání tříd a metod pro sestavení odpovědi, kterou potom odešle. Také se stará o předání vygenerované notifikace instanci *WatcherThread* do fronty. Sestavování zprávy probíhá postupným načítáním ze vstupního bufferu, až po načtení znaku 4 v ASCII. Zpráva je poté zproštěna *byte order mark* (značky pořadí bytů a typu UTF řetězce) na začátku a je předána metodě *messageHandler*.

### 4.6.3 Helpers

Třída *Helpers* sdružuje různé užitečné funkce a opakující se kusy kódu. Zjednodušuje a zpřehledňuje tak výsledný text a centralizuje funkční blok na jednom místě, což značně ulehčuje jeho případné změny. Helpers obsahuje 3 statické vnořené třídy pro práci s řetězci (Strings), Zámky (Locks), Marshalling(Marshall) a Konstanty (Konst).

## 4.7 Vývojové prostředí

Pro programování bylo použito vývojové prostředí NetBeans 7.0. Jedná se o open source projekt sponzorován především společností Sun Microsystems. Prostředí je napsané v jazyce Java, pro který je také primárně určeno. Avšak systémem stažitelných pluginů umožňuje rozšířit své pole působnosti na množství jiných jazyků (C/C++, PHP, Ruby…). Pod open source licencí byl produkt uvolněn roku 2000. Jeho vývoj probíhá z převážné části v české pobočce Sun Microsystems v Praze. NetBeans je použitelné pod operačními systémy Windows, Linux, Mac OS X a Solaris.

Umožňuje uživateli snadnou kontrolu kódu, jeho spouštění, uchování historie a profilování. Za zmínku stojí především množství maker, pokud si na ně uživatel zvykne je psaní kódu rychlejší a mnohem komfortnější. To především platí pro stále se opakující bloky kódu, jako jsou iterace přes kolekce, zápis do logu a další. Pro framework swing je dostupný grafický designer, který umožňuje generovat opakující se části kódu a umísťovat prvky na plochu stylem drag&drop, stejně tak, jako měnit jejich vizuální vlastnosti. NetBeans má sofistikovaný systém kontroly kód a upozorňuje tak uživatele například na nepoužité proměnné nebo na nedostupnou třidu u definice proměnné. Pokud však pozná, že třída je dostupná v některém z nainstalovaných balíčků nabídne uživateli import a ten ji tak může importovat jedním kliknutím. Téměř standardem na poli vývojových prostředí je doplňování kódu a jeho nabízení a i toto NetBeans podporuje.

Velmi užitečný je také refactoring kódu, který zásadním způsobem zjednodušuje práci s kódem jako takovým, úkony jako přejmenování proměnné nebo třídy by znamenaly najít všechny výskyty slova a nahradit je, refactoring toto udělá automaticky a bez zásahu uživatele. Přehlednost a správnou strukturu kódu usnadňuje velmi pružně nastavitelné automatické formátování. Jednou klávesovou zkratkou (alt+shift+f na Windows) je tak možné kód srovnat a to do takové míry, že zdrojový kód formátovaný

podle cizích zvyklostí jedním stiskem klávesnice převedeme do formy, na kterou jsme zvyklí a tím si značně zjednodušíme orientaci v textu. Opakující se, „povinné“ části kódu, především pak getter a setter metody je možné automaticky vygenerovat pomocí volby „insert code“. Profiling v NetBeans poskytuje snadný způsob jak zhodnotit aplikaci z hlediska výkonnosti, paměťové náročnosti a sledovat průběh vláken v čase. Lze tak snadno dohledat příčinu uváznutí nebo při pomalém běhu aplikace hledat místa, ve kterých program setrvává nejdelší čas a ty optimalizovat.

# 5 Datové úložiště, Hibernate

Jako datové uložiště byla zvolena volně dostupný databázový systém MySQL. K abstrakci je použit framework Hibernate, který však dokáže se správným JDBC Driverem pracovat s libovolnou databází. Framework abstrahuje práci s databází na práci s objekty a jejich vlastnostmi. V krajním případě je tak možná migrace na zcela odlišný databázový systém, bez nutnosti přepisovat výkonný kód aplikace.

## 5.1 Hibernate

Hibernate je knihovna jazyka Java pro ORM, poskytující objektově orientovaný přístup ke stávajícím relačním databázím. Jeho hlavním posláním je mapovat objekty jazyka Java na databázové tabulky a převádět datové typy Javy do typů dostupných v konkrétním databázovém systému. Hibernate taktéž poskytuje rozhraní pro dotazování a množinové operace nad daty. Při dotazování a práci s množinou dat se tak nepoužívá nativní dotazovací jazyk konkrétního použitého databázového systému, ale abstraktní HQL – Hibernate Query Language nebo Criteria API [4] . Tím jsou zcela odstíněny rozdíly v syntaxi, klíčových slovech a ostatní specifické vlastnosti různých systémů. Programátor tak může používat jen jedno společné API a nemusí se sám starat o transformaci objektů do databázového modelu. Z hlediska programátora tak máme objekty, které si jednoduše udržují svůj stav i po vypnutí a opětovném zapnutí aplikace. Tato výhoda však při neuváženém a nesprávném použití může být velmi draze vyvážena výkonnostními problémy, protože oproti konvenčnímu řešení databáze - Data Access Object je zde jedna abstraktní vrstva navíc.

Hibernate začal vyvíjet v roce 2001 Gavin King jako náhradu za Enterprise JavaBean. Jeho cílem bylo nabídnout lepší schopnost persistence než stávající řešení s jednodušším použitím a chybějícími možnostmi. V roce 2003 byl King najat

společností JBoss na plný úvazek aby pracoval na Hibernate, které získávalo tou dobou čím dál větší pozornost okolí. Roku 2006 byl JBoss převzat firmou Red Hat, ta nadále pokračuje ve vývoji Hibernate.

Hibernate je svobodný software vydávaný pod GNU Lesser General Public License.

## 5.2 Hibernate v praxi

Pro správnou funkci Hibernate je třeba každou třídu, jejíž instance chceme uchovávat, doplnit o anotace, nebo vytvořit mapovací XML. V našem případě jsme použili první možnost. Třídy datového modelu jsou anotovány persistentními anotacemi nad instančními proměnnými. K provádění dotazů nad daty je potom využito Criteria API.

### 5.2.1 Anotace tříd

Hibernate značně zkomfortní práci s datovými objekty, ale třídy je nutné celé revidovat a anotovat, což není vždy úplně triviální úkol. Setkává se tu přístup a pohled ze strany databáze – návrhu struktury a primárních klíčů a pohled ze strany objektů, kdy je vztah 1:N je vyjádřen jako proměnná typu List. Ze začátku to proto může uživateli připadat zmatené a komplikované, ale čas věnovaný studiu se vrátí při používání frameworku.

Entitní třída se označuje anotací @Entity. Volitelným parametrem je name – explicitně zadané jméno. Každá persistovaná entita musí mít jednoznačný identifikátor, Ten anotujeme jako @Id a @GeneratedValue(strategy = GenerationType.AUTO) a hodnoty budou generovány automaticky podle použitého databázového systému.

### 5.2.2 Selekce dat

Výběr dat na základě požadavku klienta se provádí dle zadaných kritérií, které musí všechny objekty bezpodmínečně splňovat. Každý z přenášených objektů má metodu, ve které podle své pattern proměnné vrací instanci třídy *DetachedCriteria*. Jedná se o třídu, která určuje výsledná kritéria dotazu, ale je odpojená, takže při své tvorbě nepotřebuje žádnou Hibernate session. Prvotním kritériem je samotná třída, jejíž instance chceme vybírat. Pak se postupně pomocí pomocí metody *add* přidávají restrikce, podle specifikace jednotlivých pattern tříd. Restrikcí je dostupné velké množství, mezi nejvýznamnější patří „je rovno“, „je menší/větší nebo rovno“

a podobnost „like“. Řazení výstupu není implementováno, neboť si jej řadí klientská aplikace podle požadavku uživatele.

## 5.3 MySQL

MySQL je databázový systém vyvinutý firmou MySQL AB, nyní vlastněný Sun Microsystems, který je dceřinou společností Oracle Corporation. Díky své implementovatelnosti na mnoho různých operačních systémů a faktu, že je zdarma, má na poli používaných databázových systémů velké zastoupení. Funguje jako server poskytující víceuživatelský přístup k více databázím. Jeho dotazování a ovládání probíhá přes SQL – Structured Query Language. Historie MySQL se datuje až do roku 1995 kdy mu Michael Wildenius a David Axmark položili základy. Zpočátku byl koncipován a optimalizován především pro rychlost a proto mu dlouho chyběly vlastnosti pokročilých systémů, jako jsou triggery, uložené procedury a pohledy. Tyto vlastnosti přinesla až verze 5.0 v říjnu roku 2005 (nepočítáme-li alfa, či beta verze).

K jeho použití jsme se přiklonili kvůli jeho snadné instalaci a dostupnosti na běžných serverech platformy Linux, kde se spolu s Apache a PHP stává téměř standardní výbavou.

# 6 Závěr

Cílem této práce bylo nastudovat a vybrat vhodný způsob ukládání a přenosu dat z klientských aplikací na server. Součástí bylo definovat komunikační rozhraní přenosu a výslednou evidenční aplikaci realizovat v jazyce Java. Při řešení vyplulo na povrch několik komplikací způsobených charakteristikou celého problému, avšak všechny se povedlo úspěšně překonat a dovést tak práci do zdárného cíle.

## 6.1 Volba technologií

Prvním úkolem bylo zvolit vhodné technologie pro řešení problém a nenarazit na jejich limity v průběhu práce, to by postup velmi zpomalilo a způsobilo zdržení. Naopak volba příliš komplikované a mohutné technologie by způsobila velkou prodlevu před zahájením práce, neboť studium a bezproblémové zvládnutí potřebuje nemalé množství času a tréninku. Proto jsme nakonec zvolili jednoduchý databázový systém MySQL. Protokol přenosu XOE byl navržen s ohledem na konkrétní potřeby a jeho pochopení a ovládnutí pro nás bylo, jako pro tvůrce, v podstatě okamžité. ORM technologie Hibernate velmi zefektivnila práci, avšak framework je už relativně hodně robustní a komplexní, tudíž byla nastudována jen jeho část pro potřeby této práce. Ač to nemusí být úplně zřejmé, technologie a charakter projektu je velmi různorodý a v komerčním prostředí střední firmy by zaměstnal více odborníků, kde by každý zpracoval oblast, v níž má zkušenosti a hodnotnou praxi.

## 6.2 Dosažené výsledky

Výsledkem práce je serverová aplikace jazyka Java, která umožňuje spravovat a evidovat zakázky laboratoře výroby plošných spojů. Tím, že je koncipována jako server aplikace, může být za splnění některých podmínek dostupná z celého světa. Aplikace je navržena jako víceuživatelská a sama si řeší problematiku zamykání editovaných zakázek a víceuživatelský přístup. Umožňuje asynchronně spravovat klienty o změnách, které vyvolal jiný připojený uživatel. Aplikace využívá pro přenos zabezpečení SSL a k ověření identity serveru je použit certifikát. Společně s autentizací jménem a heslem, lze protokol považovat za bezpečný. K dobru lze přičíst i fakt, že potenciální útočník nezná, alespoň dokud si nepřečte tento text, strukturu dotazů a dat.

## 6.3 Možnost navázání

Projekt, ačkoli se zdá dokončený, má velký potenciál a možnosti rozšíření o další funkcionalitu ať na straně serveru, tak na straně možných klientů. V tomto textu je také zmíněno, že v příštích verzích se počítá se systémem oprávnění a uživatelských rolí. Tato funkcionalita už nebyla v rozsahu této práce, neboť se jedná o komplexní problém. Aplikace by v konečném důsledku dala rozšířit a upravit i pro další laboratoře či dílny na univerzitě a to jak upravit aplikaci pro konkrétní potřeby laboratoře, tak nechat stávající a jen přidat další datové modely pro další evidenční systémy. Tím by teoreticky mohl vzniknout systém, kdy by jeden server sdružoval data několika laboratoří a dílem, ty by mezi sebou mohli sdílet uživatele a zákazníky. Opačnou větví vývoje by bylo zobecnit ještě více XOE a vytvořit dynamický datový model, to by dalo vzniku univerzální aplikace pro evidenci a správu zakázek, která by byla přizpůsobitelná na různé druhy problémů. Podobných scénářů existuje určitě ještě mnohem více, záleží jen, jak se stávající aplikace uchopí a kam se bude ubírat její vývoj, prostor pro rozšíření a navázání další prací určitě existuje.

## Seznam použité literatury

[1] World Wide Web Consortium (W3C) [online]. 1999, Revision: 1.8 1.9.2004 [cit. 2011-05-20]. HTTP/1.1: Status Code Definitions. Dostupné z WWW: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

[2] *World Wide Web Consortium (W3C)* [online]. 16.8.2006, edited in place 29.9.2006 [cit. 2011-05-20]. Extensible Markup Language (XML) 1.0 (Fourth Edition). Dostupné z WWW: <http://www.w3.org/TR/2006/REC-xml-20060816/#charsets>.

[3] *Oracle Technology Network* [online]. 20.4.1999 [cit. 2011-05-20]. Code Conventions for the Java TM Programming Language. Dostupné z WWW: <http://www.oracle.com/technetwork/java/codeconventions-135099.html#367>

[4] MINTER, Dave; LINWOOD, Jeff. *Beginning Hibernate: From Novice to Professional*. New York : Apress, 2006. 317 s. ISBN 1-59059-693-5.