MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Tvorba aplikací v J2EE

DIPLOMOVÁ PRÁCE

Bc. David Maixner

Brno 2004

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Ing. Petr Adámek

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Petrovi Adámkovi za jeho nápady a podněty.

Shrnutí

Práce seznamuje čtenáře s technologií Java 2 Platform Enterprise Edition, a to především s tvorbou komponent Enterprise JavaBeans. K efektivnějšímu použití komponent slouží např. návrhové vzory architektury J2EE, které jsou v práci rovněž popsány. Dalšími uvedenými tématy jsou použité nástroje, jako např. aplikační server JBoss, vývojové prostředí Eclipse, nástroj pro automatickou tvorbu zdrojových kódů XDoclet a další. Jedním z nejdůležitějších prvků při tvorbě ukázkové aplikace bylo rozvržení celkové architektury. K tomu je použit rámec Struts, jehož pomocí lze aplikaci rozvrhnout podle tzv. Modelu 2. Využívá se i moderních metod extrémního programování, jako je testování (testovací rámec Cactus) nebo refaktorování. Všechny principy, postupy a nástroje uvedené v práci byly použity při tvorbě ukázkové aplikace internetového obchodního domu, kterou lze nalézt na doprovodném CD.

Klíčová slova

J2EE, Servlet, JSP, EJB, JNDI, RMI, JMS, JAAS, JTA, Návrhové vzory, Struts, MVC, Ant, JBoss, MySQL, XDoclet, Eclipse, XP, Cactus, Refaktorování, JMeter

Obsah

Kapitola 1

Úvod

Programování vícevrstvých aplikací je rozsáhlou oblastí, ve které se lze setkat s řadou různých technologií, aplikačních serverů, nástrojů a postupů. Jednou z možností je technologie J2EE, která ale svým rozsahem překračuje rámec této práce. Budou zde proto popsány vesměs volně dostupné nástroje, jako jsou JBoss, Ant, XDoclet, Struts, a další.

Cílem práce je popsat obecné možnosti technologie J2EE pro vývoj rozsáhlých vícevrstvých webových aplikací včetně technik, které tento vývoj zefektivňují (refaktorování, návrhové vzory, testování). Také jde o konkrétní ukázku uplatnění J2EE s využitím nejrůznějších nástrojů, které usnadňují nejen implementaci, ale i návrh nebo nasazení aplikace. Většina uvedených produktů a postupů byla demonstrována na tvorbě aplikace jednoduchého internetového obchodního domu.

Protože je téma práce svým zaměřením poměrně obsáhlé, vyžaduje na čtenáři určité počáteční znalosti. Mezi ně patří např. HTML, HTTP, základy XML, UML a základy jazyka Java.

Literatura, která byla při tvorbě práce použita, byla vesměs psána anglicky, a proto bylo nutné většinu názvů překládat. Při překládání názvů se lze setkat se dvěma názory. První říká, že by se mělo dbát na české názvy a prosazovat tak češtinu do dalších oblastí. Druhým názorem je, že pokud se pak český čtenář setká s anglicky psanou literaturou (což je v této oblasti velice pravděpodobné), může pro něj být těžké najít souvislost mezi anglickými a českými názvy, nehledě na to, že každý český autor může názvy přeložit jinak. V této práci byl zvolen kompromis, který je představován zachováním původního anglického názvu v závorce při jeho prvním překladu.

U některých prezentovaných nástrojů se lze setkat s problémem

nedostatečné dokumentace nebo dokonce v jistých případech i špatné funkčnosti. Tyto nedostatky jsou vyvažovány tím, že nástroje jsou volně dostupné. Nejde však jen o výhodu cenovou, nýbrž také o výhodu používání širokou skupinou vývojářů. To vede k rychlejším objevům nedostatků a jejich zveřejnění, na což většina výrobců produktů reaguje poměrně rychle ve formě vydání nové verze. Nové verze také většinou pokrývají časté požadavky vývojářů, čímž se produkt stává oblíbenějším.

Práce stručně popisuje technologii J2EE a její části včetně možností, které poskytují pro tvorbu aplikací. Lze se zde dočíst i o návrhových vzorech pro EJB, z nichž některé byly použity při tvorbě internetového obchodního domu. V dalších částech jsou popsány nástroje, které lze při tvorbě J2EE aplikací použít, včetně principů testování a refaktorování. V kapitole 3 je stručně uveden postup řešení a shrnutí implementace internetového obchodního domu. Více se lze o této aplikaci dozvědět z přílohy A.

Kapitola 2

Seznámení se s použitými prostředky

V této kapitole budou popsány nástroje a postupy, které byly při studiu úkolu a při následné tvorbě aplikace použity. Jedná se vesměs o nástroje podporující technologii J2EE, ve které je internetový obchodní dům vystavěn. Proto zde bude nejprve stručně vysvětleno, z jakých částí se technologie J2EE skládá, a dále, čím se tyto části zabývají.

2.1 J2EE obecně

Co je to J2EE? *J2EE* (*Java 2 Platform Enterprise Edition*) definuje standard pro vývoj rozsáhlých vícevrstvých aplikací. Umožňuje vystavět aplikaci použitím komponent a služeb, které k těmto komponentám nabízí. Zároveň využívá výhod J2SE (Java 2 Platform Standard Edition) jako jsou přenositelnost mezi různými architekturami počítačů, jmenná služba pro přístup k prostředkům uloženým ve stromové struktuře nebo zabezpečení aplikací. J2EE k tomu přidává plnou podporu například pro EJB (Enterprise JavaBeans) komponenty, Java servlety nebo JSP (JavaServer Pages), které budou stručně popsány později. Standard J2EE zahrnuje kompletní specifikace těchto a dalších produktů (viz tabulku 2.1), které se snaží zjednodušit a zobecnit přístup k nemalé množině dodavatelů relačních databází, jmenných služeb nebo např. protokolů použitých pro vzdálená volání metod. Tvůrci aplikací se tedy nemusí specializovat na konkrétní řešení od jednoho dodavatele a obávat se přechodu aplikace na řešení od jiného dodavatele, protože jejich aplikace na něm není závislá.

Další důležitou myšlenkou použitou v J2EE je tvorba aplikací z komponent. Vývojové týmy tvoří různé komponenty, z nichž některé slouží např. pro interakci s uživatelem, jiné zajišťují vlastní

EJB	*Enterprise JavaBeans* – definuje komponenty na straně serveru a jejich kontrakt s aplikačním serverem
HTTP	*HyperText Transfer Protocol* – protokol použitý webovými servery a klienty pro odesílání požadavků a tvorbu odezvy
IIOP	*Internet Inter-ORB Protocol* – protokol technologie CORBA, který se v J2EE používá v kombinaci s RMI, a díky tomu lze nejen přistupovat k externím CORBA objektům, ale i Java komponenty jsou pak přístupné pro nejavovské aplikace
JAAS	*Java Authentication and Authorization Service* – služba zajišťující ověření pravosti a zmocňování
JavaIDL	umožňuje Java aplikacím spolupracovat s externími CORBA objekty přístupnými přes IIOP
JavaMail	poskytuje služby pro práci s elektronickou poštou; závisí na rozhraní *JAF* (*JavaBeans Activation Framework*), čímž se JAF stává také součástí J2EE
JAXP	*Java API for XML Parsing* – slouží pro práci s XML dokumenty a pro jejich XSL transformace
JCA	*Java Connector Architecture* – rozhraní SPI pro připojení adaptérů zpřístupňujících jiné existující systémy
JDBC	*Java DataBase Connectivity* – rozhraní umožňující jednotný přístup k různým relačním databázím
JMS	*Java Message Service* – mechanismus pro zajištěné posílání a přijímání asynchronních zpráv
JNDI	*Java Naming and Directory Interface* – jednotný způsob práce s jmennými a adresářovými službami
JSP	*JavaServer Pages* – HTML stránky používající jazyk Java pro generování dynamického obsahu
JTA, *JTS*	*Java Transaction API*, *Java Transaction SPI* – specifikace, které definují požadavky na správu transakcí
RMI	*Remote Method Invocation* – protokol pro vzdálené volání metod, v J2EE se používá rozšíření RMI-IIOP
Servlets	Java třídy generující dynamický obsah stránek

Tabulka 2.1: Seznam služeb, které specifikace J2EE vyžaduje

funkci nutnou k naplnění *uživatelského cíle aplikace* (*Business Logic*). Tyto komponenty lze skládat a různě kombinovat, je možné s nimi také obchodovat. Lze takto zakoupit např. sadu komponent pro tvorbu uživatelského fóra.

J2EE umožňuje, aby aplikace byla schopna pracovat v různých prostředích beze změny kódu. J2EE také pro své komponenty nabízí řadu služeb, které jsou zajišťovány automaticky, jako např. správa životního cyklu komponent nebo správa transakcí. Vývojáři se tedy mohou zaměřit na uživatelský cíl aplikace nebo na uživatelské rozhraní a nemusí tvořit často velmi složité podpůrné nástroje a funkce, které jsou navíc pro mnoho aplikací téměř totožné.

Aplikace napsané s použitím J2EE technologie mohou být uplatněny v intranetové síti, v internetu, nebo třeba pomocí mobilních telefonů. Lze toho docílit použitím klientů napsaných pomocí Java appletů, HTML, JSP apod., ale je možné také využít samostatné Java-klienty, kteří mohou k J2EE aplikaci vzdáleně přistupovat.

2.2 Java servlety

Pro tvorbu dynamického obsahu webových stránek je možné použít *Java servlety*. Servlety lze uplatnit i pro flexibilnější interakci s uživatelem nebo pro vykonání různých akcí na straně serveru.

Servlety jsou potomci třídy `HttpServlet`. Nejčastější akcí při tvorbě servletů je překrytí metod `doGet` a `doPost`, které zpracovávají HTTP požadavky GET a POST. Parametry těchto metod jsou požadavek (rozhraní `HttpServletRequest`) a odezva (rozhraní `HttpServletResponse`), ve kterých jsou uloženy např. formulářová data, návratový kód nebo vytvořená výstupní stránka.

Instance servletu je v paměti implicitně pouze jedna (lze to změnit implementací rozhraní `SingleThreadModel`) a pro jednotlivé požadavky jsou zakládána vlákna. Kromě vyšší rychlosti to má výhodu např. v tom, že servlet zůstává v paměti i po dokončení odezvy, a tím je možné sdílet data mezi jednotlivými požadavky i mezi servlety.

Životní cyklus servletu probíhá takto:

1. Inicializační část (volá se jen jednou):
    a) vyvoláním servletu (např. zadáním URL v prohlížeči) se

vytvoří jeho instance v paměti;

b) vyvolání metody `init` – zde dochází k inicializaci.

2. Při každém uživatelském požadavku:

   a) vytvoření nového vlákna;

   b) vyvolání metody `service` – zjišťuje typ požadavku, podle něhož se následně zavolá některá z metod `doGet`, `doPost`, apod.

3. Odstranění instance servletu z paměti (ještě než k odstranění dojde, je vyvolána metoda `destroy`):

   a) v případě dlouhé nečinnosti servletu;

   b) při odstranění administrátorem.

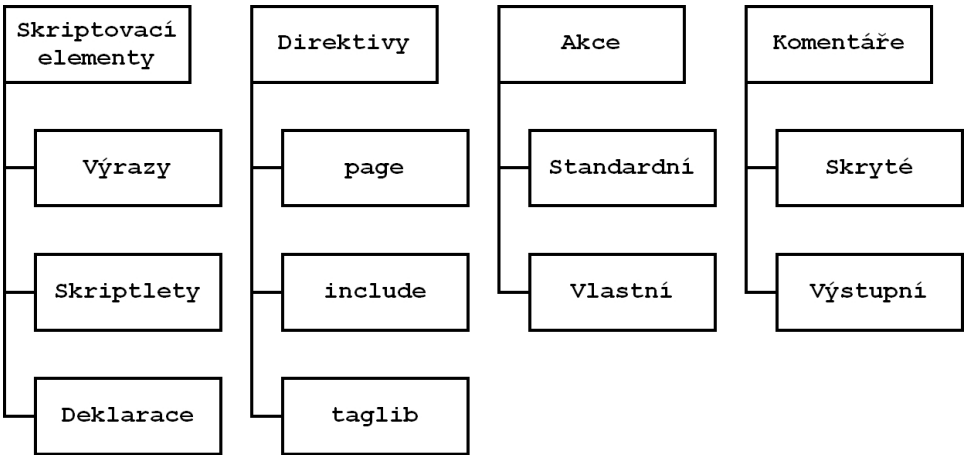
Mezi možnosti, které servlety nabízí, patří např. čtení parametrů a záhlaví z požadavku, nastavování stavových kódů a záhlaví odezvy, sledování sezení, práce s cookies nebo přesměrování požadavků. Pro více informací je možné použít např. [10, 30].

2.3 JavaServer Pages

Servlety sice umožňují vytvořit dynamické stránky, ale při tvorbě výstupu je nutné používat zápis v jazyce Java. To je pro větší množství statického HTML výstupu nepraktické a nepřehledné. Pomocí *JSP* (*JavaServer Pages*) lze část kódu (dynamický obsah stránky) napsat v jazyce Java a další část (statický obsah stránky) pomocí HTML.

Protože budou stránky vytvořené s použitím JSP (dále jen *stránky JSP*) pro další práci důležité, je nutné se o nich zmínit podrobněji. Detailní popis možností JSP poskytuje např. [10, 6, 29]. Při prvním požadavku na stránku JSP dojde k automatickému vytvoření zdrojového kódu servletu, který je svou funkčností ekvivalentem dané stránky. Poté je tento servlet zkompilován a spuštěn. Výstup generovaný servletem je již tvořen prostým HTML kódem a je odezvou na původní požadavek. Tento proces je zdlouhavý a náročný na systémové prostředky. Proto se při dalších požadavcích na stránku JSP používá již zkompilovaná verze servletu.

Zdrojový kód stránky JSP je tvořen pomocí standardních HTML značek a dále pomocí speciálních JSP značek. Tyto speciální značky se používají k oddělení statické a dynamické části stránky a lze je zařadit do struktury podle obrázku 2.1.



Obrázek 2.1: Konstrukce JSP

2.3.1 Skriptovací značky

Skriptovací značky (též *skriptovací elementy*) umožňují přímé vkládání kódu napsaného v jazyce Java do stránek JSP. Tento kód je dále použit při tvorbě servletu, ke které dochází při prvním požadavku na stránku JSP.

Jedním ze tří typů skriptovacích značek jsou *výrazy*. Výrazy reprezentují hodnotu, která se dynamicky vkládá přímo do výstupu generované stránky. Zápis výrazů vypadá takto:

```
<%= výraz v jazyce Java %>
```

Tento výraz se vyhodnotí, převede se na řetězec a vloží se do stránky. K vyhodnocení výrazu dochází při každém vyžádání stránky. Při tvorbě servletu ze stránky JSP se výstup hodnot výrazů umisťuje do metody `_jspService`, která se volá prostřednictvím metody `service`.

Dalším typem skriptovacích značek jsou *skriptlety*, které umožňují vkládání složitějších částí kódu zapsaných v jazyce Java do výsledného servletu. Zápis skriptletu je následující:

```
<% kód v jazyce Java %>
```

Skriptlety mohou vykonávat řadu úkolů, které lze provést v jazyce Java. Tyto úkoly se provádí při každém vyžádání stránky, neboť kód uvedený ve skriptletu se ve vytvořeném servletu umísťuje přímo do metody `_jspService`. Ze stejného důvodu jsou entity (třídy, metody, proměnné), které lze ve skriptletech deklarovat, pouze lokální.

Pro globální deklarace slouží poslední typ skriptovacích značek, kterým jsou *deklarace*. Entity deklarované globálně se vkládají do těla třídy vytvořeného servletu, vně metody `_jspService`. Pokud je servlet vytvořen tak, aby pro každý požadavek bylo založeno nové vlákno, ale instance servletu byla jen jedna, jsou tyto globální entity dostupné ve všech požadavcích. Další možností je servlet, který implementuje rozhraní `SingleThreadModel`, kde pro každý požadavek vzniká samostatná instance. Pro tento typ servletu by bylo možné sdílet deklarované entity použitím klíčového slova `static`. Zápis deklarace v kódu stránky JSP vypadá takto:

```
<%! jedna nebo více deklarací %>
```

2.3.2 Direktivy

K ovlivnění celkové struktury servletu, který vznikne ze stránky JSP, slouží *direktivy*. Zápis direktivy je následující:

```
<%@ direktiva atribut="hodnota" %>
```

Atributů může být více. Specifikace JSP definuje tři direktivy: `page`, `include` a `taglib`.

Ovlivňování vlastností servletu se provádí direktivou `page`. Pomocí atributů této direktivy lze např. nastavovat, které třídy servlet rozšiřuje nebo které knihovny importuje. Tato direktiva se také často používá pro nastavení chybové stránky.

Direktiva `include` slouží ke vkládání souborů do stránek JSP v době, kdy je stránka překládána do servletu. Vkládaný dokument tedy může obsahovat i konstrukce JSP, které se pak vloží na místo direktivy `include` v hlavním dokumentu.

Direktiva `taglib` se používá při rozšiřování množiny dostupných značek o uživatelské značky. Vývojáři mají možnost vytvořit své vlastní značky, které jsou tvořeny třídami v jazyce Java implementujícími speciální rozhraní. Značky se poté sdružují do knihoven značek, které jsou popsány tzv. *popisovači* (*Tag Library Descriptor*, *TLD*).

Při využití takto popsaných značek je nutné v kódu stránky JSP určit, který popisovač se má použít a k tomu právě slouží direktiva `taglib`.

2.3.3 Akce

Akce přidružují funkce ke zvláštním značkám. Stránky JSP se využívají pro zvýšení přehlednosti a k oddělení statické a dynamické části stránek. Přesto někdy dochází k tomu, že směs statických a dynamických prvků je nepřehledná. Akce umožňují vyčlenit funkce do komponent a poté, v kódu stránky JSP, s těmito komponentami pracovat.

Standardní akce používají předponu `jsp` a lze s jejich pomocí např. využívat speciálně navržených tříd *JavaBeans* (viz [10, 28]) nebo vkládat výstup z jiných stránek do aktuální stránky.

U *vlastních akcí* se určuje předpona direktivou `taglib` a slouží k využívání knihoven značek definovaných uživatelem.

2.3.4 Komentáře

Při tvorbě stránek JSP je možné použít dva typy *komentářů*. Jedním z nich je *skrytý komentář*, který slouží jako komentář JSP kódu a není tedy převzat do vytvořeného HTML výstupu. Druhým typem je *výstupní komentář*, což je standardní komentář HTML dokumentu, který se zahrne i do vytvořené stránky.

2.3.5 Implicitní objekty

Pro zpřístupnění některých služeb aplikačního serveru stránkám JSP slouží *implicitní objekty*. Tyto objekty vývojář nemusí nikde inicializovat, protože jejich existenci musí podle specifikace zajistit aplikační server. Některé z implicitních objektů jsou uvedeny v tabulce 2.2.

2.4 Kontejnery

J2EE definuje několik *kontejnerů* včetně JSP kontejneru, servlet kontejneru (souhrnně webový kontejner) a Enterprise JavaBeans kontejneru. Kontejner si lze představit jako část serveru, která poskytuje

`request`	reprezentuje analyzovaný požadavek
`response`	reprezentuje tvořenou odezvu
`session`	slouží pro sledování relace (sezení) klienta
`application`	objekt, který je sdílený pro všechny požadavky na danou aplikaci (lze sem např. ukládat data společná pro všechny servlety aplikace)

Tabulka 2.2: Implicitní objekty

úplné aplikační prostředí, v němž řídí životní cyklus nasazených entit (servletů, stránek JSP, komponent Enterprise JavaBeans) a poskytuje jim různé služby.

2.5 Enterprise JavaBeans

Nejrozsáhlejší část práce se týká technologie *EJB* (*Enterprise JavaBeans*, někdy jen *Enterprise Beans*), která spolu s dalšími součástmi J2EE určuje *rámec* (*Framework*) pro tvorbu distribuovaných aplikací. Komponenty vytvořené s pomocí EJB se používají k zapouzdření aplikační logiky vyvíjené aplikace. Specifikace EJB verze 2.0 rozlišuje tři typy komponent, které budou později popsány podrobněji spolu se základními postupy, s nimiž se lze při tvorbě těchto komponent setkat. Detailní popis možností EJB poskytuje např. [22, 26, 17]. Stručné shrnutí lze nalézt např. v [21].

2.5.1 Motivace pro EJB

Při přechodu od monolitických k distribuovaným aplikacím, kde jsou jednotlivé vrstvy od sebe oddělené a kde je spousta klientů přistupujících k několika serverům přes síť, je nutné vzít v úvahu několik důsledků. Některé z nich jsou shrnuty v tabulce 2.3.

Součástí aplikačního serveru je tzv. *prostředník* (*Middleware*), který poskytuje služby nasazeným aplikacím. Služby slouží pro řešení uvedených důsledků. Dále jsou tyto služby součástí požadavků specifikace EJB na EJB kontejner, tudíž komponenty napsané pro jeden EJB kontejner lze snáze přenést i na jiný EJB kontejner. Dochází i k urych-

vzdálené volání metod	potřeba spojení mezi klientem a serverem a odesílání požadavků na volání metod
rozložení výkonu (Load Balancing)	klienti musí být směrováni na server s nejmenší zátěží
transakce	pro několik klientských operací zajišťují vlastnosti nerozdělitelnosti, neporušenosti, odloučení a trvalosti
vlákna	server by měl reagovat na požadavky klientů souběžně
sdílení zdrojů (Resource Pooling)	některé zdroje mohou být po použití jedním klientem vráceny kontejneru a poté použity jiným klientem
bezpečnost	po ověření pravosti identity uživatele mu jsou povoleny pouze ty operace, na které má oprávnění

Tabulka 2.3: Důsledky přechodu k distribuovaným aplikacím

lení vývoje, protože služby nejsou součástí samotných aplikací, nýbrž jsou poskytovány prostředníkem. Prostředník může být buď explicitní nebo implicitní a liší se v používání. Při použití *explicitního prostředníka* musí vývojář psát do svého kódu volání jednotlivých API, které prostředník poskytuje (např. API pro správu transakcí). Tím dochází k míchání kódu uživatelského cíle aplikace s kódem obslužným. *Implicitní prostředník* (použitý např. v EJB) se používá deklarativně. To znamená, že vývojář má v kódu svojí komponenty pouze kód, který zajišťuje uživatelský cíl aplikace a veškeré obslužné funkce jsou zajištěny automaticky prostředníkem podle deklarovaného popisu umístěného v popisujícím souboru.

2.5.2 Technické řešení EJB

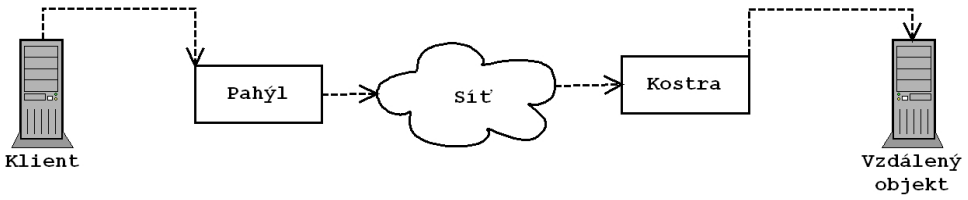
K porozumění technologii EJB do značné míry pomáhá pochopení toho, jak EJB pracuje. A k tomu je nutné porozumět několika dalším produktům.

RMI-IIOP

Jedním z nich je *RMI-IIOP* (*Remote Method Invocation over Internet Inter-ORB Protocol*), což je mechanismus pro zajištění síťové komunikace mezi distribuovanými objekty. RMI-IIOP používá oddělení rozhraní a vlastní implementace. Rozhraní definuje metody, které objekt zveřejňuje. Implementace je pak vlastní objekt, ve kterém jsou naprogramovány požadované funkce objektu. To umožňuje změnu kódu bez změny rozhraní, a tudíž i bez změny klienta. *Vzdálený objekt* je speciální objekt, který je vystaven na vzdáleném serveru. V EJB architektuře tvoří vzdálené objekty důležitý základ EJB komponent.

Při použití RMI-IIOP není možné volat vzdáleně metody na implementaci objektu, je nutné použít rozhraní. Tomuto rozhraní se proto říká *vzdálené rozhraní* nebo též (v EJB častěji používaný název) *rozhraní komponenty* (*Remote Interface*). Vzdálené rozhraní obsahuje hlavičku každé metody, kterou objekt zveřejňuje pro vzdálené volání. Klient, který má přístupné pouze vzdálené rozhraní, poté může vzdáleně volat metody objektu (pomocí vzdáleného rozhraní), jakoby šlo o lokální objekt. RMI-IIOP si ve skutečnosti takový lokální objekt u klienta udržuje a tento lokální objekt implementuje vzdálené rozhraní, aby obsahoval všechny metody zveřejněné vzdáleným objektem. Lokálnímu objektu se říká *pahýl* (*Stub*) a je odpovědný za lokální přijetí vzdáleného volání a jeho delegování aktuální vzdálené implementaci objektu. Pro klienta je toto delegování skryto.

Stejně jako klient vyvolává metody na pahýlu, který je vzhledem ke klientovi lokální, je potřeba, aby i vzdálený objekt přijímal volání svých metod od nějakého objektu, který je lokální vzhledem k němu. Tomuto objektu se říká *kostra* (*Skeleton*). Kostry jsou odpovědné za přijetí vzdáleného volání a jeho doručení implementaci vzdáleného objektu. Pahýl i kostra jsou vyobrazeny na obrázku 2.2.



Obrázek 2.2: Pahýl a kostra

Jednou z nejdůležitějších odpovědností pahýlů a koster je vypořádání se s předáváním parametrů. Důležitým rozdílem oproti standardnímu programování v jazyce Java je, že RMI-IIOP při vzdáleném volání metod předává všechny parametry hodnotou. Tedy všechny objekty předávané jako parametry jsou přenášeny mezi klientem a vzdáleným objektem. Oproti tomu při předání objektu jako parametru volané metody v Javě, se tento objekt předává odkazem. Data objektu tedy nejsou kopírována, ale je použit pouze odkaz na objekt. Tento fakt může způsobit komplikace při používání vzdálených klientů v EJB. Pokud totiž metodě vzdáleného objektu klient předá jako parametr objekt, který bude touto metodou změněn, změna se u klienta neprojeví. Projeví se pouze v lokální kopii parametru (objektu), která vznikla přenesením od klienta. Další potíží s předáváním hodnotou je, že pokud se jako parametr předává objekt, který zahrnuje ve svých instančních proměnných spoustu dalších objektů, přenáší se sítí velké množství dat, a to může být časově náročné. Pro předávání objektů hodnotou při vzdáleném volání metod se používá princip *serializace*, známý z jazyka Java (viz např. [11, 22, 23]).

Aby bylo možné využít výhod volání odkazem, nabízí RMI-IIOP *simulaci volání odkazem*. Objekt, který má klient předat jako parametr vzdáleně volané metodě, je implementovaný jako vzdálený objekt. Má tedy svoje vzdálené rozhraní, které se používá pro volání metod, a je vystaven na vzdáleném serveru. Dále je k němu vytvořen pahýl a kostra. Při předávání parametru zajistí RMI-IIOP, že se předá pouze pahýl k předávanému objektu. Vzdáleně volaná metoda poté přistupuje k objektu pomocí pahýlu. Existuje tedy pouze jediná kopie objektu. Pokud dojde ke změně objektu ve vzdálené metodě, projeví se změna i u klienta. Není to pravé volání odkazem, protože se nepředává odkaz, ale pahýl objektu, který se předává hodnotou (tedy serializuje se).

Pro další informace o RMI-IIOP je možné použít např. [32].

JNDI

Dalším konceptem klíčovým pro pochopení architektury EJB je *JNDI* (*Java Naming and Directory Interface*). JNDI se v EJB používá pro umisťování a následné vyhledávání entit (objektů, strojů, služeb, proměnných prostředí apod.). JNDI umožňuje vystavit komponentu EJB na

serveru a také umožňuje klientům vystavenou komponentu najít. Jde o poměrně rozsáhlý nástroj a pro pochopení EJB bude stačit jen zjednodušený princip. Detailní popis JNDI lze nalézt např. v [31].

Jak název napovídá, jde o jmennou a adresářovou službu. *Jmenná služba* zajišťuje dva úkoly. Prvním z nich je svazování jmen s entitami tzv. *vazbami*. Druhým úkolem je umožnit vyhledání entity podle zadaného jména. *Adresářová služba* je rozšíření jmenné služby o možnost manipulovat s atributy, které přísluší jednotlivým entitám.

Důležitým pojmem, který se při použití JNDI v EJB často vyskytuje, je *kontext*. Je to objekt, který obsahuje nula a více vazeb. JNDI ukládá entity do stromové struktury, která je tvořena *podkontexty*. Ty jsou opět plnohodnotnými kontexty.

Protože kontextů může vedle sebe existovat více, je nutné zajistit hledání v tom správném stromu, majícím za kořen tzv. *počáteční kontext* (*Initial Context*). Počáteční kontext je potřeba nastavit ještě před jakýmkoliv hledáním.

Problém se jmennými a adresářovými službami je, že těchto produktů je dnes hodně. Ke každému z nich se přistupuje jinak, což znamená, že není snadné změnit dodavatele služby. JNDI tento problém řeší podobným způsobem jako např. JDBC řeší problém s velkým počtem dodavatelů databází. Vývojář používá jediné rozhraní JNDI (*Application Programming Interface*, *API*). JNDI poté přistupuje k rozhraní dané jmenné a adresářové služby (pomocí *Service Provider Interface*, *SPI*). Další výhodou JNDI může být např. to, že je možné současně používat služby od více dodavatelů.

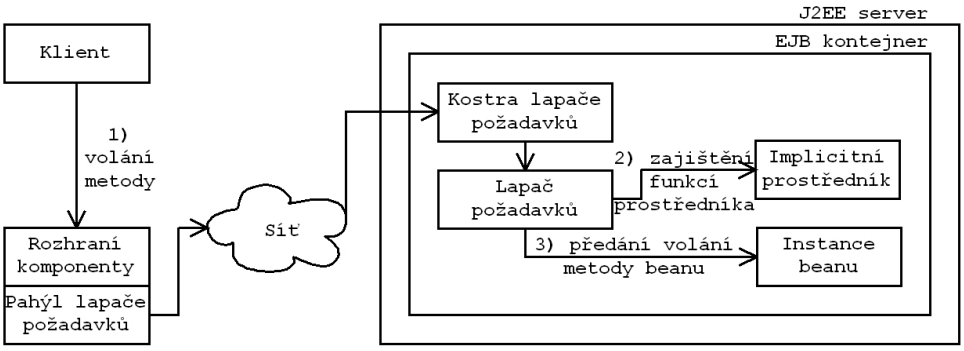
2.5.3 Složení EJB komponenty

Architektura J2EE umožňuje klientům pracovat se vzdálenou aplikací. Klientem může být Java program nebo např. webová vrstva tvořená servlety a stránkami JSP. Vzdálená aplikace je složena z komponent, které tvoří logicky ucelené části. Těmito komponentami jsou Enterprise JavaBeans.

EJB komponenty (*beany*) jsou fyzicky tvořeny třídou beanu, rozhraním komponenty a třídním rozhraním. Jakmile existuje instance beanu v EJB kontejneru, klient má možnost s touto instancí pracovat. Avšak nemůže k ní přistupovat přímo. Kdyby to šlo, server by neměl možnost uplatnit funkce implicitního prostředníka.

Např. je v aplikaci deklarováno, že má implicitní prostředník zajistit, aby operace s beanem probíhaly v transakcích řízených kontejnerem. Ale pokud by klient přistupoval k beanu přímo, kontejner by nemohl začít řídit transakce, protože by nerozpoznal přístup klienta k beanu.

Architektura práce s komponentou je tedy navržena jinak. Je nutné, aby mezi beanem a klientem byl ještě nějaký objekt, který zachytí volání metody od klienta, uplatní deklarované funkce implicitního prostředníka a teprve poté zavolá odpovídající metody instance beanu. Objekt, který je oním mezičlánkem, se nazývá *lapač požadavků* (*Request Interceptor*, jinak též *EJB Object*). Lapač požadavků je vygenerován automaticky kontejnerem podle rozhraní komponenty, které vytváří vývojář. Rozhraní komponenty obsahuje ty metody, jež zveřejňuje třída beanu pro klienta. Tedy i lapač požadavků musí tyto metody implementovat, a to tak, aby nejdříve zajistil volání deklarovaných funkcí implicitního prostředníka a poté volání odpovídající metody instance beanu. EJB komponenta není vzdáleným objektem, tak jak byl definován v RMI-IIOP. Klient k ní tedy nemůže přistupovat jinak než s využitím lapače požadavků, který již vzdáleným objektem podle požadavků RMI-IIOP je. Tedy rozhraní komponenty vytvořené vývojářem patří k lapači požadavků, přesněji řečeno, umožňuje klientům přistupovat k lapači požadavků jako ke vzdálenému objektu. Rozhraní komponenty musí rozšiřovat rozhraní `javax.ejb.EJBObject`. U klienta existuje i pahýl lapače požadavků tak, jak bylo uvedeno v části 2.5.2 vysvětlující RMI-IIOP. Celá architektura je zobrazena na obrázku 2.3.



Obrázek 2.3: Architektura EJB s lapačem požadavků

Kromě rozhraní komponenty musí vývojář EJB komponenty vytvořit *rozhraní třídy* (*třídní rozhraní*, *Home Interface*). Lapač požadavků je objekt, který je spravován kontejnerem. Klient nemůže přímo zakládat instance tohoto objektu, protože jde o vzdálený objekt. Ale aby klient mohl přistupovat ke třídě beanu pomocí lapače požadavků, je nutné, aby měl na instanci lapače požadavků odkaz. K získání odkazu používá klient tzv. *továrnu na lapače požadavků* (*EJB Object Factory*). Tato továrna je odpovědná za zakládání a rušení instancí lapačů požadavků. Továrna též nese název *třídní objekt* (*Home Object*).

Třídní objekty jsou stejně jako lapače požadavků závislé na implementaci EJB kontejneru a jsou vytvářeny automaticky podle třídního rozhraní. Třídní rozhraní obsahuje hlavičky metod pro zakládání, rušení, případně vyhledávání (pro entitní beany, viz 2.5.7) instancí lapačů požadavků. Musí rozšiřovat rozhraní `javax.ejb.EJBHome`.

Třídní objekt je také vzdálený objekt podle RMI-IIOP požadavků, ale na rozdíl od lapače požadavků není instance třídního objektu klientem zakládána, nýbrž je nalezena již existující instance pomocí JNDI (viz 2.5.2). Instance třídního objektu bývá pro každý lapač požadavků jediná (je to však záležitostí vnitřní implementace kontejneru, která se mezi dodavateli může lišit) a je spravována kontejnerem. Všichni klienti používají nalezenou instanci pro zakládání, rušení a případné vyhledávání svých vlastních instancí lapačů požadavků. Zatímco tedy každý klient potřebuje mít vlastní instanci komponenty EJB (přístupnou pomocí instance lapače požadavků, která tudíž musí být také pro každého klienta vlastní), instance třídního objektu může být mezi klienty sdílena.

Důležitým konceptem, nově uvedeným ve specifikaci EJB verze 2.0, jsou tzv. *místní rozhraní* (*Local Interfaces*). Problém s třídním objektem je, že zakládání lapačů požadavků přes tento objekt je pomalé. Stejně tak i volání metod přes lapač požadavků je pomalé. Souvisí to se vzdáleným voláním metod. Všechny parametry musí pahýl nejdříve připravit pro cestu sítí, poté jsou poslány kostře, ta je vrátí opět do původního tvaru a pak teprve lze volat metody vzdáleného objektu.

Klient ale nemusí být vždy vzdálený. Může jít o klienta, který je na stejném serveru. Klientem mohou být např. jiné EJB komponenty, které jsou nasazené do stejného EJB kontejneru. Tyto komponenty jsou vzhledem k sobě samým lokální, a tudíž vzdálené

volání metod není potřebné. Zůstává ale potřeba používat deklarovaného implicitního prostředníka, tedy i potřeba používat lapač požadavků, což vede i k nutnosti používání třídního objektu. Proto existuje *místní rozhraní komponenty* (*Local Interface*, rozšiřuje rozhraní `javax.ejb.EJBLocalObject`) a také *místní rozhraní třídy* (*Local Home Interface*, rozšiřuje rozhraní `javax.ejb.EJBLocalHome`). Tato místní rozhraní pracují s objektem lapače požadavků a s třídním objektem lokálně, bez vzdáleného volání. Důležitou změnou oproti rozhraní komponenty a třídnímu rozhraní je, že jejich místní verze předávají parametry odkazem (čili tak, jak je to v Javě obvyklé), protože nepotřebují používat vzdálené volání metod. Použití místních rozhraní nijak nevylučuje použití vzdálených verzí. Do komponenty je možné zahrnout oba typy. Ve vzdálených klientech pak lze používat vzdálená rozhraní a v místních klientech místní rozhraní.

Jak bylo uvedeno výše, služby implicitního prostředníka se využívají deklarováním jejich potřeby. Vývojář komponenty si určí, které služby chce mít prostředníkem zajištěny, a jejich konfiguraci deklaruje ve zvláštním souboru nazvaném *popisovač nasazení* (*Deployment Descriptor*). Popisovač nasazení je součástí komponenty a slouží pro deklaraci služeb, jako jsou transakce, trvalost (persistence), kontrola životního cyklu či bezpečnost. Kontejner poté zajistí podle popisovače nasazení požadované služby.

Některé kontejnery nabízí význačné rysy, které jiné kontejnery buďto nemají nebo je řeší jiným způsobem. Specifikace EJB dává v tomto ohledu značnou volnost, takže záleží čistě na dodavateli kontejneru, jakým způsobem implementuje požadované služby. Většina kontejnerů tedy vyžaduje ještě jeden popisovač, ve kterém jsou deklarovány specifické rysy pro určitý kontejner. I tento popisovač je součástí komponenty a předurčuje její použití pro daný kontejner. V případě potřeby přenositelnosti mezi více kontejnery se do komponenty zařadí popisovače pro různé kontejnery.

Poslední součástí, ze které je tvořena EJB komponenta, je vlastní třída beanu. V této třídě jsou implementovány všechny metody pro splnění uživatelského cíle aplikace, které jsou volány prostřednictvím lapače požadavků. Kromě těchto metod jsou zde implementovány i tzv. *metody zpětného volání* (*Callback Methods*). Tyto metody volá kontejner při určitých událostech. Podrobněji jsou metody zpětného volání vysvětleny pro každý typ EJB komponent zvlášť v sekcích

2.5.6, 2.5.7 a 2.5.8. Životní cyklus instancí jednotlivých typů beanů souvisí s možností sdílení instancí (viz 2.5.4). Obecně lze říci, že instance beanu jsou spravovány kontejnerem, klient spravuje pouze instance lapačů požadavků.

Komponenta je tedy tvořena třídou beanu, třídním rozhraním, rozhraním komponenty (možné jsou i místní varianty), popisovačem nasazení a popisovači závislými na dodavateli kontejneru. Každá z těchto částí musí být vytvořena vývojářem.

2.5.4 Sdílení instancí beanů

EJB kontejner zajišťuje na pozadí funkci nazvanou *sdílení* (*Pooling*). Tato funkce umožňuje efektivněji využívat systémové prostředky.

Jako příklad poslouží situace, kdy klient prochází internetovým obchodem a vybírá zboží z katalogu. V prodlevách, kdy klient např. čte informace o nabízených produktech, je instance komponenty zajišťující aplikační logiku na straně serveru nevyužita. Proto je možné tuto instanci přidělit jinému klientovi, který prochází katalog zboží.

Podobně je možné sdílet např. databázová připojení nebo i komponenty, jejichž stav je závislý na klientovi. Stav, který je pro daného klienta specifický (např. obsah košíku), je odložen na disk a do komponenty je vložen stav příslušející jinému klientovi (toto se děje v případě, kdy není možné získat od kontejneru další volnou instanci). Jakmile první klient opět komponentu potřebuje, může mu být přiřazena volná instance (v případě, že je již k dispozici, jinak to bude opět instance získaná odložením stavu z jiné instance na disk), do které je z disku přesunut dříve odložený stav.

Instance komponent jsou spravovány kontejnerem. Také jejich přidělování klientům zajišťuje kontejner. V případě, že není volná instance, kontejner založí novou (maximální počet instancí lze obvykle nastavit v popisovači závislém na dodavateli kontejneru). V případě, že je volných instancí příliš mnoho, kontejner nějaké uvolní. Toto chování se nazývá *sdílení instancí* (*Instance Pooling*).

2.5.5 Přístup k funkcím kontejneru

EJB komponenty aplikace někdy potřebují přistupovat ke kontejneru, ve kterém jsou nasazeny, a zjišťovat jeho stav. Např. když

chce EJB komponenta zjistit bezpečnostní pověření klienta, který přistupuje k jejím metodám. Proto kontejner poskytuje beanům objekt, jehož pomocí je prostředí kontejneru beanům dostupné. Tento objekt se nazývá *EJB kontext* (*EJB Context*) a je popsán rozhraním `javax.ejb.EJBContext`. Podle typu beanu se využívají potomci tohoto rozhraní uvedení v sekcích 2.5.6, 2.5.7 a 2.5.8, kteří přidávají další funkce kontejneru závislé na typu beanu.

Následuje popis tří typů EJB komponent podle specifikace EJB verze 2.0.

2.5.6 Beany sezení

Pro modelování uživatelského cíle aplikace a dalšího procesního chování slouží *beany sezení* (*Session Beans*). Lze si je představit jako slovesa, protože jsou akcemi, které aplikace vykonává. Třída beanu sezení implementuje rozhraní `javax.ejb.SessionBean`, čímž získává povinnost implementovat několik metod zpětného volání. Jednou z nich je `setSessionContext`, která umožňuje beanu použít přístup k prostředí, ve kterém je komponenta nasazena (viz 2.5.5). Parametr metody je typu `javax.ejb.SessionContext`. Toto rozhraní přidává funkce pro získání odkazu na lapač požadavků, který lze poté předávat jako parametr při volání metod. Odkaz na lapač požadavků v tomto případě zastupuje klíčové slovo `this`. Nelze totiž předávat přímo odkaz na instanci beanu (užitím `this`), protože s instancí se pracuje prostřednictvím lapače požadavků (viz 2.5.3).

Existují dva podtypy beanů sezení. *Stavový bean sezení* se používá v případech, kdy je nutné sledovat stav klienta v průběhu času. Např. v elektronickém obchodě to může být košík, do kterého klient přidává zboží. Stavový bean sezení je navržen pro obsluhu procesů, které se dělí do více volání metod nebo do více transakcí. Při změně stavu v jednom volání metody je změněný stav přenesen k dalšímu volání metody. Rozlišení podtypu beanu sezení se provádí v popisovači nasazení.

Při sdílení instancí (viz 2.5.4) stavových beanů sezení je nutné odkládat stav na disk. Této akci se říká *pasivace* (*Passivation*). Přesun stavu zpět do instance se nazývá *aktivace* (*Activation*). Tyto dvě akce jsou beanu oznámeny vyvoláním příslušných metod zpětného volání `ejbPassivate` a `ejbActivate`.

Další metodou vyžadovanou při tvorbě stavového beanu sezení je `ejbCreate...` Tato metoda odpovídá hlavičce uvedené v třídním rozhraní pod názvem `create...` s tím rozdílem, že zatímco metoda `create...` vrací typ rozhraní komponenty, metoda `ejbCreate...` vrací `void`. Metod může být více, s různými parametry i s různými názvy, které si však musí odpovídat.

Některé procesy lze uskutečnit pomocí jediného požadavku. Není tedy nutné přenášet stav mezi jednotlivými voláními metod. Pro tento typ konverzace mezi klientem a beanem se používá *bezstavový bean sezení*. Po každém volání metody může instance beanu ztratit svůj stav konverzace. Protože si instance tohoto typu komponent nepotřebují uchovávat prvky specifické pro jednotlivé klienty, je možné, že budou kontejnerem předány jinému klientovi. Zda se tak opravdu děje, závisí na implementaci kontejneru. Instance jsou sdružovány do skupiny, ze které se v případě potřeby jedna instance vybere a po použití se tam opět vrací. Bezstavový bean si může udržovat stav, který není závislý na klientovi (např. připojení do databáze).

Příkladem bezstavového beanu sezení je komponenta pro komprimování dat. Obsahuje metodu pro kompresi, která má jako vstupní parametr nekomprimovaná data a výstupem jsou komprimovaná data. Není nutné si mezi jednotlivými voláními metody pamatovat jakýkoliv stav.

Pro bezstavové beany sezení je nutné mít pouze jednu metodu `ejbCreate`, která nemá žádné parametry.

Stavové i bezstavové beany sezení vyžadují metodu zpětného volání `ejbRemove`, která je volána kontejnerem těsně před zrušením instance beanu.

Klientem komponenty, která využívá RMI-IIOP, může být např. vzdálený program v jazyce Java, Java applet, servlet nebo jiná komponenta. Implementace jednotlivých klientů jsou si ale podobné. Nejdříve je nutné získat počáteční kontext JNDI (viz 2.5.2). Poté se vyhledá třídní objekt s použitím JNDI. Pomocí třídního objektu se klientovi založí lapač požadavků (může se založit nová instance beanu nebo se využije některá ze stávajících – viz 2.5.4). K tomu se volá metoda třídního rozhraní `create`. Ta nakonec deleguje volání až do instance beanu k odpovídající metodě `ejbCreate`. Poté jsou klientem volány metody lapače požadavků, které zajišťují uživatelský cíl aplikace tím, že jsou delegovány instanci beanu. Nakonec je lapač

požadavků odstraněn.

2.5.7 Entitní beany

Entitní beany (*Entity Beans*) slouží pro modelování dat. Lze si je představit jako podstatná jména, protože jsou to datové objekty. Entitní bean reprezentuje data z databáze. Jedna instance beanu odpovídá jednomu řádku tabulky z relační databáze. Při založení beanu vznikne v tabulce nový řádek, při zrušení beanu se řádek z tabulky vymaže. Třída entitního beanu musí implementovat rozhraní `javax.ejb.EntityBean`. Musí tedy implementovat metody `ejbActivate` a `ejbPassivate`, které mají stejnou funkci jako u beanů sezení (viz 2.5.6). Další nutnou metodou je `setEntityContext`, která beanu umožňuje přístup k prostředí kontejneru pomocí rozhraní `javax.ejb.EntityContext`.

Beany sezení často využívají entitní beany k provedení akcí, které představují. Důležitým rozdílem mezi beany sezení a entitními beany je délka jejich existence. Zatímco bean sezení je čistě paměťový objekt, entitní bean je objekt reprezentující data uložená na disku (v databázi), proto může existovat i po havárii serveru. Doba existence beanu sezení by měla být přibližně stejně dlouhá jako sezení, ve kterém pracuje klient, zatímco doba existence entitního beanu může být podstatně delší.

Entitní beany jsou objekty, které se umí synchronizovat s obsahem databáze. Nemusí jít jen o relační databázi, ale např. o objektovou databázi nebo lze použít i princip serializace známý z jazyka Java. Komponentu entitního beanu někdy tvoří, kromě dříve uvedených částí (viz 2.5.3), třída *primárního klíče*. V této třídě se určuje, které proměnné entitního beanu tvoří primární klíč. Třída primárního klíče není povinná, používá se pouze v případě, kdy je klíč tvořen více než jednou proměnnou.

Pokud chce více klientů přistupovat ke stejným datům, musí existovat více instancí entitního beanu. Aby v tomto případě nedošlo k poškození dat navzájem mezi klienty, používají se transakce (viz 2.5.10).

Existují dva podtypy entitních beanů. Liší se od sebe podle způsobu, kterým dochází k synchronizaci obsahu entitního beanu a obsahu databáze. První způsob se nazývá *trvalost spravovaná beanem*

(*Bean-Managed Persistence*, *BMP*). Při tomto způsobu provádí synchronizaci obsahů beanu a databáze bean pomocí metod zpětného volání `ejbStore` a `ejbLoad`. Tyto metody volá kontejner, jakmile usoudí, že je nutné synchronizovat obsahy beanu a databáze. Vývojář pak musí sám zajistit uložení dat do databáze v metodě `ejbStore` a načtení dat z databáze do beanu v metodě `ejbLoad`. Stejně tak musí zajistit vytvoření řádku v tabulce při založení beanu v metodě `ejbCreate` a zrušení řádku tabulky při rušení beanu v metodě `ejbRemove`. Při tomto přístupu se často používá *JDBC* (*Java Database Connectivity*), o kterém se lze více dozvědět např. v [23, 10].

Druhým způsobem synchronizace je *trvalost spravovaná kontejnerem* (*Container-Managed Persistence*, *CMP*). Při tomto způsobu provádí synchronizaci obsahů beanu a databáze kontejner. Nastavení této činnosti se provádí v popisovači nasazení a také v popisovači závislém na dodavateli. Kontejner poté podle nastavení případně založí potřebné tabulky a zajišťuje synchronizaci obsahu databáze s obsahy instancí entitních beanů. S využitím tohoto deklarativního ukládání dat lze poté přenášet komponenty mezi databázemi bez změny kódu, protože přístup k databázi zcela zajišťuje kontejner. Třída CMP entitního beanu je abstraktní a kontejner od této třídy automaticky vytvoří potomka, v němž je obsažen všechen potřebný JDBC kód. V potomkovi jsou rovněž vytvořeny proměnné, které reprezentují jednotlivé položky řádku v databázové tabulce. Jsou vytvořeny podle tzv. *abstraktního schématu trvalosti* (*Abstract Persistence Schema*) deklarovaného v popisovači nasazení.

Třídní objekt plní pro entitní beany kromě zakládání instancí a jejich rušení ještě další funkci. Továrna na lapače požadavků se používá ještě k vyhledávání existujících instancí entitních beanů. Protože beany odpovídají řádkům v databázové tabulce, mohou být, stejně jako řádky tabulky, nalezeny. Entitní beany lze hledat pomocí vyhledávacích metod, které mohou mít různé parametry a různá jména (začínající `find...` v třídním rozhraní a `ejbFind...` ve třídě beanu). Sémantika vyhledávacích metod je v případě BMP entitních beanů naprogramována vývojářem s použitím JDBC. Pro CMP entitní beany je sémantika deklarována v popisovači nasazení pomocí jazyka *EJB-QL* (*EJB Query Language*) podobného SQL. Z této definice se poté vytváří JDBC kód pro hledání. Hlavičky vyhledávacích metod jsou umístěny i v třídním rozhraní, kde se využívají klientem pro nalezení

instancí entitních beanů.

Na rozdíl od beanů sezení nemusí mít entitní beany metodu pro zakládání instancí (tedy `ejbCreate`). Může to být užitečné v případě, kdy klient nemůže zakládat nová databázová data. Data mohou být do tabulky vložena jinými způsoby (např. jinou aplikací). Instance entitních beanů lze poté vyhledávat. Pokud třída entitního beanu použije metodu `ejbCreate...`, musí pro každou takovou metodu definovat i metodu `ejbPostCreate...`, která má stejné parametry a je volána kontejnerem po `ejbCreate...`.

2.5.8 Beany řízené zprávami

Novinkou ve specifikaci EJB verze 2.0 je poslední typ EJB komponent, a to *beany řízené zprávami* (*Message-Driven Beans*). Svým účelem jsou podobné beanům sezení, avšak s tím rozdílem, že s beany řízenými zprávami lze pracovat pouze pomocí zasílání zpráv.

Klíčovým prvkem při zasílání zpráv je tzv. *zprostředkovatel* (*Middleman*), který leží mezi klientem a serverem, přijímá zprávy od jednoho nebo více *producentů* a zasílá je jednomu nebo více *konzumentům*. Zprostředkovatel také umožňuje přijmout zprávu od producenta, který poté může pokračovat ve zpracování. Později, až přijde odezva od konzumenta, může být upozorněn a odezvu přijmout. Pokud konzument v danou chvíli není dostupný, zpráva může vyčkat u zprostředkovatele a být doručena později. Tomuto se říká *asynchronní programování*.

Zasílání zpráv má i spoustu nevýhod. Např. nelze použít výjimky, když se zpracování nezdaří. Při zaslání odpovědi na požadavek se musí producent stát konzumentem a konzument producentem, kdežto při použití RMI-IIOP je odezva návratovou hodnotou požadavku. Zasílání zpráv je méně výkonné z důvodu zvýšení režie na zprostředkovatele.

Pro zasílání zpráv existuje mnoho systémů. Podobně jako JNDI řeší problém více produktů, existuje takové řešení i pro dodavatele systémů pro zasílání zpráv. Tímto řešením je *JMS* (*Java Message Service*). Obsahuje také jediné API, které je používáno vývojářem, a pak SPI, které je použito jednotlivými dodavateli.

Při použití zasílání zpráv je nutné zvolit si tzv. *doménu*, která se určuje v popisovači nasazení. Doménou je např. tzv. *zveřejnit/přihlásit*

(*Publish/Subscribe*). V této doméně je možné mít několik producentů a několik konzumentů. Jednotliví konzumenti se přihlašují k odebírání určitého *tématu* (*Topic*). Producenti poté zasílají zprávy všem přihlášeným.

Další doménou je *dvoubodový spoj* (*Point-To-Point*). Zprávy jsou zde zasílány producenty do tzv. *fronty* (*Queue*). Jakmile nějaký konzument přijme zprávu, dochází k jejímu odstranění z fronty, tudíž každá zpráva je přijata právě jedním konzumentem.

Postup při použití JMS API je tento:

1. vyhledat *továrnu na spojení* (*Connection Factory*) – zajišťuje přístup ke konkrétnímu JMS ovladači, umožňuje založit spojení, vyhledává se pomocí JNDI;
2. založit *spojení* (*Connection*) s poskytovatelem JMS – umožňuje založit sezení;
3. založit *sezení* (*Session*) – slouží pro zakládání objektů zpráv, producentů a konzumentů;
4. vyhledat *místo určení* (*Destination*) – místo, kam se posílají nebo odkud se přijímají zprávy, vyhledává se pomocí JNDI;
5. založit *producenta* (*Producer*) nebo *konzumenta* (*Consumer*);
6. poslat nebo přijmout *zprávu* (*Message*) – v případě posílání je nutné zprávu nejdříve vytvořit, v případě přijímání je nutné zprávu analyzovat.

Bean řízený zprávami je komponenta, která umožňuje přijímat JMS zprávy. K tomuto beanu nejsou žádná rozhraní komponenty ani třídní rozhraní, pro komunikaci nepoužívá RMI-IIOP, ale JMS. Zprávy se beanům posílají pomocí JMS API. Beany je přijímají prostřednictvím jediné metody `onMessage`, která je vyvolána při příchodu zprávy.

Kromě rozhraní `javax.ejb.MessageDrivenBean` musí bean implementovat i rozhraní `javax.jms.MessageListener`. Rozhraní `javax.ejb.MessageDrivenContext` umožňuje přístup ke službám kontejneru.

2.5.9 Bezpečnost

Zabezpečení EJB komponenty má dvě etapy. První etapou je tzv. *ověření pravosti* (*Authentication*). Jde o ověření klienta, zda je tím, za koho se vydává.

Druhou částí je tzv. *zmocnění* (*Authorization*). Klientovi, který prošel první etapou, jsou přidělena oprávnění pro provádění požadovaných operací.

Obě tyto etapy jsou zahrnuty v *JAAS* (*Java Authentication and Authorization Service*). Pro ověření pravosti JAAS umožňuje používat různé metody. Může to být např. ověření oproti databázi, oproti jmenné službě nebo cokoliv jiného.

Klient může být v tomto případě dvojího druhu. Buďto jde o vzdáleného klienta, přistupujícího k EJB komponentě pomocí programu napsaného v jazyce Java, nebo jde o klienta používajícího webový prohlížeč. Při použití webového prohlížeče jsou možné čtyři způsoby (viz tabulku 2.4), jak může klient předat ke kontrole svoje *přihlašovací údaje* (např. jméno a heslo nebo bezpečnostní certifikát). Podrobněji

základní ověření	*Basic Authentication*
ověření pomocí otisku	*Digest Authentication*
ověření založené na formuláři	*Form-Based Authentication*
ověření pomocí certifikátu	*Certificate Authentication*

Tabulka 2.4: Čtyři způsoby ověření pravosti u webového klienta

se lze o těchto čtyřech přístupech dočíst např. v [6]. Při základním ověření webový klient zadává jako přihlašovací údaje svoje jméno a heslo, které se předají serveru, kde se ověří např. proti databázi jmen a hesel, a podle výsledku se klientovi přiřadí oprávnění. Ověření pomocí formuláře je podobné, jen umožňuje klientovi použít speciální webovou stránku s formulářem pro zadání jména a hesla. V obou případech se nejedná o bezpečný způsob, protože jméno i heslo putují k serveru nechráněné. Lze ale nastavit webový server tak, aby používal bezpečný přenos informací pomocí *SSL* (*Secure Socket Layer*).

Pokud nejde o webového klienta, ale o samostatnou aplikaci v Javě, pak je nutné přečíst jméno a heslo v rámci aplikace a tyto

údaje poté poslat serveru k ověření pravosti.

JAAS je velmi obecný nástroj, a proto je také poměrně složitý. Zde je postup pro ověření pravosti pomocí JAAS:

1. Klient vytvoří novou instanci *přihlašovacího kontextu* (*Login Context*). Jde o třídu poskytnutou aplikačním serverem, která je odpovědná za koordinaci procesu ověřování pravosti.
2. Přihlašovací kontext zjistí požadované typy ověření pravosti podle *konfigurace* (*Configuration*). V konfiguraci může být např. nastaveno, že je potřeba ověření pravosti jménem a heslem oproti databázi i ověření pravosti pomocí certifikátu.
3. Pro každý požadovaný typ je vytvořena instance tzv. *přihlašovací jednotky* (*Login Module*). Každá z přihlašovacích jednotek obsahuje konkrétní ověřovací mechanismus. Např. přihlašovací jednotka pro ověření oproti databázi obsahuje kód pro přístup k databázi, načtení údajů, porovnání se zadanými údaji od klienta a vrácení výsledku.
4. Na přihlašovacím kontextu se zavolá metoda `login`. Tato metoda volá metody `login` všech požadovaných přihlašovacích jednotek.
5. Při úspěchu přihlašovací jednotky (např. jméno a heslo odpovídají záznamu v tabulce uživatelů) se volá metoda `commit`. Jinak je volána metoda `abort`.

V případě, kdy by bylo nutné psát vlastní přihlašovací jednotky, by uvedený postup musel být podrobnější. Avšak některé aplikační servery nabízí několik základních jednotek ve své implementaci, proto není nutné zabývat se přihlašovacími jednotkami příliš zevrubně. Přihlášení je, při použití dodávaných přihlašovacích jednotek, záležitostí přihlašovacího kontextu a konfigurace. Pokud se přihlášení nezdaří, je hozena výjimka `javax.security.auth.login.LoginException`.

Druhým krokem k zajištění zabezpečení je zmocnění. *Naprogramované zmocnění* se realizuje programováním bezpečnostních testů přímo do kódu komponenty. K tomu lze využít metod EJB kontextu (viz 2.5.5) `getCallerPrincipal` a `isCallerInRole`.

Deklarované zmocnění je realizováno v popisovači nasazení, kde jsou stanoveny tzv. *role*. Role je skupina uživatelů se stejnými oprávněními. Po ověření pravosti je uživatel k nějakým rolím přiřazen. U jednotlivých metod komponenty je pak v popisovači nasazení deklarováno, jaké role mohou metodu volat a jaké ne.

Pro další informace týkající se JAAS lze použít např. [24].

2.5.10 Transakce

Při pracování s databází se lze setkat s problémem přístupu více klientů ke stejným datům. Aby nedošlo např. ke vzájemnému přepisování dat, je možné použít *transakce*. Transakce zajišťuje, že operace, které jsou v transakci obsaženy, se uskuteční buďto všechny, nebo žádná. Základní vlastnosti transakcí jsou *nerozdělitelnost* (*Atomicity*), *neporušenost* (*Consistency*), *odloučení* (*Isolation*) a *trvalost* (*Durability*), které jsou rovněž známé pod zkratkou *ACID*.

Existuje několik *transakčních modelů*. Jedním z nich jsou *přímé transakce* (*Flat Transactions*). Je to nejjednodušší model, ve kterém po zahájení transakce proběhnou buď všechny požadované operace, nebo žádná. Nevýhodou tohoto modelu je, že pokud se zpracování skládá z mnoha operací a až některá z posledních způsobí chybu, celá předchozí práce musí být stornována.

Dalším modelem jsou *vnořené transakce* (*Nested Transactions*). V některých případech je vhodné zachovat výsledky již provedených operací a neúspěšnou operaci např. opakovat později. Proto se všechny požadované operace obalí jednou transakcí a každá z operací se navíc obalí menšími transakcemi (a takto lze pokračovat). Při neúspěchu menší transakce je možné zachovat jiné již provedené menší transakce a neúspěšnou část opakovat později. Pokud ani později není možné provést menší transakci, je stornována celá transakce obalující všechny operace. Nevýhodou tohoto modelu je, že jej specifikace EJB nevyžaduje, a tudíž ne každý aplikační server jej podporuje.

Transakce mohou být v EJB použity trojím způsobem. *Programové transakce* jsou zajištěny vývojáři komponent. Třídy beanu tedy musí ve svém kódu transakce zakládat a volat jejich zapsání (metoda `commit`) i stornování (metoda `rollback`). Programové transakce se používají prostřednictvím *JTA* (*Java Transaction API*). Existuje několik dodavatelů služeb zajišťujících transakce. Podobně jako JNDI řeší

problém s více dodavateli jmenných služeb, tak i JTA řeší problém s více dodavateli transakčních služeb. Dodavatelé používají rozhraní *JTS* (*Java Transaction SPI*), na které napojují ovladače pro svůj vlastní produkt. JTA je potom rozhraní použité vývojáři, kteří díky provázání JTA s JTS mohou používat jediné rozhraní pro přístup k různým transakčním službám. Pro programové transakce se využívá rozhraní `javax.transaction.UserTransaction`. Toto rozhraní popisuje metody objektu, který je nalezen pomocí JNDI.

Deklarované transakce umožňují komponentám, aby byly automaticky zahrnuty v transakcích. Při použití tohoto typu jsou transakce spravovány kontejnerem (je využit lapač požadavků pro komunikaci s transakční službou) a je nutné použít *transakční atribut*, který může nabývat šesti hodnot. Tyto hodnoty jsou shrnuty v tabulce 2.5.

Required	Komponenta je vždy zapojena v transakci. Pokud již transakce existuje, zapojí se komponenta do ní, pokud ne, založí se nová transakce.
RequiresNew	Pro komponentu je vždy založena nová transakce.
Supports	Komponenta je zapojena v transakci jen pokud nějaká transakce již existuje.
Mandatory	Transakce musí existovat již při použití komponenty. Pokud neexistuje, je vyhozena výjimka.
NotSupported	Komponenta nebude používat transakce. Pokud nějaká transakce již existuje, je odložena, a po ukončení činnosti komponenty je v ní pokračováno.
Never	Komponenta nebude používat transakce. Pokud již nějaká existuje, je hozena výjimka.

Tabulka 2.5: Transakční atribut

Poslední možností je použít *klientem spravované transakce*. U tohoto typu jsou transakce založeny klientem, stejně tak klient zajišťuje zapsání i stornování.

Pro další informace o transakcích a dalších databázových principech lze použít např. [34].

2.5.11 Vztahy entitních beanů

Mezi komponentami lze používat *vztahy* (*Relationships*), které umožňují přehledně strukturovat vznikající aplikaci. Příkladem vztahu může být např. objednávka a položky objednávky. Pro CMP entitní beany je možné nechat zajištění vztahů na kontejneru. Pro popis vztahů se v tomto případě používá popisovač nasazení. Vztahy mezi ostatními typy beanů zajišťuje vývojář v kódu komponent. Nejčastěji se však vztahy používají mezi entitními beany.

Vztahy mají několik atributů. Jedním z nich je *mohutnost* (*Cardinality*). Mohutnost vztahu může být 1:1, 1:N nebo M:N, přičemž vztah typu M:N může být převeden na dva vztahy typu 1:N.

Dalším atributem vztahu je *směrovost* (*Directionality*), která se používá pro určení směru vztahu. Někdy je potřeba komponentě, která je ve vztahu s jinou komponentou, zakázat, aby k ní mohla přistupovat.

Líné načítání (*Lazy Loading*) lze použít v případě, kdy není potřeba číst z databáze všechny komponenty, se kterými má daná komponenta vztah. Z databáze se načtou vztažené komponenty, až když jsou potřeba, tedy až je k nim přistupováno. Opakem je *agresivní načítání* (*Aggressive Loading*), kdy jsou přečteny všechny vztažené komponenty během jedné transakce.

Vztah mezi beany může být dvojího druhu. Buď jde o *seskupení* (*Aggregation*), nebo o *složení* (*Composition*). Seskupení lze popsat slovem „používá“. Např. student používá kurz. Při smazání entity studenta není nutné mazat entitu kurzu. Složení oproti tomu lze popsat slovy „je složeno z“. Např. objednávky jsou složeny z položek objednávky. Při smazání objednávky budou smazány i položky objednávky. Rozdíl mezi těmito dvěma přístupy se v EJB zajistí pomocí *řetězového mazání* (*Cascading Delete*), které se uplatňuje pro složení.

Důležitou výhodou, kterou poskytují vztahy u CMP entitních beanů, je použití EJB-QL pro vztahy. U entitních beanů (viz 2.5.7) bylo uvedeno, že se EJB-QL používá pro vyhledávání instancí entitních beanů. Stejně tak lze EJB-QL použít i pro vyhledávání instancí entitních beanů, které jsou ve vztahu. V příslušném dotazu je možné vztahy procházet a kontrolovat např. počet provázaných komponent i jejich atributy.

EJB umožňuje mít i *rekurzivní vztahy*, pro které platí vše, co platí pro nerekurzivní vztahy.

2.5.12 Novinky ve specifikaci EJB verze 2.1

Hlavní novinkou v připravované verzi specifikace EJB je podpora *webových služeb* (*Web Services*). Díky tomu došlo k přidání dalšího typu rozhraní, které se nazývá *koncový bod* (*Endpoint*). Dále byla přidána kontejnerem spravovaná služba *časovače* (*Timer*), která umožňuje využívat časově řízené události. Byla také zobecněna architektura beanů řízených zprávami tak, aby podporovaly i další typy zasílání zpráv (např. email). Důležité je i rozšíření jazyka EJB-QL, který nyní umožňuje např. seřazení výsledku dotazu nebo seskupující funkce.

Pro další informace o chystané specifikaci lze použít např. [27, 18, 19].

2.6 Návrhové vzory pro EJB

Návrhové vzory (*Design Patterns*) popisují a řeší stále se opakující problémy objektově orientovaného návrhu. K nabídnutým řešením vzory poskytují i důsledky jejich použití. Podrobné informace o návrhových vzorech včetně jejich katalogu lze nalézt v [9]. Zde budou popsány některé ze vzorů zaměřených na EJB. Pro úplný a podrobnější popis EJB návrhových vzorů je možné použít [15]. Návrhové vzory pro aplikace psané s použitím J2EE technologie jsou popsány např. v [25]. EJB návrhové vzory jsou rozděleny do pěti kategorií. Tyto kategorie budou popsány v následujících částech.

2.6.1 Návrh architektury EJB vrstev

Do první kategorie patří vzory pro *návrh architektury EJB vrstev*. Zde se lze setkat s často používaným vzorem *fasáda sezení* (*Session Façade*). Tento vzor se používá pro oddělení vrstvy zajišťující funkce systému od vrstvy spravující databázi. Hlavní výhodou je umožnění klientovi vykonat požadované funkce v jedné transakci pomocí jediného vzdáleného volání. Myšlenka, pomocí které toho lze dosáhnout, je použít beany sezení jako prostředníka mezi klientem a entitními beany. Klient tedy nemůže přímo přistupovat k metodám entitních beanů, k tomu slouží metody beanů sezení, ke kterým klient přistupuje vzdáleně. Tyto metody poté přistupují k entitním beanům

prostřednictvím místních rozhraní, čímž je práce s entitními beany efektivnější (viz 2.5.3 a 2.5.7).

Dalším vzorem v této kategorii je *továrna na objekty pro přenos dat* (*Data Transfer Object Factory*). Objekt pro přenos dat (*DTO*, viz 2.6.2) se používá jako seskupení dat potřebných pro daný případ užití. Může jít např. o sjednocení podmnožin atributů různých entitních beanů. DTO je potřeba spravovat. K tomuto účelu slouží továrna na DTO, která obsahuje metody vracející správně naplněné DTO. Metoda v továrně na DTO tedy např. načte potřebné entitní beany a z některých jejich atributů (z těch, které jsou potřebné v požadovaném DTO) vytvoří novou instanci DTO. Stejně tak továrna obsahuje metody pro zápis DTO zpět do databáze, které převezmou atributy z DTO a vloží je do odpovídajících entitních beanů.

Uživatelské rozhraní (*Business Interface*) je jednoduchým vzorem, který umožňuje zjednodušit práci s rozhraním komponenty daného beanu. Motivací tohoto vzoru bylo použití stejných hlaviček metod v rozhraní komponenty i ve třídě beanu. Vzor navrhuje použít uživatelské rozhraní, ve kterém jsou shrnuty všechny společné hlavičky metod. Rozhraní komponenty poté bude rozšiřovat uživatelské rozhraní a třída beanu jej bude implementovat. Tím je zajištěno, že přidáním hlavičky metody do uživatelského rozhraní se automaticky přidá i do rozhraní komponenty. Zároveň nahlásí kompilátor chybu překladu, pokud tato metoda ve třídě beanu nebude implementována.

2.6.2 Přenos dat mezi vrstvami

Další kategorii tvoří vzory pro *přenos dat mezi vrstvami*. Hlavním vzorem v této části je *objekt pro přenos dat* (*Data Transfer Object*, *DTO*). Tento objekt je použit např. továrnou na DTO (viz 2.6.1). Další využití DTO je např. pro přenos dat mezi klientem a serverem, kdy je možné přenést větší množství údajů pomocí jednoho vzdáleného volání. Lze je tedy s výhodou použít ve spojení se vzorem fasáda sezení (viz 2.6.1).

2.6.3 Práce s transakcemi a trvalostí

Třetí kategorií jsou vzory pro *práci s transakcemi a trvalostí* (*Persistence*). Patří sem např. vzor *číslování verzí* (*Version Number*), který je možné použít v případech přístupu k entitním beanům pomocí více transakcí. Např. jeden uživatel přečte hodnotu z entitního beanu do DTO v jedné transakci, pak ji přečte druhý uživatel v jiné transakci. Oba uživatelé hodnotu změní a zapíší zpátky v dalších dvou transakcích. Protože akce přečtení, změny a zapsání neproběhly v jedné transakci, přepsal jeden z uživatelů hodnotu v databázi druhému uživateli (došlo k porušení odloučení, viz 2.5.10). V tomto případě lze použít číslování verzí, kdy se přímo do beanu přidá atribut *číslo verze*, který se mění při zápisu do entitního beanu. Pokud se při zápisu liší číslo zapisované verze (uložené v DTO) od té, která je v entitním beanu, zápis se neprovede a je vyhozena výjimka.

Dalším vzorem v této kategorii je *JDBC pro čtení* (*JDBC for Reading*), který doporučuje používat JDBC (viz [23, 10]) v případech, kdy je nutné přečíst data z několika entitních beanů. Lze si představit případy, ve kterých je potřeba z několika rozsáhlých entitních beanů přečíst z každého jen malý počet atributů. Pro takové případy je neefektivní přistupovat k jednotlivým entitním beanům a získávat tak postupně potřebná data. Mnohem efektivnější je přistoupit pomocí JDBC přímo k databázi a přečíst pomocí jediného dotazu potřebná data z několika tabulek. Tento postup nelze použít pro zápis, protože entitní beany mohou být ve vztazích (viz 2.5.11), které by se přímým zápisem do databáze mohly porušit. JDBC pro čtení lze použít např. prostřednictvím vzoru fasáda sezení (viz 2.6.1).

2.6.4 Interakce na straně klienta

Do čtvrté kategorie se řadí vzory pro *interakce na straně klienta*. Patří sem např. vzor *továrna na třídní objekt* (*EJBHome Factory*). Instance třídního objektu nějakého beanu je obvykle vyhledávána klientem pomocí JNDI (viz 2.5.3). Nalezená instance se pak často použije pro založení instance lapače požadavků daného beanu. Pokud potřebuje instanci třídního objektu i jiný klient, musí ji znovu pomocí JNDI vyhledávat. To vždy zahrnuje inicializaci počátečního kontextu a následné vyhledávání. Tento stále se opakující postup je časově náročný.

Proto vzor továrna na třídní objekt navrhuje vytvořit třídu s třídní metodou vracející třídní objekt. Jeho vyhledání proběhne jen jednou, poté se nalezený objekt uloží do třídní proměnné, která je při příštím volání třídní metody vrácena.

2.6.5 Tvorba primárních klíčů

Poslední kategorii tvoří vzory popisující způsoby *tvorby primárních klíčů*. Jedním ze vzorů této kategorie jsou *sekvenční bloky* (*Sequence Blocks*). Tento vzor slouží pro unikátní generování celých čísel, která jsou poté použita jako primární klíče entitních beanů. Využívá tabulku (reprezentována entitním beanem), která má dvě položky. První je jméno entitního beanu a druhá je celé číslo reprezentující poslední použitý primární klíč pro daný entitní bean. Položka jména beanu je primárním klíčem tabulky. Jakmile je potřeba získat pro nějaký entitní bean další primární klíč, zjistí se hodnota posledního přiřazeného klíče podle jména beanu, zvýší se o jedna, zapíše se do tabulky a přiřadí se nově vytvářenému beanu. Protože by však takové použití bylo neefektivní (pro každý nový entitní bean je potřeba hledání v databázové tabulce, přečtení řádku a zápis řádku), navrhuje vzor spolupráci se vzorem fasáda sezení (viz 2.6.1). Kromě použití beanu sezení je ve vzoru navrženo ještě jedno zlepšení, které spočívá ve využití vyrovnávací paměti. Do vyrovnávací paměti se načítá několik vygenerovaných primárních klíčů pro každý bean a do tabulky se pak zapíše poslední z nich. Při vytvoření nového entitního beanu se nejprve prohledá vyrovnávací paměť. Pokud v ní již není dostupný žádný nový primární klíč, dojde k novému naplnění vyrovnávací paměti blokem klíčů načtených z databáze.

2.7 Struts

Důležitým produktem, který je možné použít při tvorbě J2EE aplikací, je *rámec* (*Framework*) *Struts*[1]. Jedná se o rámec, který řídí kontrolní tok aplikace, umožňuje snadnou vícejazyčnou podporu, zpřehlednění kódu JSP stránek nebo kontrolu vstupních údajů z webových formulářů. V této části budou podrobněji popsány možnosti, které

[1] Struts znamená *podpěry*, což naznačuje i účel tohoto rámce.

rámec Struts nabízí. Další podrobnosti lze nalézt např. v [7], kde se pojednává o Struts verze 1.1. V této nové verzi došlo oproti předchozí verzi k několika podstatným změnám, a tudíž většina volně dostupné dokumentace a příkladů se stala zastaralou.

2.7.1 Model 2

Klíčovým prvkem rámce Struts je využití tzv. *Modelu 2* pro řízení kontrolního toku J2EE aplikace. Hlavní rozdíl mezi Modelem 1 a Modelem 2 spočívá v zajištění zpracování požadavku různými komponentami. U Modelu 1 za zpracování odpovídá stránka JSP, která také zobrazuje výstup u klienta.

Model 2 (nazýván též *Model-Pohled-Řadič* či *Model-View-Controler*, *MVC*) je tvořen třemi komponentami. Za zpracování požadavku zde odpovídá servlet, který se nazývá *řadič* (*Controller*). Řadič po zpracování požadavku určí, která stránka JSP se zobrazí jako odezva. Zodpovídá tedy za kontrolu průběhu a uživatelského vstupu. Komponenta *modelu* (*Model*) zodpovídá za správu vnitřní struktury. Komponenta *pohled* (*View*) zodpovídá za vnější reprezentaci vnitřní struktury.

2.7.2 Přeposlání versus přesměrování

Při požadavku na stránku je někdy potřebné tento požadavek sdílet mezi více komponentami. Sdílení je možné provést dvěma způsoby. Jedním je *přeposlání požadavku* (*Request Forward*) a druhým je *přesměrování požadavku* (*Request Redirect*). Oba principy jsou použity řadičem rámce Struts.

Pro přeposlání je použit tento postup:

1. Klient pošle prostřednictvím webového prohlížeče požadavek webovému serveru.
2. Webový server předá požadavek (i atributy, které jsou uloženy v požadavku) jinému zdroji na serveru (tedy klient o tomto není informován).
3. Webový server pošle odezvu klientovi.

Pro přesměrování je použit tento postup:

1. Klient pošle prostřednictvím webového prohlížeče požadavek webovému serveru.
2. Webový server vrátí chybový kód a žádá nové URL (nabízí pro přesměrování URL uložené v záhlaví odezvy `Location`).
3. Webový prohlížeč vytvoří a pošle nový požadavek (tím dojde ke ztrátě všech atributů uložených v původním požadavku).
4. Webový server pošle odezvu klientovi.

Přesměrování je pomalejší než přeposlání, protože je nutné poslat požadavek a odpověď dvakrát. Protože při přeposlání není klient informován o tom, že na jeho požadavek reaguje jiný zdroj, URL zadané ve webovém prohlížeči se nezmění.

2.7.3 Architektura Struts

Úlohy řadiče jsou ve Struts plněny několika komponentami. Jednou z nich je třída `org.apache.struts.action.ActionServlet`. Ta odpovídá za směrování HTTP požadavků odpovídajícím částem rámce Struts a konfiguruje se v popisovači webové aplikace.

Částí odpovědnou za analyzování a zpracování požadavku je třída `org.apache.struts.action.RequestProcessor`, od níž je možné vytvořit potomka a upravit tak způsob, jímž bude požadavek zpracován.

Mezičlánkem mezi požadavkem a odpovídající aplikační logikou je třída `org.apache.struts.action.Action`, tzv. *akce*. Nejčastějším využitím této třídy je překrytí metody `execute` v potomkovi. Metoda `execute` je volána řadičem při vyvolání akce, která je popsána v konfiguračním souboru Struts pro danou aplikaci. Konfigurace akce je dostupná i v průběhu aplikace pomocí třídy `org.apache.struts.action.ActionMapping`. Po dokončení akce je kontrola předána např. stránce JSP nebo jiné akci pomocí třídy `org.apache.struts.action.ActionForward`. Pro předání kontroly lze v konfiguračním souboru Struts k dané akci určit, zda půjde o přesměrování nebo přeposlání (viz 2.7.2). Značky pro konfiguraci přesměrování (přeposlání) mohou být deklarovány globálně nebo v rámci akce. Rámec Struts obsahuje i některé předdefinované akce, jejichž použití se zajistí deklarováním akce v konfiguračním souboru Struts.

2.7.4 Zásuvné jednotky

V konfiguračním souboru Struts lze nastavit, jaké *zásuvné jednotky* (*Plug-In*) se v aplikaci použijí. Mohou to být např. *dlaždice* (*Tiles*), které umožňují použít šablony pro návrh uživatelského rozhraní. Šablonou lze určit vzhled stránek aplikace, aniž by bylo nutné definovat jejich obsah. Ten se do stránek doplní za běhu. Dlaždice také poskytují knihovnu uživatelských značek pro zacházení se šablonami na stránkách JSP.

Další zásuvnou jednotkou dodávanou s rámcem Struts je *ověřovač správnosti* (*Validator*), který poskytuje několik předdefinovaných kontrolních funkcí pro uživatelské vstupy. Další funkce lze definovat v konfiguračním souboru ověřovače správnosti.

Je také možné si vytvořit vlastní zásuvné jednotky.

2.7.5 Formuláře

Často používanou třídou, která se uplatňuje při zachycení vstupních dat z formuláře a při jejich přenosu do třídy akce, je třída `org.apache.struts.action.ActionForm`. Třídu je možné použít pro ověření správnosti vstupních hodnot ve formuláři. K tomu slouží překrytí metody `validate` v potomkovi této třídy. Druhou překrývanou metodou bývá `reset`, která slouží pro nastavování počátečních hodnot instančních proměnných třídy formuláře. Tato metoda je volána při každém požadavku. Formuláře se nastavují v konfiguračním souboru Struts, kde je nutné nastavit typ formuláře, a pak jej svázat s jednou nebo několika akcemi.

Struts umožňují použít i tzv. *dynamické formuláře*. Jejich základem je třída `org.apache.struts.action.DynaActionForm`. Rozdílem oproti formulářům `ActionForm` je, že pro dynamické formuláře není nutné psát kód. Vznikají dynamicky podle popisu, který je uveden v konfiguračním souboru Struts. Problém s dynamickými formuláři je v tom, že již není možné úplně kontrolovat obsahy metod `reset` a `validate`. Je ale možné vytvořit potomka dynamického formuláře, kde lze tyto metody překrýt. Drouhou možností je využít zásuvnou jednotku ověřovače správnosti (viz 2.7.4).

2.7.6 Knihovny uživatelských značek

Struts poskytuje pro návrháře stránek JSP několik knihoven s uživatelskými značkami. Tematicky jsou to HTML značky, značky pro práci s objekty *JavaBeans* (viz [10, 28]), značky zajišťující logiku a značky pro práci se šablonami.

HTML značky rámce Struts umožňují definici formulářů a jiných prvků, ve kterých lze používat vícejazyčnou podporu nebo objekty JavaBeans. Mezi logické značky patří např. porovnávání proměnných se zadanými hodnotami, přesměrování či přeposlání, cykly apod. Značky pro práci s objekty JavaBeans usnadňují manipulaci s těmito objekty a umožňují jejich přejímání např. z požadavku. Také zde lze nalézt značky pro definici proměnných např. podle cookies nebo podle hlavičky požadavku. Značky šablon je možné použít pro dynamickou tvorbu uživatelského rozhraní, avšak v současné verzi Struts se spíše využívá zásuvná jednotka dlaždice (viz 2.7.4), která má mnohem více možností.

2.7.7 Vícejazyčná podpora

Za často uváděnou zkratkou *i18n*[2] se skrývá vícejazyčná podpora, která je pomocí Struts zajištěna rozšířením podpory nabízené jazykem Java. Je možné vytvořit několik souborů, které obsahují texty stránek v různých jazycích spolu s klíči. Klíče se poté pomocí speciálních značek rámce Struts (viz 2.7.6) umístí do stránek JSP. V konfiguračním souboru Struts lze potom nastavit umístění souborů a jejich jména.

2.7.8 Deklarace výjimek

Struts umožňuje webové aplikaci používat deklarativní způsob spravování výjimek. V konfiguračním souboru Struts lze deklarovat typ výjimky spolu se zdrojem, na který se provede přeposlání, pokud k výjimce dojde. Deklarování výjimky je možné v rámci akce nebo globálně pro celou aplikaci. V případě použití v rámci akce má tato deklarace přednost před globální deklarací.

[2] Zkratka vznikla tak, že *i* a *n* jsou počátečním a koncovým písmenem slova *Internationalization* a mezi nimi je 18 písmen.

2.8 Ant

Ant je sestavovací nástroj nezávislý na platformě, který je možné použít nejen pro automatizovanou kompilaci zdrojových kódů, ale např. i pro sestavení aplikace, její nasazení do aplikačního serveru, vytvoření dokumentace nebo spuštění testů. Ant podporuje ještě řadu dalších funkcí a lze jej rozšířit i o uživatelské funkce.

Ant se používá prostřednictvím *definičních souborů* (též *skriptů*), kde je pomocí XML deklarován seznam *úkolů* (*Task*), které se mají provést. Úkoly se sdružují do skupin, jimž se dávají jména. Skupiny se nazývají *cíle* (*Target*). Cíle jsou zařazeny do *projektu* (*Project*). U projektu se také určuje jméno cíle, který se provede při spuštění nástroje Ant, pokud nebude specifikováno jméno některého z deklarovaných cílů.

Jednou z výhod nástroje Ant je např. použití *vlastností* (*Property*). Do vlastností lze ukládat hodnoty, které je možné použít při pozdějším vykonávání skriptu. Pro uložení vlastností se také často používají externí soubory. Při distribuci zdrojových kódů aplikace je pak umožněno použít skript pro sestavení aplikace beze změny, protože všechny potřebné změny se promítnou pouze do souboru s vlastnostmi.

Často používanými značkami jsou definice cest. Používají se např. pro nastavení proměnné `CLASSPATH`. Další častou značkou je definice množiny souborů. Uplatňuje se např. při určení, které soubory se použijí při sestavování aplikace, které se použijí při kopírování nebo při převádění kódování (to se hodí např. pro vícejazyčnou podporu, viz 2.7.7).

Podrobnější informace o použití nástroje Ant lze nalézt v [1].

2.9 JBoss

Aplikační server *JBoss* je složen z několika komponent, jako jsou např. *JBossServer* (EJB kontejner), *JBossCMP* (zajišťuje CMP pro entitní beany), *JBossMQ* (zajišťuje služby JMS), *JBossSX* (bezpečnost pomocí JAAS) nebo *JBossTX* (zajišťuje transakční služby). JBoss také podporuje webové komponenty prostřednictvím integrace s webovými kontejnery jako jsou *Tomcat* nebo *Jetty*.

Tato část práce bude věnována aplikačnímu serveru JBoss verze 3.2.1. Nové verze serveru JBoss vznikají poměrně rychle a přináší většinou několik důležitých zlepšení. Bohužel s tím také často souvisí změna konfiguračních souborů, což může být v některých případech nežádoucí. Většina rysů, které JBoss nabízí, je již nakonfigurována a připravena k použití (např. JNDI). Avšak někdy je nutné některá nastavení upravit nebo dopsat. Potom se lze setkat s problémem, že dokumentace k serveru JBoss není nejkvalitnější. Kromě placené verze dokumentace je možné využít jen [33]. Dobrým zdrojem informací jsou ale např. soubory popisující struktury popisovačů nasazení a jiných konfiguračních souborů. Tyto soubory lze nalézt v instalaci serveru JBoss v adresáři `docs\dtd`.

Pro JBoss je možné vytvořit několik *konfigurací*. Každá z nich je uložena v instalačním adresáři serveru JBoss v podadresáři `server` a obsahuje svoje konfigurační soubory, knihovny apod. Nasazení J2EE aplikace (např. zabalené v archivu EAR) na server JBoss se rovná nakopírování aplikace do adresáře `deploy` příslušné konfigurace. V adresáři `log` dané konfigurace lze také nalézt žurnálové soubory s hlášeními, která se vypisují při startu či ukončení serveru nebo při požadavcích na jednotlivé aplikace.

2.9.1 MySQL

Kvůli demonstračním účelům byl při tvorbě aplikace použit databázový server *MySQL* (viz [20]). Pro přístup z aplikace prostřednictvím serveru JBoss byl použit následující postup:

1. Nakopírování konektoru databáze do adresáře `lib` příslušné konfigurace serveru JBoss.
2. Založení databáze v databázovém serveru MySQL.
3. Nastavení vytvořené databáze pro server JBoss pomocí souboru `mysql-ds.xml`, který vznikl úpravou souboru dodávaného se serverem JBoss (`docs\examples\jca\mysql-ds.xml`). Poté byl tento soubor nakopírován do adresáře `deploy` příslušné konfigurace serveru JBoss.

Podobný postup konfigurace lze uplatnit i pro jiné databázové servery.

2.9.2 Bezpečnost

Další nastavení, které bylo nutné změnit, se týkalo použití zabezpečení v aplikaci. JBoss rozlišuje zabezpečení na straně klienta a na straně serveru. Postup uvedený v 2.5.9 je tedy aplikován na obou stranách. Proto je nutné v popisovači nasazení pro server JBoss určit *bezpečnostní doménu*, ve které se aplikace nachází. Jméno bezpečnostní domény je použito při zakládání přihlašovacího kontextu. Pokud jménu bezpečnostní domény odpovídá jméno JAAS konfigurace, jsou podle této konfigurace založeny přihlašovací jednotky. Pokud neodpovídá, je použito jméno konfigurace `other`. Nastavení JAAS konfigurace na straně serveru se provede v souboru `login-config.xml`, který se nachází v adresáři `conf` příslušné konfigurace serveru.

Aplikační server JBoss nabízí několik přihlašovacích jednotek již ve své instalaci. Pomocí těchto jednotek lze kontrolovat přihlašovací údaje oproti databázi, souborům vlastností nebo např. jmenné službě. Je zde také klientská přihlašovací jednotka, která zajišťuje prosté předání jména a hesla serveru.

JBoss využívá pro přihlašovací údaje vyrovnávací paměť. To bohužel způsobuje, že pokud je změněno heslo uživatele, může uživatel i nadále používat staré heslo, protože JBoss jej porovnává se svojí vyrovnávací pamětí. Proto je nutné změnit nastavení v souboru `conf\jboss-service.xml` příslušné konfigurace serveru. V tomto souboru je potřeba změnit implicitní hodnoty dvou atributů. Jsou to `DefaultCacheTimeout` a `DefaultCacheResolution`. Tyto dva atributy se přidají do nastavení zásuvné jednotky (Plug-In) `JaasSecurityManagerService`.

2.10 XDoclet

Nástroj *XDoclet* slouží pro automatické vytváření kódu. Kódem zde může být v podstatě cokoliv, protože kromě vestavěných jednotek obsahujících definice tvořeného kódu lze používat i jednotky uživatelské. Vestavěné jednotky umožňují automatické vytváření kódu pro EJB jako jsou např. rozhraní komponenty, třídní rozhraní, popisovač nasazení (včetně některých popisovačů závislých na dodavateli). Také lze vytvářet kód uplatňující některé návrhové vzory jako např. továrnu na třídní objekt. Kromě kódu pro EJB je možné vytvářet

kód i v dalších oblastech. Např. pro webové aplikace (servlety, JSP stránky) lze vytvářet popisovač aplikace nebo konfigurační soubor nástroje Struts.

Nevýhodou nástroje XDoclet je, že vydání nové verze není závislé na vydání nové verze produktů, jejichž kód umožňuje vytvářet, čili některé prvky je nutné dopisovat ručně. XDoclet s touto variantou počítá a používá proto mechanismus tzv. *slévání* (*Merge*). Ten umožňuje napsat ručně pouze části, které XDoclet nevytváří, a tyto části pak XDoclet vloží na požadovaná místa.

Používání nástroje má dvě fáze. První je zahrnutí speciálních značek přímo do kódu beanu. Tyto značky se umisťují stejně jako značky sloužící pro vytváření dokumentace pomocí nástroje *JavaDoc*. Jsou tedy rozlišovány úrovně třídy, metody a proměnné objektu. Značky nástroje XDoclet mají i podobný formát. Podrobný popis značek lze nalézt v [35]. Bohužel je popis někdy neúplný.

Druhou fází použití je samotná automatizovaná tvorba kódu podle zadaných značek. Pro tento krok se používá nástroj Ant (viz 2.8). Použitím různých úkolů pro nástroj Ant, které jsou definovány v rámci nástroje XDoclet, se vytvoří různé kódy.

2.11 Eclipse

Vývojové prostředí *Eclipse* (viz [13]) je jedním z mnoha tzv. *IDE* (*Integrated Development Environment*). Další prostředí, která lze pro vývoj J2EE aplikací použít, jsou např. *JBuilder* nebo *NetBeans*.

V této práci bylo použito prostředí Eclipse, které má řadu velice příjemných usnadnění. Prostředí je uspořádáno pomocí *pohledů* a *editorů*, které tvoří tzv. *perspektivu*. Pohledy jsou často dodávány výrobci nástrojů, které lze poté v Eclipse prostřednictvím pohledů používat. Takovými nástroji jsou např. Ant (viz 2.8), JBoss (viz 2.9) nebo JUnit (viz [14]).

Perspektivy slouží pro definování použitých pohledů a editorů. Je tedy snadné přepínat se mezi perspektivami pro psaní kódu, ladění, testování apod.

Eclipse obsahuje zásuvnou jednotku pro programování v jazyce Java. Lze však instalovat i jiné jednotky pro jiné jazyky.

2.12 Extrémní programování

Extrémní programování (*Extreme Programming*, *XP*) je soubor postupů pro malé a střední vývojové týmy, které se často potýkají s nepřesnými nebo měnícími se požadavky. XP využívá běžně známé principy, které dotahuje do extrémů. Např. pokud se osvědčí revize kódu, má se kód revidovat neustále (programování v párech). Nebo pokud se osvědčí testování, bude se testovat neustále (testování jednotek). Při osvědčení návrhu se bude neustále vylepšovat návrh (refaktorování). A další.

V XP se lze setkat s modelem vývoje softwaru řízeným čtyřmi proměnnými. Jsou to *náklady*, *čas*, *kvalita* a *šíře zadání*. Vnější síly (zákazníci, manažeři) si volí hodnoty tří z těchto proměnných a vývojový tým si poté určí hodnotu čtvrté proměnné.

Protože většina technik extrémního programování se týká týmů, řízení, vybavení, rolí nebo spolupráce se zákazníkem, bylo možné použít tyto techniky pouze v omezené míře. Proto jsou v této práci použity především dva principy, a to testování (viz 2.12.1) a refaktorování (viz 2.12.2).

Více se lze o XP dočíst např. v [5].

2.12.1 Testování a Cactus

Jedním ze základních pilířů XP je testování. Hlavní myšlenkou je testy řízené programování, kdy se nejdříve napíší testy funkčnosti a poté se programuje. Programování je u konce, jakmile všechny testy projdou bez chyby a nelze již najít test, který by mohl skončit s chybou. Zároveň musí testy existovat pro všechny funkce a rozhodnutí, která byla učiněna.

Častým nástrojem uplatňovaným při testování aplikací napsaných v jazyce Java je *JUnit* (viz [14]). Při tvorbě aplikací s využitím J2EE tento nástroj není vhodný, protože neobsahuje podporu pro testování na straně serveru. Proto vzniklo rozšíření JUnit, které se jmenuje *Cactus*. Tento nástroj již testování na straně serveru podporuje.

Dalšími nástroji, které umožňují testovat komponenty nasazené na aplikační server, jsou např. *MockObjects* (viz [16]) nebo *HttpUnit* (viz [12]). Každý z těchto nástrojů používá jiný přístup. Např. test

pomocí MockObjects pracuje s testovanou třídou v izolovaném prostředí. Všechny potřebné objekty (např. požadavek, odezva) jsou simulovány speciálními objekty testovacího aparátu. Po volání testované metody s využitím simulovaných objektů jsou porovnány výsledky s očekávanými výsledky. HttpUnit např. nabízí strukturované testy odezvy, kdy přesně analyzuje vytvořený výstup, a umožňuje tak testovat např. hodnoty v tabulkách HTML pomocí objektů jazyka Java.

Naproti tomu Cactus spouští testy v opravdovém prostředí aplikačního serveru. Protože jde o rozšíření nástroje JUnit, je zachována struktura testů. Jednotlivé testy tvořené *testovacími případy* (*Test Case*) je možné shromažďovat do *skupin* (*Suite*). Instance třídy testovacího případu jsou dvě. Jedna je na straně klienta a druhá na straně serveru. Správu instancí zajišťuje Cactus pomocí dvou servletů. Jedním je `TestRunner` (pro místní instanci) a druhým `Redirector` (pro vzdálenou instanci). `Redirector` také na serveru spravuje objekty požadavku, odezvy a sezení.

Cactus přidává metodu `begin...`, která je volána ještě na straně klienta (pomocí `TestRunner`) a umožňuje nastavit objekt požadavku. Poté je předáno řízení serverové instanci, kde `Redirector` volá metody `setUp`, `test...` a `tearDown`. V metodách `test...` lze používat standardní metody nabízené třídami testovacích případů z JUnit. V případě výjimky způsobené metodou `test...` je výjimka zachycena servletem `Redirector` a poslána servletu `TestRunner`, který ji vypíše klientovi. Pokud k výjimce nedošlo, je na straně klienta zavolána metoda `end...`, která vrací vytvořenou odezvu.

Více podrobností o nástroji Cactus lze získat např. v [3]. Uvedená dokumentace zatím příliš nepočítá s použitím pro EJB, proto je podpora pro tuto technologii malá, avšak jedním z cílů vývojářů nástroje Cactus je tento stav změnit.

2.12.2 Refaktorování

Refaktorováním (*Refactoring*) se rozumí vylepšování vnitřní struktury kódu programu bez ovlivnění jeho vnější funkcionality. Refaktorování se provádí, protože zlepšuje přehlednost kódu programu, čímž usnadňuje jeho další úpravy. Také umožňuje snadněji hledat chyby,

protože pomáhá porozumění kódu. Podrobně se lze s principy, důvody a problémy refaktorování seznámit v [8].

Techniky refaktorování lze rozdělit do sedmi skupin. První skupinou jsou *úpravy metod*. Refaktorování patřící do této skupiny jsou např. vyjmutí metody, vložení metody nebo zavedení vysvětlující proměnné. Druhou skupinu tvoří *přesouvání prvků mezi objekty*, kam patří např. vyjmutí třídy nebo přesunutí metody. Další skupinou jsou refaktorování zabývající se *organizací dat* jako např. změna hodnoty na odkaz nebo nahrazení datové položky objektem. Refaktorování zařazená do čtvrté skupiny *zjednodušování podmíněných výrazů* jsou např. rozložení podmínky nebo zavedení předpokladu. Pátou skupinou je *zjednodušování volání metod*. Sem patří např. přidání parametru nebo skrytí metody. Další skupinu tvoří *generalizace*, kde lze nalézt refaktorování jako přesunutí položky výš nebo vyjmutí podtřídy. Poslední skupinou jsou *velká refaktorování*, kam náleží např. roztržení dědičnosti nebo oddělení datového modelu od prezentace.

Některá refaktorování lze snadno automatizovat, proto existují i specializované nástroje, které jednotlivé techniky refaktorování provádí. Vývojové prostředí Eclipse (viz 2.11) také obsahuje několik technik refaktorování.

2.13 JMeter

Přehledným a výkonným nástrojem pro testování zátěže je nástroj *JMeter*. Pomocí tohoto nástroje lze simulovat zátěž aplikace. JMeter používá pro definici zátěže různé zásuvné jednotky, pomocí kterých lze např. nadefinovat, kolik uživatelů bude simulováno, jaké posílají požadavky nebo jakou metodu posílání dat používají. Lze takto také určit formát výstupu provedeného testování. Celému nastavení se říká *testovací plán* (*Test Plan*). Možností JMeteru je také testovat odezvu nebo používat časovač pro náhodnější rozmístění požadavků. Lze také uložit testovací plán použitím JMeteru jako zástupce (*Proxy*) ve webovém prohlížeči.

Více informací o tomto nástroji lze nalézt v [2].

Kapitola 3

Tvorba internetového obchodního domu

Tato kapitola uvede postup vhodný pro tvorbu J2EE aplikací, který byl uplatněn při vývoji aplikace internetového obchodního domu. Kompletní aplikaci lze nalézt v příloze A. Při návrhu aplikace a její následné tvorbě byl dán největší důraz na rozmanitost použitých postupů a na obecnost řešení. Šlo o to ukázat co nejvíce možných použití technologií a nástrojů uvedených v kapitole 2.

3.1 Architektura

Internetový obchodní dům používá dva modely architektury. Prvním je *čtyřvrstvá architektura*, která je určena pro klienty používající aplikaci prostřednictvím webového prohlížeče. Jednotlivé vrstvy budou popsány v následujících částech. Dělení do vrstev může být provedeno fyzicky, kdy každá z vrstev může být implementována na jiném stroji, nebo logicky, podle funkcí, které vrstvy vykonávají.

Druhým modelem byla *třívrstvá architektura*, která je použita pro klienty, kteří přistupují k aplikaci pomocí klientských programů napsaných v jazyce Java. Tyto klientské programy přistupují přímo k aplikační vrstvě (viz 3.1.3) bez využití webové vrstvy (viz 3.1.2) jako prostředníka. Musí tedy zajistit funkce jak prezentační, tak i webové vrstvy.

3.1.1 Prezentační vrstva

Prezentační vrstva slouží pro komunikaci aplikace s uživatelem. Tato vrstva zajišťuje přijetí požadavku a jeho poslání webové vrstvě. Také přijímá odezvu od webové vrstvy, kterou poté prezentuje uživateli. Dále umožňuje přijímat vstupní data od uživatele např. pomocí

HTML formulářů.

3.1.2 Webová vrstva

Jako prostředník mezi prezentační a aplikační vrstvou působí *webová vrstva*, která zpracovává požadavek přijatý od uživatele a získává z něj předaná data. Tato data pak mohou být využita při volání metod aplikační vrstvy, z jejichž vrácených výsledků je tvořena odezva, která je zaslána prezentační vrstvě. Do webové vrstvy se řadí stránky JSP a servlety, které jsou umístěny ve webovém kontejneru.

3.1.3 Aplikační vrstva

V *aplikační vrstvě* se nachází aplikační server (v tomto případě EJB kontejner), který poskytuje nasazené aplikaci hostitelské prostředí. Některé jednodušší aplikace tuto vrstvu nevyužívají a veškeré funkce aplikace implementují ve webové vrstvě. Aplikační vrstva zajišťuje uživatelský cíl aplikace. Přijímá volání svých funkcí z webové vrstvy, které následně poskytuje výsledky těchto volání. Také zajišťuje komunikaci s databázovým serverem.

3.1.4 Vrstva databáze

Vrstvu databáze často představuje databázový server, ale může jít i o jiný existující podnikový systém, ke kterému lze přistupovat pomocí aplikační vrstvy. V případě databázového serveru se často využívá služby JDBC.

3.2 Seznámení s úkolem

Aplikace, která demonstruje použité technologie a postupy uvedené v kapitole 2, je jednoduchý internetový obchodní dům. Cílem implementace bylo vytvořit katalog zboží, ve kterém budou moci uživatelé listovat a vyhledávat s použitím webového prohlížeče. U každého zboží lze zobrazit jeho detailní informace. Zboží lze vkládat do košíku. Uživatel se také může přihlásit a poté jsou mu zpřístupněny další funkce jako pořízení objednávky, přehled objednávek včetně jejich stavu nebo uložení košíku a odhlášení ze systému.

Součástí požadavků na aplikaci je i možnost plnění katalogu zboží a evidence uživatelů, kteří se mohou do systému přihlásit.

3.3 Návrh aplikace

Aplikaci lze rozdělit do několika skupin určených architekturou (viz 3.1). Hlavní skupinou jsou EJB komponenty, které zajišťují funkčnost celé aplikace. Další skupiny tvoří testovací třídy, třídy Java klientů, třídy DTO, třídy pro webové klienty a pomocné třídy. Vztahy mezi těmito třídami lze znázornit pomocí UML (pro použití UML s EJB viz např. [4]).

3.3.1 Produkty a složky

Základem pro celý návrh bylo určení objektů, které budou reprezentovat katalog. Při jejich návrhu bylo stěžejní myšlenkou využít možností, které nabízí technologie EJB (viz 2.5), tedy především vztahy mezi entitními beany typu CMP a líné načítání (viz 2.5.11).

Díky tomu bylo možné navrhnout třídu, která reprezentuje produkty v katalogu. Touto třídou je `ProduktBean`. Produkty lze slučovat do skupin (složek), které mohou obsahovat i jiné složky. Produkt může být také umístěn v několika složkách.

Složky jsou reprezentovány opět třídou `ProduktBean`, ve které je příznak, zda se jedná o složku nebo o produkt. Využitím příznaku bylo možné vytvořit rekurzivní vztah definovaný na třídě `ProduktBean`. Takto byla získána stromová struktura katalogu zboží, ve které jsou rodiče vždy složkami a potomci jsou složky nebo produkty.

Byl možný i jiný přístup, který by oddělil složky a produkty do dvou různých tříd, avšak tento přístup by neumožnil demonstrovat možnosti EJB při tvorbě vztahů.

Dalším důvodem pro využití rekurzivního vztahu bylo zefektivnění vyhledávání, které se děje pomocí jazyka EJB-QL. Tento jazyk umožňuje jednoduše procházet proměnné tříd, čímž bylo možné velice snadno rozlišit produkty od složek.

3.3.2 Atributy

Protože se jedná o všeobecný internetový obchodní dům, není možné znát povahu zboží, které nabízí. Proto bylo nutné implementovat produkty obecně. K tomu je využita třída `AtributBean`, která uchovává dvojice klíč a hodnota. Tyto dvojice určují atributy produktu (nebo složky). Lze tedy snadno přidávat jakékoliv atributy a definovat tak konkrétní zboží.

Navíc je zajištěno, že přidáním atributu ke složce je tento atribut přidán ke všem produktům a složkám, které obsahuje. Neděje se tak ale automaticky. Pro každý produkt i složku je zkontrolováno, zda už atribut neobsahuje a pokud ne, je uživatel dotázán, zda si jej přeje změnit. Pokud ne, je převzat s již nastavenou hodnotou.

Kromě klíčů a hodnot je uchováván ještě typ dat daného atributu, který slouží pro řazení, protože např. abecední řazení je odlišné od číselného.

3.3.3 Košík

Při návrhu košíku bylo potřeba vzít v úvahu dvě situace. Zaprvé, že uživatel musí být schopen vkládat zboží do košíku, i když ještě není přihlášen. Zadruhé, že uživatel musí mít možnost si košík uložit a vrátit se k němu později. Každá z těchto variant byla implementována jiným způsobem.

Uživatel vkládá zboží do košíku sezení (bean sezení, viz 2.5.6), který je spojen se sezením uživatele (nepřihlášeného i přihlášeného). Tento košík je reprezentován třídou `KosikSessionBean`.

Existuje i entitní košík, který je reprezentován třídami `KosikBean` a `KosikPolozkaBean`. Entitní košík je spojen pouze s přihlášeným uživatelem a využívá se jen při jeho přihlašování a odhlašování. Při odhlašování je uložen obsah košíku sezení do entitního košíku (tedy do databáze) s tím, že všechen jeho předchozí obsah je odstraněn. Při přihlášení uživatele se ověří, zda pro něj existuje nějaký entitní košík z minula a pokud ano, načte se jeho obsah do košíku sezení.

3.3.4 Objednávky

Uživatel může vytvářet objednávky ze svého košíku sezení. Také má možnost prohlížet si vytvořené objednávky a kontrolovat jejich stav. Existují i zvláštní uživatelé, kteří mají právo stav objednávek měnit. Objednávky jsou reprezentovány třídami `ObjednavkaBean` a `ObjednavkaPolozkaBean`.

3.3.5 Uživatelé

Uživatelé jsou reprezentováni svými přihlašovacími jmény. Pro každého uživatele existují role, ve kterých může vystupovat. Může jít např. o anonymního webového uživatele, správce katalogu, správce uživatelů nebo přihlášeného uživatele. Aplikační server JBoss (viz 2.9) doporučuje použít dvě tabulky. Proto jsou uživatelé reprezentování dvěma entitními beany `RoleBean` a `PrincipalBean`.

Bylo nutné také vyřešit automatické založení uživatele při nasazení aplikace, který má právo spravovat uživatelské účty.

3.4 Implementace

Cílem implementace bylo aplikovat technologie, nástroje a poznatky uvedené v kapitole 2 a zachovat řešení co nejvíce obecné. Proto je v této části uvedeno shrnutí použitých nástrojů a postupů.

Pro implementaci *entitních beanů* (viz 2.5.7) byla zvolena varianta *CMP*, aby bylo možné snadno zaměnit databázový server. Entitní beany byly také obaleny *beany sezení* (viz 2.5.6) podle návrhového vzoru *fasáda sezení* (viz 2.6.1). Dalšími použitými vzory (viz 2.6) byly např. *továrna na třídní objekt*, *DTO*, *továrna na DTO* nebo *sekvenční bloky* pro tvorbu primárních klíčů. Návrhový vzor *uživatelské rozhraní* ztrácí svůj význam s použitím nástroje *XDoclet* (viz 2.10), který byl uplatněn např. pro tvorbu rozhraní a popisovačů nasazení. Zabezpečení aplikace bylo provedeno využitím služby *JAAS* (viz 2.5.9). U jednotlivých komponent byl použit model *deklarování transakcí* (viz 2.5.10). Pro tvorbu zdrojových kódů bylo použito vývojové prostředí *Eclipse* (viz 2.11), které má vestavěný nástroj *Ant* (viz 2.8). Ant byl uplatněn pro automatizovaný překlad a nasazení aplikace na server nebo pro překlad a provedení testů. Pro testování byl použit nástroj *Cactus* (viz

2.12.1). Aplikace byla nasazena na aplikační server *JBoss* (viz 2.9). Při tvorbě uživatelského rozhraní bylo využito technologie *JSP* (viz 2.3), která byla podpořena rámcem *Struts* (viz 2.7).

Kapitola 4

Závěr

Práce se zabývá jednotlivými postupy a principy, které se využívají při tvorbě aplikací v J2EE. Nejdříve je stručně popsána technologie *J2EE* (viz 2.1) včetně některých jejích částí. Jsou zde také stručně shrnuty základní postupy pro použití J2EE v praxi spolu s nástroji, které takové použití usnadňují. Byla vypracována aplikace jednoduchého internetového obchodního domu (viz příloha A) demonstrující uvedené postupy.

Největší důraz je kladen na využití *EJB komponent* (viz 2.5), které poskytují širokou škálu možností. Lze se tak věnovat skutečnému programování uživatelského cíle aplikace a není nutné se v implementaci zabývat službami, jako jsou např. *transakce* (viz 2.5.10) nebo *trvalost* (viz 2.5.7). Tyto služby jsou zajištěny kontejnerem.

Velkým přínosem pro EJB je použití nástroje *XDoclet* (viz 2.10). XDoclet velice zpřehledňuje a usnadňuje administrativní část programování komponent EJB tím, že umožňuje automaticky vytvářet zdrojové kódy nejen rozhraní, ale např. i popisovačů nasazení.

Dalším důležitým nástrojem použitým při tvorbě je rámec *Struts* (viz 2.7), který zavedením modelu MVC zpřehledňuje celou aplikaci a umožňuje jednoduše definovat její kontrolní tok. Usnadňuje také použití vícejazyčné podpory nebo rozšíření možností *JSP* (viz 2.3) přidáním knihoven uživatelských značek.

Návrhové vzory (viz 2.6) dovolují nejen lepší pochopení a využití možností EJB, ale i ucelenější návrh architektury celé aplikace.

Celkově byla práce velice zajímavá a různorodá, protože se jednalo o celou řadu nástrojů a principů, s nimiž bylo nutné se seznámit. Některé z nástrojů mají slabší prvky jako např. dokumentaci nebo častost vydání nové verze, ale vesměs se jedná o výkonné, přizpůsobitelné a užitečné produkty.

Literatura

[1] Apache Software Foundation: *Apache Ant Manual*. Dokument dostupný na URL
http://ant.apache.org/manual/index.html (únor 2004).

[2] Apache Software Foundation: *Apache JMeter documentation*. Dokument dostupný na URL
http://jakarta.apache.org/jmeter/index.html (únor 2004).

[3] Apache Software Foundation: *Cactus documentation*. Dokument dostupný na URL
http://jakarta.apache.org/cactus/index.html (únor 2004).

[4] Arrington, C. T.: *Enterprise Java with UML*. John Wiley & Sons, New York, 2001.

[5] Beck, Kent: *Extrémní programování*. Grada Publishing, Praha, 2002.

[6] Bollinger, Gary, Natarajan, Bharathi: *JSP – Java Server Pages*. Grada Publishing, Praha, 2003.

[7] Cavaness, Chuck: *Programujeme Jakarta Struts*. Grada Publishing, Praha, 2003.

[8] Fowler, Martin: *Refaktoring*. Grada Publishing, Praha, 2003.

[9] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John: *Návrh programů pomocí vzorů*. Grada Publishing, Praha, 2003.

[10] Hall, Marty: *Java servlety a stránky JSP*. Neocortex, Praha, 2001. Dokument dostupný v angličtině na URL
http://pdf.coreservlets.com (únor 2004).

[11] Herout, Pavel: *Učebnice jazyka Java*. Kopp, České Budějovice, 2001.

[12] *HttpUnit*. Dokument dostupný na URL http://httpunit.sourceforge.net (únor 2004).

[13] IBM Corporation: *Eclipse*. Dokument dostupný na URL http://www.eclipse.org (únor 2004).

[14] *JUnit*. Dokument dostupný na URL http://www.junit.org/index.htm (únor 2004).

[15] Marinescu, Floyd: *EJB Design Patterns*. John Wiley & Sons, New York, 2002. Dokument dostupný na URL http://www.theserverside.com/books/wiley/EJBDesignPatterns (únor 2004).

[16] *MockObjects*. Dokument dostupný na URL http://www.mockobjects.com/wiki (únor 2004).

[17] Monson-Haefel, Richard: *Enterprise JavaBeans*. O'Reilly & Associates, Sebastopol, 2001.

[18] Monson-Haefel, Richard: *Enterprise JavaBeans*. O'Reilly & Associates, dosud nevyšlo. Dokument dostupný na URL http://www.theserverside.com/books/review/ejbReview.jsp (kapitoly 1–17, únor 2004).

[19] Monson-Haefel, Richard: *What's New in EJB 2.1*. Dokument dostupný na URL http://www.theserverside.com/articles/article.jsp?l=MonsonHaefel-Column1 (únor 2004).

[20] MySQL AB: *MySQL Manual*. Dokument dostupný na URL http://www.mysql.com/documentation (únor 2004).

[21] Nožička, Josef, Zelený, Jindřich: *COM+, Corba, EJB*. BEN – technická literatura, Praha, 2002.

[22] Roman, Ed, Ambler, Scott, Jewell, Tyler: *Mastering Enterprise JavaBeans*. John Wiley & Sons, New York, 2002. Dokument dostupný na URL

http://www.theserverside.com/books/wiley/masteringEJB (únor 2004).

[23] Spell, Brett: *Java Programujeme profesionálně*. Computer Press, Praha, 2002.

[24] Stark, Scott: *Integrate security infrastructures with JBossSX*. 2001. Dokument dostupný na URL http://www.javaworld.com/javaworld/jw-08-2001/jw-0831-jaas.html (únor 2004).

[25] Sun Microsystems: *Blueprints – Core J2EE Patterns*. Dokument dostupný na URL http://java.sun.com/blueprints/patterns/index.html (únor 2004).

[26] Sun Microsystems: *Enterprise JavaBeans Specification, Version 2.0*. 2001. Dokument dostupný na URL http://java.sun.com/products/ejb/docs.html#specs (únor 2004).

[27] Sun Microsystems: *Enterprise JavaBeans Specification, Version 2.1*. Dokument dostupný na URL http://java.sun.com/products/ejb/docs.html#specs (únor 2004).

[28] Sun Microsystems: *JavaBeans Specification*. Dokument dostupný na URL http://java.sun.com/products/javabeans/glasgow/index.html (únor 2004).

[29] Sun Microsystems: *JavaServer Pages Specification*. Dokument dostupný na URL http://java.sun.com/products/jsp/download.html#specs (únor 2004).

[30] Sun Microsystems: *Java Servlet Specification*. Dokument dostupný na URL http://java.sun.com/products/servlet/download.html#specs (únor 2004).

[31] Sun Microsystems: *JNDI tutorial*. Dokument dostupný na URL http://java.sun.com/products/jndi/tutorial (únor 2004).

[32] Sun Microsystems: *RMI-IIOP tutorial*. Dokument dostupný na URL
http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop (únor 2004).

[33] The JBoss Group: *JBoss 3.0 Quick Start Guide*. Dokument dostupný na URL
http://sourceforge.net/projects/jboss (únor 2004).

[34] Ullman, Jeffrey D.: *Principles of Database Systems*. W. H. Freeman & Co., New York, 1983.

[35] XDoclet Team: *XDoclet Manual*. Dokument dostupný na URL
http://xdoclet.sourceforge.net (únor 2004).

Příloha A

Internetový obchodní dům

Na přiloženém CD lze nalézt kompletní aplikaci internetového obchodního domu, včetně zdrojových kódů a dokumentace. CD má strukturu uvedenou v tabulce A.1.

`apls`	zde lze nalézt prostředí Eclipse s nastaveným projektem (např. potřebné knihovny pro Ant, Cactus, Struts, XDoclet, ...) a také nakonfigurovaný server JBoss (konfigurace `default`)
`eshop`	adresář obsahující zdrojové kódy aplikace, API dokumentaci, balíčky připravené pro nasazení a klientskou část aplikace i se vzorovými dávkami pro spuštění
`install`	v tomto adresáři jsou nástroje a produkty jako server JBoss, databáze MySQL, Struts, Cactus a další ve verzích, které byly použity při vývoji ukázkové aplikace
`specs`	adresář obsahující specifikace komponent EJB, stránek JSP a Java servletů
`text`	zde je uložen tento text ve formátech PDF, PS, DVI i zdrojový kód v LaTeXu

Tabulka A.1: Struktura přiloženého CD

Příloha B

Nasazení aplikace eShop.ear

Aplikace je odzkoušena pro aplikační server JBoss verze 3.2.1 a databázový server MySQL verze 4.0.12. Nasazení aplikace probíhá takto:

1. Stáhnout a nainstalovat JBoss a MySQL.
2. Nastavit JBoss, aby využíval MySQL (viz 2.9.1) a založit databázi `jbossdb`.
3. Nastavit zabezpečení aplikace v serveru JBoss (viz 2.9.2).
4. Nasadit aplikaci (viz 2.9).
5. Spustit server MySQL.
6. Spustit server JBoss.
7. Klientem pro správu uživatelů založit potřebné uživatele (např. pro správu katalogu).
8. Klientem pro správu katalogu naplnit katalog zboží.
9. V prohlížeči zadat URL http://server:port/eshop, kde server je např. `localhost` a port např. `8080`.