Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Architekturou řízený vývoj softwaru v Javě

Bc. Petr Prošek

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Katedra informačního inženýrství

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Petr Prošek

Informatika

Název práce

Architekturou řízený vývoj softwaru v Javě

Název anglicky

Architecture-driven software develpoment in Java

Cíle práce

Cílem diplomové práce je představení architekturou řízeného vývoje aplikací v jazyce Java. Práce popíše základní architektury softwaru, tj. jednovrstvá, dvouvrstvá a více vrstvá (typicky třívrstvá) architektura (JEE). Práce popíše možnosti využití rozhraní v programovacím jazyce Java, jeho implementace a použití. Popíše návrhový vzor Model View Controller a prakticky jej implementuje ve vzorové aplikaci. Práce také zhodnotí možnosti vývoje softwaru pomocí architektury Black Box.

V praktické části autor vytvoří jednovrstvou aplikaci v programovacím prostředí Eclipse. Aplikace bude využívat MVC návrhový vzor a architekturu Black Box. Na závěr autor zhodnotí výhody použité Black Box architektury a navrhne případně varianty jejího vylepšení.

Metodika

- Literární rešerše, studium zdrojů o programovacích architekturách a nástrojích.
- Vytvoření programu na základě připravených zdrojů v modelovacím prostředí Eclipse.
- Testování vytvořeného programu.
- Zhodnocení a závěr.

Doporučený rozsah práce

60-80 stran

Doporučené zdroje informací

Pecinovský, Rudolf. 2007. Návrhové vzory. Brno : Computer Press, a.s., 2007. ISBN 978-80-251-1582-4.
Pecinovský, Rudolf. 2012. Java 7 – učebnice objektové architectury pro začátečníky. Grada Publishing, a.s., 2012. ISBN 978-80-247-3665-5
Pecinovský, Rudolf. 2014. Java 8 – úvod do objektové architectury pro mírně pokročilé. Grada Publishing, a.s., 2014. ISBN 978-80-247-4638-8

Předběžný termín obhajoby

2015/06 (červen)

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Elektronicky schváleno dne 30. 3. 2015

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 30. 03. 2015

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Architekturou řízený vývoj softwaru v Javě“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne _____

Poděkování

Rád bych poděkoval doc. Ing. Vojtěchovi Merunkovi, Ph.D. za odborné vedení a rady. Dále bych rád poděkoval členům katedry informačního inženýrství za ochotu, odborné rady a čas, který mi během přípravy diplomové práce věnovali v době hospitalizace vedoucího práce. A také bych rád poděkoval celé své rodině za podporu během studia.

Architekturou řízený vývoj softwaru v Javě

Architecture driven software development in Java

Souhrn

Diplomová práce obsahuje představení architekturou řízeného vývoje aplikací v jazyce Java. Popisuje základní architektury softwaru, tj. jednovrstvá, dvouvrstvá a více vrstvá (typicky třívrstvá) architektura (JEE). Zobrazuje možnosti využití rozhraní v programovacím jazyce Java, jeho implementace a použití. Popisuje návrhový vzor Model View Controller a prakticky jej implementuje ve vzorové aplikaci. Zhodnocuje možnosti vývoje softwaru pomocí architektury Black Box.

V praktické části je popsána jednovrstvá aplikace, kterou autor vytvořil jako součást diplomové práce. V této části jsou popsány použité postupy a architektury při tvorbě aplikace a závěrečná diskuse, kde autor poukazuje na nedostatky a možná vylepšení programovacích postupů programovacího nástroje Java a jejích komponent.

Klíčová slova

Java, Objektově orientované programování, Architektury softwaru, Black Box programming.

Summary

Diploma thesis contains an introduction into architecture driven software development in java. Thesis describes basic software architectures, eg. one-layered two-layered a multi-layered (usually three-layered) architecture (JEE). This thesis also displays possible use of interface in programming language Java and its implementation. Explains design pattern Model-View-Controller, which is practically implemented in application. Thesis also contains evaluation and possibilities of Black Box software development.

Author has created a one-layered application which is described in practical part of the thesis. This part also explains techniques and architectures used for developing the application. And at the end is final discussion where author points out the missing functionality of programming language Java and Java components.

Keywords

Object oriented programming, Programming architectures, Black Box programming, Java.

Obsah

1 Úvod

S programovacím jazykem Java se autor seznámil na kurzech, které absolvoval v rámci studijního programu Erasmus na Institutu technologií v Irském Waterfordu. Java se autorovy jevila jako zajímavá oblast k rozvoji svých odborných znalostí s možností budoucího kariérního využití a proto se rozhodl „Architekturou řízený vývoj softwaru v Javě“ použít jako téma své diplomové práce.

Java je objektově orientovaný jazyk vycházející z programovacího jazyka C a je jedním z nejpoužívanějších programovacích jazyků v současnosti. Své popularity Java dosáhla díky své přenositelnosti, a také proto, že je distribuována jako opensource.

V první části práce jsou popsány teoretická východiska z oblasti přístupu řešení problémů, architektury programů a jejich druhy. Dále jsou zde charakterizovány základní návrhové vzory a programové architektury, jež jsou z nich odvozeny. Dále je zde vysvětlena současně velice oblíbená architektura konstrukce aplikací pomocí vzoru Model-View-Controller. Je zde objasněn pojem Black Box programming a další programovací postupy.

Další část práce popisuje vývoj softwaru prostřednictvím nejrozšířenější podnikové technologie Java Enterprise Edition. V této části jsou vysvětleny pojmy úzce související s problematikou tvorby aplikací Java EE. Na tuto část navazuje představení možných nadstaveb a rozšíření, které vývoj podnikových aplikací v Java EE velice usnadňují. Ať už se jedná o nadstavby komerční nebo opensourcové.

V praktické části je popsána jednovrstvá aplikace, kterou autor vytvořil jako součást diplomové práce. V této části jsou analyzovány použité postupy a architektury při tvorbě aplikace a závěrečná diskuse, kde autor poukazuje na nedostatky a možná vylepšení programovacích postupů programovacího nástroje Java a jejích komponent.

Pro vypracování teoretických východisek bylo využíváno aktuálních dostupných tuzemských a zahraničních knižních zdrojů doplněných o zdroje elektronické.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem diplomové práce je představení architekturou řízeného vývoje aplikací v jazyce Java. Práce popíše základní architektury softwaru, tj. jednovrstvá, dvouvrstvá a více vrstvá (typicky třívrstvá) architektura (JEE). Dále uvede možnosti využití rozhraní v programovacím jazyce Java, jeho implementace a použití. Charakterizuje návrhový vzor Model View Controller a prakticky jej implementuje ve vzorové aplikaci. Práce také zhodnotí možnosti vývoje softwaru pomocí architektury Black Box.

V praktické části autor vytvoří jednovrstvou aplikaci v programovacím prostředí Eclipse. Aplikace bude využívat MVC návrhový vzor a architekturu Black Box. Na závěr autor zhodnotí výhody použité Black Box architektury a navrhne případně varianty jejího vylepšení.

2.2 Metodika

Rešeršní část DP bude založena na analýze odborných a vědeckých dokumentů (zejména monografií) a následně budou získané poznatky prakticky využity k navrhnutí a konstrukci jednovrstvé aplikace v modelovací prostředí Eclipse. Aplikace bude popsána v praktické části spolu s použitými metodami a architekturami. Na závěr autor testuje a zhodnotí použité architektury a navrhne případně varianty jejího vylepšení.

3 Literární rešerše

3.1 Přístupy řešení

Rozklad na jednodušší problémy je vhodným postupem při řešení složitějších úloh. Nejsou-li dělené části triviální, je vhodné je dále rozkládat dokud se triviálními nestanou. Rozklad problému (úlohy) je označován jako dekompozice. Při řešení dekomponovaného problému je možno postupovat několika způsoby.

1. Postup Shora-dolů (angl.. Top-Down design) – při tomto postupu se nejprve navrhuje celkové řešení problému (úlohy), přičemž části, které zatím nejsou vyřešené, se zaslepí, aby nehlásily chybu v programu. Záslepka může mít nadefinovanou signaturu, ale nemusí mít nadefinovaný kontrakt, který lze řešit později.

    - Výhoda: Představa o celkovém řešení problému od jeho počátku.
    - Nevýhoda: Program není možné plně testovat od jeho začátku, je nutné počkat, dokud nebudou naprogramovány metody v jeho nejnižších vrstvách. (1)

2. Postup Zdola-nahoru (angl. Bottom-Up design) – Oproti první postupu se tento zabývá řešením nejnižších (triviálních) úloh, které může vzápětí otestovat. Dokončené a otestované částí se spojují do větších bloků atd., dokud není program hotov. Program se tedy řeší od nejnižších funkcí jako základů, ze kterých se postupuje do vyšších vrstev programu.

    - Výhoda: Funkční části již od počátku řešení úlohy.
    - Nevýhoda: Při přechodu na vyšší vrstvy často dochází k odhalení jistých problémů v návrhu, které budou vyžadovat přepracování nižších vrstev. Může se také stát, že některé nižší úlohy bude třeba vyřešit znovu a jiným způsobem, který bude odpovídat potřebám vyšší vrstvy. (1)

Je také možné postupovat způsobem, který bude kombinovat oba předchozí postupy. Zpočátku je využíváno postupu Zdola-nahoru a jsou tak vytvořeny funkční bloky. Otestované bloky tvoří základnu a poté je program navržen postupem Shora-dolů.

- Výhody: Tato kombinace se snaží eliminovat nevýhody obou přechozích postupů a přináší možnost disponovat funkčními bloky jako předčasnými výsledky. Bloky se dají použít jako záslepky v dalším stádiu postupu.
- Nevýhody: Je možné, že tyto metody se vzájemně nepotkají. (1)

3.2 Vrstvená architektura

Vrstvená architektura je způsob návrhu aplikace, kdy jsou operace nebo funkce rozděleny do vrstev. Jsou-li všechny programy a funkce realizovány v jedné vrstvě jedná se o architekturu Jednovrstvou (Monolitickou). Jsou-li funkce rozděleny do více vrstev, jedná se o architekturu Dvouvrstvou nebo vícevrstvou (typicky Třívrstvou) využívající současně klient/server architekturu. (2)

3.2.1 Jednovrstvá (Jednoúrovňová / Monolitická) architektura

Charakteristika

Je jeden z nejstarších způsobů jak navrhnout informační systém nebo aplikaci a zároveň je typický pro centralizované zpracování. V jednovrstvé architektuře jsou všechny funkce programu (komunikace s uživatelem, pořízení a validace vstupů, zpracování, vyhodnocení, uskladnění a zobrazení dat) vzájemně propleteny a program je spuštěn na jednom počítači.

Aplikace vytvořené touto architekturou nepoužívají síť, protože ji ke svému chodu nepotřebují. (2)

Hodnocení monolitického systému

Nevýhody

- Realizace větším počtem programátorů vyžaduje bezchybnou komunikaci a koordinaci.
- Systém je obtížně rozšiřitelný, protože není rozdělen na logické části a nemá pevně definované hranice mezi logickou částí aplikace a daty.

- Chod aplikace lze urychlit pouze posílením hardwaru, na kterém je aplikace spuštěna.
- Náklady na údržbu systému rostou úměrně s počtem instalací, protože systém není možné spravovat centrálně.
- Aplikace jsou obtížněji přenositelné z důvodu kompatibility s jinými operačními systémy.
- Je-li monolitická aplikace používána na více zařízeních najednou, data jsou vzájemně odlišná. Data se mohou duplicitně vyskytovat na více místech a mohou obsahovat různé hodnoty. Jejich synchronizace je velice obtížná, nákladná a většinou je obstarávána jiným softwarem. Data jsou konzistentní pouze bezprostředně po synchronizaci.
- Nejsou-li data synchronizována pravidelně, hrozí jejich ztráta v důsledku havárií, nehod nebo jiných přírodních katastrof. Zabezpečení integrity dat a fyzické zabezpečení je tedy značně nákladné.

Výhody

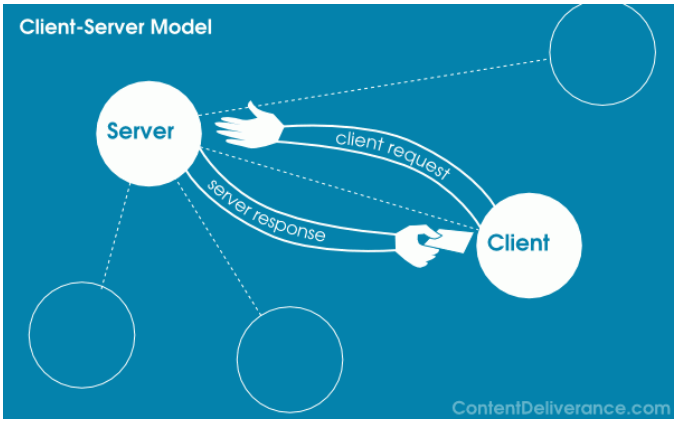
- Protože jednovrstvé systémy nepoužívají připojení k síti, je jejich zabezpečení značně jednodušší proti útokům zvenčí. Odpadají také starosti s ověřováním důvěrnosti komunikace a pozornost zabezpečení lze soustředit na zajištění integrity a dostupnosti dat.
- Absencí sítě také není potřeba řešit potíže při chybné synchronizaci, neaktuální verzi softwaru nebo absenci sdílených knihoven.

Jednovrstvá architektura je tedy vhodná pro tvorbu menších aplikací, které nepotřebují sdílet data nebo nemají přístup na síť. (2) (3)

3.2.2 Dvouvrstvá architektura (klient-server)

Charakteristika

Základním znakem je rozdělení funkcí aplikace na dvě vrstvy, klienta a serveru. Principem tohoto modelu je dotaz (požadavky) klienta odeslaný serveru, ten jej vyhodnotí, a pošle odpověď zpět.



Obrázek 1 - Klient-Server Model
[Zdroj: contentdeliverance.com]

Běžný model vyžití architektury pracuje tak, že klient se stará o komunikaci s uživatelem a server zpracovává klientovy požadavky a pracuje s daty. Tento model je nejčastější. Situace, kde by se server o data nestaral, nejsou příliš časté, ale rozvržení systému závisí na konkrétní implementaci.

Architektura se rozděluje na dvě varianty podle rozvržení funkcí (validaci, zpracování a vyhodnocení dat). (2) (3) (17)

Tlustý klient & Tenký server

Na straně klienta je umístěna většina logiky aplikace. Klient se stará o zpracování, vyhodnocení, zobrazení dat a také o komunikaci se serverem. Server se v tomto modelu stará o služby spojené se správou, organizací a přístupem k datům. Příkladem může být mail server nebo databázový server (komunikace skrze SQL).

Výhody

Tento model vyčleňuje vrstvu (server), která poskytuje a provádí služby. Jelikož je vrstva oddělena mohou její služby být nabídnuty i jiným aplikacím. Komfortní zpracování dat serverem. Data je možno centrálně zabezpečit na straně serveru. Ty jsou snadněji přenositelná než v případě jednovrstvé aplikace.

Nevýhody

Všechna data se před zpracováním přesouvají ke klientovi, což zatěžuje síť. Na klienta jsou také kladeny vyšší hardwarové nároky, neboť se všechna data zpracovávají u něj. Každá změna v systému aplikace vyžaduje aktualizaci u klienta, takže je nutné provést aktualizaci verze nebo přeinstalování aplikace na každé stanici s klientem. (2) (3)

Tenký klient & Tlustý server

Server kromě správy dat také provádí validaci a zpracování dat. Klient se stará o komunikaci se serverem a zobrazuje výsledky (odpovědi) serveru na dotazy, které klient zaslal. Většinou webový prohlížeč.

Výhody

Minimalizace pohybu dat po síti znamená rychlejší zpracování. Není třeba čekat, než se data ke zpracování přenesou ke klientovi, zobrazuje se pouze výsledek operace. Data jsou zpracovávána z jazyku serveru (databáze). Tento model umožňuje centralizovanou správu a zabezpečení dat. Změny v systému lze provádět bez zásahu na straně klienta. Aplikace jsou lépe přenositelné na další platformy.

Nevýhody

Nevýhodou je limitovaný výběr vývojových nástrojů, funkce musí být nadefinované v jazyce kompatibilním s databází. Zvýšení výkonu vyžaduje vylepšení nebo rozšíření hardwaru na straně serveru. (2) (3)

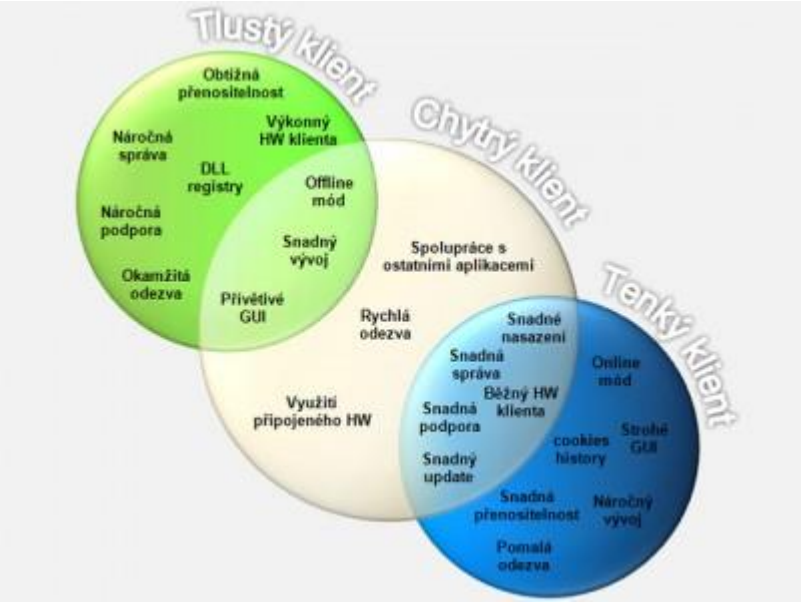
Chytrý klient (Smart client)

Chytrý klient potlačuje nevýhody tlustého a tenkého klienta a kombinuje jejich výhody. Obsahuje jisté funkce a uchovává data, takže dokáže pracovat v režimu off-line.

Po navázání spojení se data synchronizují. Využívá vlastní systémové zdroje (operační paměť, procesor nebo prostor na disku), ale dokáže komunikovat a využívat zdroje externě připojených zařízení a také již nainstalovaných aplikací (MS Office). Pro svou práci předpokládá existenci JRE[1] nebo .NET Framework[2], záleží na povaze aplikace.

Má-li klient přístup k síti, tak po svém spuštění zkontroluje, zda existuje novější verze aplikace a pokud ano, tak ji stáhne a spustí. Nalezne-li klient novější verzi aplikace v době, kde je používán, upozorní uživatele, zda si přeje aplikaci aktualizovat. Pokud ne, klient se aktualizuje při dalším spuštění. Opět záleží na konkrétní implementaci.

Automatické aktualizace v momentě připojení k serveru a možnost pracovat off-line jsou považovány za přední vlastnosti chytrého klienta. (2) (3)



Obrázek 2 – Druhy a vlastnosti Klient-Server modelů
[Zdroj: cleverandsmart.cz]

1 **JRE** (Java Runtime Environment) nebo také Java Runtime, je součástí JDK (Java Development Kit). JDK je set programovacích nástrojů pro vývoj aplikací v Javě. (6)

2 Základní komponentou je Microsoft **.NET** Framework, prostředí potřebné pro běh aplikací a nabízející jak spouštěcí rozhraní, tak potřebné knihovny. (6)

3.2.3 Vícevrstvá architektura

Bývá také označována jako multi-tier nebo n-tier, kdy n je počet použitých vrstev. Funkčnost celé aplikace je rozdělena do několika spolupracujících vrstev, které spolu komunikují přes definované rozhraní, proto vícevrstvá architektura.

Třívrstvá architektura bývá nejběžnějším příkladem vícevrstvé architektury. A nejčastěji jí využívají webové aplikace, které se dělí (do vrstev) na uživatelské rozhraní, databázi a logickou část aplikace. (2)

Rozdíl mezi Tier a Layer

Tyto dva pojmy se ve vícevrstvé architektuře vyskytují často. Překlad výrazů se příliš nepoužívá, spíše se drží anglických termínů. Hovoří-li se o Tier, jedná se o fyzickou (HW) vrstvu, zatímco Layer znamená vrstvu (SW) logickou.

Horizontální a vertikální přístup

Vertikální přístup znamená, že je dopředu přiřazen fyzický stroj (tier) k vykonání logického úkolu (layer). Horizontální přístup posiluje výkon daného tieru (např. přidáním serveru do clusteru[3]), čímž je vyrovnána více zatížená vrstva. Kombinace přístupů se označuje jako diagonální. (2)

Počet vrstev

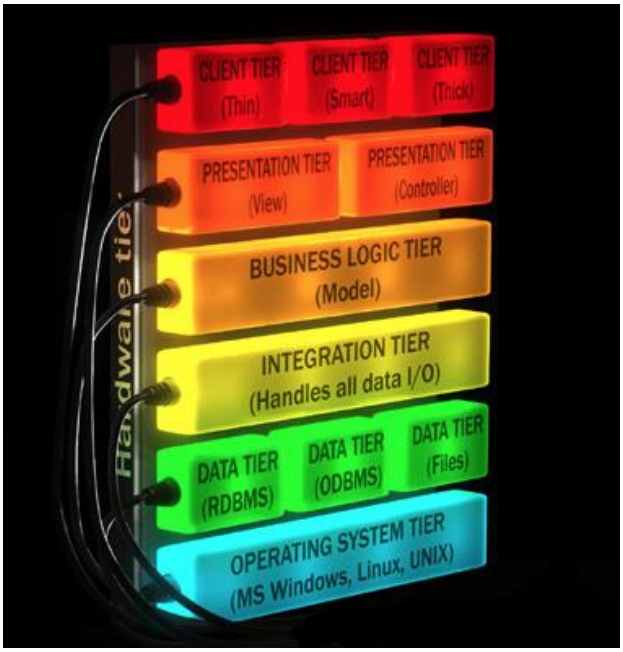
Počet vrstev, který by byl optimální, nelze stanovit. Závisí to na konkrétním použití a celkovém řešení architektury a potřebách uživatelů. Přidání vrstvy by mohlo snížit efektivitu celého řešení. Je-li aplikace navrhnuta tak, aby komunikace několika logických členů (layerů) probíhala v jednom serveru (tieru), je vše v pořádku. Ale přidáním dalšího serveru a rozdělením logických členů na několik serverů by mohla vzniknout přebytečná mezi-serverová komunikace. Komunikují-li logické členy v rámci jednoho serveru, mohou

3 Serverový **cluster** je skupina nezávislých počítačových systémů, pro které je používán termín uzly, ve kterých je spuštěn operační systém a které společně fungují jako jeden systém s cílem zajistit klientům dostupnost zvláště důležitých aplikací a prostředků. Uzly clusteru spolu neustále komunikují pomocí periodických zpráv nazývaných prezenční signály. Přestane-li být některý z uzlů clusteru k dispozici v důsledku selhání či údržby, začne službu okamžitě poskytovat jiný uzel (proces je známý jako převzetí služeb při selhání). (6)

využívat chatty interface („ukecaný“). Jsou-li logické členy rozděleny na několik serverů, jejich komunikace může být zajištěna pomocí chunky interface („sporý“). (2)

Vrstvy architektury

Počet vrstev a jejich pojmenování není jednotné. Následující model je založen na 5 vrstvé architektuře webové aplikace.



Obrázek 3 - Model vícevrstvé architektury
[Zdroj: cleverandsmart.cz]

Vrstva klienta (Client tier)

Tato vrstva bývá označována jako GUI vrstva a komunikuje s prezentační vrstvou. Jedná se o hardwarové zařízení většinou mobilní telefon, notebook nebo stolní počítač. Snahou developerů je vyvíjet moderní aplikace tak, aby výsledná aplikace byla pokud možno nezávislá na tom, na jaké platformě je spuštěna. (2)

Klienti jsou rozlišováni běžně na tři typy:

- Tenký (Thin) – většinou internetový prohlížeč, do kterého je aplikace po zadání URL stažena ve formě (X)HTML stránky. Může využívat úpravu vzhledu pomocí CSS a pro správný chod může být vyžadováno použití např. JavaScriptu.
- Tlustý (Thick) - aplikace, která je nainstalována na zařízení.
- Chytrý (Smart) - nachází se někde mezi tenkým a tlustým klientem a často využívá Java, ActiveX, Flash či Silverlight.

Prezentační vrstva (Presentation tier)

Někdy také nazývána jako web tier. Pokud byl jako architektonický vzor použit MVC (Model-View-Controller), obsahuje dvě vrstvy, a to Controller a View. Nejčastěji je provozována na webovém serveru, jako například IIS nebo Apache. Prezentační vrstva zodpovídá za poskytnutí statického obsahu (HTML, CSS, JavaScript, obrázky, video nebo animace) na základě požadavků z vyšší vrstvy. Dále obstarává komunikaci s business vrstvou. Obsahuje-li požadavek generování dynamického obsahu nebo službu, je zaslán na aplikační server. Server prokazuje svou identitu klientovi serverovým certifikátem, který se často nachází v této vrstvě. Šifrované spojení SSL/TSL navazuje klient také s webovým serverem. Bývá jediná vrstva s veřejnou IP adresou a tedy i přímo dostupná z internetu. (2)

Business vrstva (Business tier)

Nazývaná také jako doménová a nachází se na aplikačním serveru (např. Tomcat, GlassFish nebo Websphere). Pokud byl jako architektonický vzor použit MVC (Model-View-Controller) obsahuje Model. V této vrstvě bývá prováděno vyhodnocování a výpočty, autentizace (authentication), autorizace (authorization) a úprava profilu uživatele, tedy personalizace (personalization). Kvůli rozložení zátěže mohou být aplikační servery zapojeny do clusterů.

Integrační vrstva (Integration tier)

Má za úkol zajistit přístup k datům pro business vrstvu, takže se vyšší vrstva nestará o to, jaká databáze byla použita. V případě změny databáze jsou veškeré úpravy a připojení prováděny v této vrstvě, takže business vrstva může zůstat nezměněna.

Komunikace mezi Business a Integrační vrstvou se nezmění a data se budou zapisovat a číst stále správně. (2)

Datová vrstva (Enterprise tier)

Označována za nejnižší vrstvu modelu. Obsahuje data v podobě objektové databáze (ODBMS), relační databáze (RDBMS) nebo prostého souboru (file). Může se jednat o MySQL, MS SQL, Oracle, ale i textové soubory ve formátu CSV, ASC, XML. Provádí příkazy zaslané od Integrační vrstvy jako například zápis, čtení a mazání. Má na starosti ukládání a poskytování dat. (2)

Vhodný software a hardware

Nevhodnou volbou operačního systému a strojů, může dojít k nepříjemnostem, protože v produkčním prostředí aplikace nefunguje stejně jako v testovacím prostředí dodavatele.

Každá vrstva musí mít svůj operační systém, který nemusí být ve všech vrstvách stejný. Může se lišit ve verzi nebo dokonce výrobcem. Nejčastěji se jedná o MS Windows, Unix a Linux.

Stroje, na kterých jsou operační systémy provozovány, mohou být také od různých výrobců nebo 32bitové i 64bitové.

Ideální je tedy dosáhnout takové kombinace hardwaru a softwaru v každé vrstvě, aby byl zajištěn bezproblémový a bezporuchový chod aplikace ve všech vrstvách. (2) (17)

3.3 Návrhové vzory

Návrhový vzor je doporučený postup řešení často se vyskytujících úkolů. Není knihovnou, nebo částí zdrojového kódu, která by se přímo vložila do programu. Jedná se tedy o popis řešení problému nebo šablonu, která muže být použita v různých situacích. Objektově orientované návrhové vzory typicky ukazují na vztahy a interakce mezi třídami a objekty, aniž by určovaly implementaci konkrétní třídy. Algoritmy nejsou považovány za návrhové vzory, protože řeší konkrétní problémy a ne však problémy návrhu.

Na návrhový vzor by nemělo být pohlíženo jako na osamocenou jednotku vědomostí, ale spíše jako na výsledek kombinace praktických a teoretických zkušeností. Příkladem mohou být návrhové vzory, které vycházejí z programovacího jazyka. Pro jejich použití musí být čtenář i autor velmi dobře seznámen s prostředím, na které je daný vzor použit.

Lze tedy tvrdit, že prakticky existují dva druhy návrhových vzorů, implicitní a explicitní. Implicitní jsou zatím nikde nepopsaná řešení odvíjející se od znalostí a zkušeností. Explicitní, kterých je značně méně, jsou zdokumentované zkušenosti při řešeních konkrétních problémů. Tato dokumentace umožňuje využití vytvořených postupů pro širokou veřejnost.

Návrhové vzory nepocházejí zcela ze softwarového inženýrství, jsou běžné v každodenním životě. K nejstarším a nejznámějším oborům využívající návrhových vzorů je architektura. (4) (5)

„Každý vzor popisuje problém, který se v našem prostředí neustále vyskytuje.

Potom popisuje jádro řešení daného problému tak, že nám umožňuje toto řešení

používat třeba milionkrát, aniž bychom to dělali dvakrát stejným způsobem.“

– [Alexander, Ishikawa, Silverstain, 2014]

Přestože se citát týkal oblasti architektury, je zřejmé, že myšlenka návrhových vzorů je aplikovatelná v mnohem širším spektru. Dále o návrhových vzorech jen z oblasti programování a Javy. (5) (4)

3.3.1 Základní typy

Creational Patterns (vytvářející)

Creational Patterns řeší problematiku související s tvorbou objektů v systému. Většinou přenechávají tvorbu podtřídám nebo jiným objektům. Tyto návrhové vzory popisují postup výběru třídy nového objektu a zajištění správného počtu objektů. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu.

- **Abstract Factory** (Abstraktní továrna) – Definuje rozhraní pro vytváření rodin objektů, které jsou na sobě závislé nebo spolu nějak souvisí bez určení konkrétní třídy.
- **Factory Method** (Tovární metoda) – Deklaruje rozhraní s metodou pro získání objektu. Rozhodnutí o konkrétním typu vráceného objektu však ponechává na svých potomcích, tj. na překrývajících verzích deklarované metody.
- **Builder** (Stavitel) – Odděluje tvorbu komplexu objektů od jejich reprezentace tak, aby stejný proces tvorby mohl být použit i pro jiné reprezentace.
- **Lazy Initialization** (Odložená inicializace) – Odkládá vytváření objektu, počítání hodnoty, nebo provádění nějakého procesu až do okamžiku, kdy je ho poprvé potřeba.
- **Object pool** (Fond, lidově bazén) – Umožňuje vyhnout se drahému vytváření a uvolňování zdrojů recyklováním objektů, které už se nepoužívají.
- **Prototype** (Prototyp, Klon) – Specifikuje druh objektů, které se mají vytvořit použitím prototypového objektu. Nové objekty se vytváří kopírováním toho prototypového.
- **Singleton** (Jedináček) – Je-li potřeba, aby měla třída maximálně jednu instanci. [Terminologie podle (5)]

Structural Patterns (strukturální)

Structural Patterns představují skupinu vzorů, která se zaměřuje na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Obecně popisují struktury objektů a tříd. Snahou je zpřehlednit systém a využít možností strukturalizace kódu.

- **Adapter** (Adaptér) – Je-li potřeba, aby spolu pracovaly dvě třídy, které nemají kompatibilní rozhraní. Adaptér převádí rozhraní jedné třídy na rozhraní druhé.
- **Bridge** (Most) – Oddělí abstrakci od implementace tak, aby se tyto dvě mohly libovolně lišit.

- **Composite** (Strom, Skladba) – Komponuje objekty do stromové struktury a umožňuje klientovi pracovat s jednotlivými i se složenými objekty stejným způsobem.
- **Decorator** (Dekorátor) – Lze jej použít v případě, že existují nějaké objekty, kterým je potřeba přidávat další funkce za běhu. Nový objekt si zachovává původní rozhraní.
- **Facade** (Fasáda) – Nabízí jednotné rozhraní k sadě v podsystému. Definuje rozhraní vyšší úrovně, které zjednodušuje použití podsystému.
- **Flyweight** (Muší váha) – Je vhodná pro použití v případě, že existuje příliš mnoho malých objektů, které jsou si velmi podobné.
- **Proxy** – Nabízí náhradu nebo zástupný objekt za nějaký jiný pro kontrolu přístupu k danému objektu. [Terminologie podle (5)]

Behavioral Patterns (chování)

Behavioral Patterns se zajímají o chování systému. Obecně popisují algoritmy nebo spolupráci objektů. Mohou být založeny na třídách nebo objektech. U tříd využívají při návrhu řešení především principu dědičnosti. V druhém přístupu je řešena spolupráce mezi objekty a skupinami objektů zajišťující dosažení požadovaného výsledku.

- **Chain of responsibility** (Zřetězení zodpovědnosti) – Umožňuje, aby zaslaný požadavek zpracoval jiný objekt než ten, kterému byl zadán. Doporučuje objekty řetězit tak, aby objekt, který není schopen požadavek zpracovat, mohl požadavek předat dalšímu objektu v řetězu.
- **Command** (Příkaz) – Zabalí metodu do objektu tak, že s ní pak lze pracovat jako s běžným objektem. To umožňuje dynamickou výměnu používaných metod za běhu programu a optimalizaci přizpůsobení programu požadavkům uživatele.
- **Interpreter** (Interpret) - Vytváří jazyk, což znamená definování gramatických pravidel a určení způsobu, jak vzniklý jazyk interpretovat. [Terminologie podle (5)]

- **Iterator** (Iterátor) – Nabízí způsob, jak přistupovat k elementům skupinového objektu postupně bez toho, aby byl vystaven vnitřní reprezentaci tohoto objektu.
- **Mediator** (Prostředník) – Odstraní vzájemné vazby mezi řadou navzájem komunikujících objektů tím, že zavede objekt, který bude prostředníkem mezi těmito objekty. Zruší tak přímou vzájemnou závislost komunikujících objektů a umožní upravovat chování každého z nich nezávisle na ostatních.
- **Memento** (Memento) – Zabezpečuje uchovávání stavů objektů, aby je bylo v případě potřeby možno uvést do původního stavu. Přitom zabezpečuje, aby při uchovávání stavu nebylo narušeno ukrytí implementace.
- **Null Object** (Prázdný objekt) – Jde o objekt, který má fungovat jako základní stav objektu, a který je v podstatě náhradou stavu null.
- **Observer** (Pozorovatel) – V případě, kdy je na jednom objektu závislých mnoho dalších objektů, poskytne tento vzor způsob, jak všem dát vědět, když se něco změní.
- **Servant** (Služebník) – Skupině tříd nabídne další funkčnost, aniž by zabudovával reakci na příslušnou zprávu do každé z nich. Služebník je třída, jejíž instance poskytují metody, které si vezmou potřebnou činnost na starost, přičemž objekty, s nimiž danou činnost vykonávají, přebírají jako parametry.
- **State** (Stav) – Umožňuje objektu měnit své chování, pokud se změní jeho vnitřní stav. Objekt se tváří, jako kdyby se stal instancí jiné třídy.
- **Strategy** (Strategie) – Definuje množinu vyměnitelných objektů řešících danou úlohu a umožňuje mezi nimi dynamicky přepínat.
- **Template method** (Šablonová metoda) – Definuje kostru toho, jak nějaký algoritmus funguje, s tím, že některé kroky nechává na potomcích. Umožňuje tak potomkům upravit určité kroky algoritmu bez toho, aby mohly měnit strukturu algoritmu.

- **Visitor** (Návštěvník) – Umožňuje definovat pro skupinu tříd nové operace jejich instancí, aniž by bylo nutno jakkoliv měnit kód těchto tříd. [Terminologie podle (5)]

3.3.2 Singleton (jedináček)

Je-li potřeba sdílet jednu instanci mezi bloky nebo objekty v programu bez toho, aby byla stále předávána v konstruktoru, nadefinuje se podle vzoru singleton. Příkladem je databázové připojení, kdy celý program pracuje s jedním připojením a bylo by nepraktické ho stále předávat.

Úkolem vzoru je to, aby instance dané třídy existovala pouze jednou.

- Implementací prázdného konstruktoru definovaného jako private, zamezí uživateli vytvářet další instance třídy.
- Dále je vytvořena běžná instanční proměnná, do které je přiřazena instance třídy, která bude sdílena v programu.
- Nyní třída vytvoří instanci sebe sama a tu uloží do statické proměnné.
- O instanci se takto stará třída a uživatel má přístup pouze k oné jedné vytvořené instanci a nemá možnost tvořit další.
- Instance je nastavena jako privátní a final.
- Nakonec je vytvořena veřejná metoda, která umožní přístup k instanci vně třídy.

Příklad použití vzoru Singleton v příloze 7.1.

Nevýhody

Při využívání více vláknových aplikací je možné, že při přepínání vláken bude konstruktor zavolán vícekrát a vytvoří více instancí, což vylučuje podstatu jedináčka. Předejít vytvoření více instancí při více vláknových aplikací je možno použitím locku (zámku).

Další problém nastává u serializovatelnosti jedináčka. Při načítání jedináčka ze streamu je nutné kontrolovat, jestli nějaký jedináček už neexistuje a případně ho doplnit metodou readSolve(). (5) (8) (9)

3.3.3 Messenger (přepravka)

Messenger je druh třídy, která umožňuje sloučit několik atributů do jednoho objektu k přepravě. Přenese tolik atributů, kolik je v ní nadefinováno. Instance messengeru jsou tzv. immutable objects (neměnné objekty).

- Atributy jsou definovány jako veřejné konstanty, které jsou nastaveny v konstruktoru. Po vytvoření je nelze změnit, což nám zajišťuje bezpečný přenos dat. Použité modifikátory: public final.
- Přepravka je považována za kontejner, tj. objekt určený k uložení jiných objektů.
- Pro přepravku lze definovat pouze konstruktor s parametrem pro každý atribut. Může mít však nadefinované i přístupové nebo jiné metody.

Messenger bude použit v případě, kdy je potřeba, aby metoda vracela více hodnot najednou. Vrátí sice stále jeden messenger, ale ten obsahuje požadované atributy. Nebo je-li potřeba předat více hodnot jedním parametrem.

Typické použití: puntík se souřadnicemi. Příklad použití v příloze 7.2. (9)

3.3.4 Utility class (knihovní třída)

- Utility class je třída, která má nadefinované všechny atributy a metody jako statické.
- Ve třídě je uveden konstruktor jako privátní, aby bylo zmezeno možnosti vytvoření instance.
- Dále je také znemožněno od třídy dědit, což je provedeno klíčovým slovem `final` v Javě nebo `sealed` v C#.

Příklad použití v příloze 7.3.

Metody z utility class budou tedy vždy volány staticky a objekty či proměnné, se kterými budou metody pracovat, jí musí být předány jako formální parametry. Typickým příkladem knihovní třídy v jazyce Java je třída Math. (9)

3.3.5 Static factory method (statická tovární metoda)

Static factory method je návrhový vzor využívaný při práci v týmech na rozsáhlých projektech. Není tomu však podmínkou. Zakládá se na principu privátního konstruktoru, ke kterému se nepřistupuje přímo, ale volá se prostřednictvím statických metod.

Zdánlivě složitý návrh přináší však jisté výhody.

- První výhoda je spíše estetického charakteru. Přináší jisté zlepšení přehlednosti tím, že je možné metodu pojmenovat, jak je potřeba na rozdíl od konstruktoru.
- Druhá výhoda umožňuje napsání statických metod přesně podle logických požadavků navrhovaného systému.

Jako příklad použití může sloužit jistý firemní systém, který bude evidovat domy. Domy mají spoustu různých vlastností (budou to tedy instanční atributy), ale tato fiktivní firma má v evidenci pouze jednopatrové a dvoupatrové domy. Bude-li tedy vytvářet instanci domu, nejprve rozhodne, jestli bude dům jednopatrový či dvoupatrový a ostatní atributy budou nastaveny dále individuálně.

Příklad použití v příloze 7.4.

Při dalším použití kódu bude možné, aby byly vytvářeny instance pouze jednoposchoďových a dvouposchoďových domů. Pokud by bylo potřeba rozšířit o další víceposchoďové domy, stačí nadefinovat příslušnou odpovídající metodu. (9)

3.3.6 Služebník (Servant)

„Návrhový vzor služebník je další ze vzorů, které jsou tak primitivní, že se do knihy GoF(Gang of Four) nedostaly. Jak jsem již říkal, začátečnické učebnice o návrhových vzorech nehovoří a pro pokročilé programátory je tato konstrukce příliš primitivní na to,

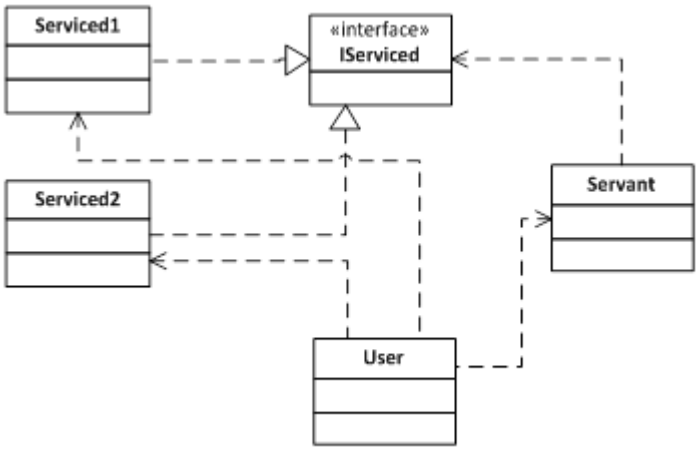
aby jí dávali jméno. Chtěl-li jsem se na tento vzor v učebnici dále odvolávat, musel jsem mu jméno vymyslet sám.“ [Pecinovský, 2014]

Tento návrhový vzor řeší problém, kde instance několika tříd současně potřebují nadefinovat nějakou společnou funkčnost, ale nemohou mít společného rodiče, kde by byla tato společná funkčnost nadefinována.

Existuje-li několik metod, které jsou definovány v několika objektech a mají funkci téměř stejnou, lze tyto metody zkopírovat a použít při tvorbě nového objektu.

Nicméně mezi důležité programátorské zásady patří vyvarovat se duplikování kódu. Je možné, že kopírovaný kód bude v tu chvíli nesprávný a jeho šířením je šířena také chyba. Nebo dojde-li ke změně zadání, bude muset programátor projít všechna místa, kde kód duplikoval a všechna je změnit podle nových požadavků. Duplikování je také potencionální zdroj nepříjemných chyb.

Jednou z metod předcházení duplikování kódu je právě definování služebníka.(10) (11)

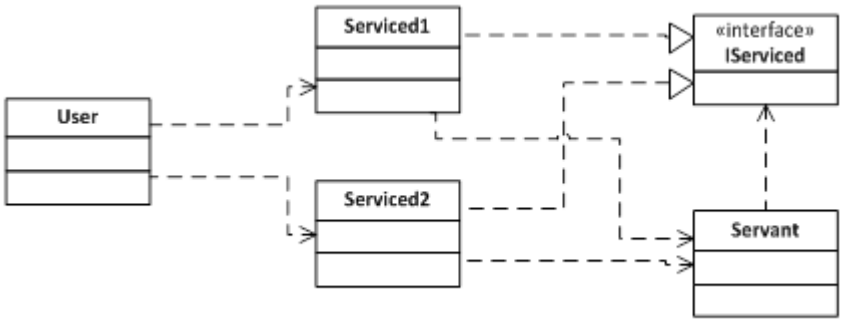


Obrázek 4 - Využití služebníka
[Zdroj: pecinovsky.cz]

Na obrázku 1 je služebník použit k zajištění nějaké funkcionality, kterou posílá spravovanému objektu jako parametr. (12)

Návrhový vzor *Služebník* je vhodný pro případy:

- Předcházení duplicitnímu kódu. Společná funkčnost je nadefinována v rozhraní. Toto rozhraní pak implementují všechny třídy, které chtějí onu společnou funkčnost využívat.
- Objekt má úkol, který by bylo příliš obtížné naprogramovat, ale řešení úkolu už je vytvořené někde jinde. Proto je úkol delegován služebníkovi, který se o řešení postará.
- Definice řešení, které bude dostatečné obecné, aby nezáleželo na tom, kdo jej bude v případě potřeby implementovat.



Obrázek 5 - Využití služebníka 2

[Zdroj: pecinovsky.cz]

Na obrázku 2 uživatel vyžádal operaci od spravované instance, která předala požadavek k vyřízení služebníkovi.

Implementace vzoru

Služebník musí komunikovat s obsluhovanými instancemi, aby dohlédl, že obsluhované instance dodržují veškeré podmínky, které jsou na ně kladeny.

Služebník:

Definuje rozhraní, kde deklaruje požadavky na obsluhované instance. Metody služebníka pak budou akceptovat instance tohoto rozhraní jako své parametry.

Obsluhovaná instance:

Musí implementovat příslušné rozhraní, aby se za jeho instanci mohla vydávat. Implementací tohoto rozhraní prohlašuje, že splňuje všechny podmínky vyžadované služebníkem, aby byla zajištěna plnohodnotná funkčnost. (10) (11) (17)

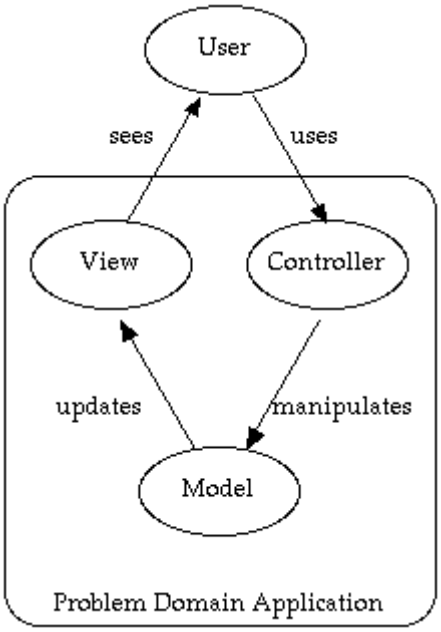
Pan Pecinovský ve své knize uvedl Implementaci vzoru služebník následovně:

1. *„Prvním krokem při návrhu služebníka je analýza toho, co má mít na starosti. Musíme si ujasnit, jaké metody musí služebník definovat a co budou tyto metody potřebovat od obsluhovaného parametru. Jinými slovy: co bude muset obsluhovaný parametry umět, aby metody mohly bezpečně splnit svůj úkol.*
2. *Druhým krokem je definice rozhraní, které deklaruje požadované vlastnosti obsluhovaného parametru, tj. vyjmenovává, na jaké zprávy bude muset umět reagovat (jinými slovy: jaké metody musí implementovat). Bude-li chtít nějaká instance využít služeb metod služebníka, bude muset implementovat toto rozhraní.*
3. *Třetím krokem je definice testů, které prověří, jestli následně definované metody služebníka dělají opravdu to, co mají.*
4. *Čtvrtým krokem je definice příslušného služebníka (a pokud možno jeho otestování).*
5. *Pátým krokem je implementace definovaného rozhraní obsluhovanými třídami, tj. třídami, s jejichž instancemi mají metody služebníka a/nebo jeho instancí pracovat (a opět jejich otestování).“* - [Pecinovský, 2008]

3.4 Model View Controller

Model View Controller (dále jen MVC) je oblíbeným způsobem návrhu aplikace. Bývá považován i za návrhový vzor. Jeho původu pochází ze Smalltalku, kde byl původně používán ke komponování tradičního postupu vstup – zpracování – výstup v GUI programech. Dále se hojně využívá ve vícevrstvých webových aplikacích a je také rozšířený v jazyce Java, jeho obecná definice jej ale umožňuje používat i v jakémkoliv jiném programovacím jazyce.

MVC rozděluje aplikaci na tři logické části: Model, View a Controller. Každá část má nadefinováno za co v aplikaci přesně zodpovídá. (14)



Obrázek 6 – MVC model
[Zdroj: http://cristobal.baray.com]

3.4.1 Model

Model reprezentuje data nebo aktivitu aplikace (databázová tabulka). Je nejčastěji implementována pomocí objektových třída a v nižší vrstvě je tvořen vybranou formou uložiště dat.

Jeho úkolem je poskytovat prostředky pro přístup k datům a stavům aplikace, její aktualizace a ukládání. Měl by představovat softwarový obraz procesu reálného světa. Jako celek by měl být zapouzdřený a měl by obsahovat přesně nadefinované rozhraní pro View a Controller. (14)

3.4.2 View

View má za úkol zobrazovat obsah Modelu, zajišťuje grafický či jiný výstup aplikace. Skrze Model má přístup k datům a stavům aplikace a specifikuje, jakým způsobem mají být prezentovány.

Je-li provedena změna stavu modelu, je potřeba, aby View zobrazení aktualizoval. Zaznamenávání změn stavů v modelu je získáváno Push nebo Pull přístupem. V případě Push přístupu jsou změny zaznamenávány modelem, který posílá notifikace o změnách do View. Pull přístup znamená, že View „vytahuje“ data z Modelu pokaždé, když potřebuje zobrazit nejaktuálnější data.

U webových aplikací se View stará o generování příslušného HTML kódu, který pak odešle prohlížeči k zobrazení. V případě stand-alone (offline) aplikací je aktualizace obsahu a překreslování zobrazených oken zajišťováno průběžně.

Je-li zobrazený výstup zároveň prostor pro zachycení událostí od uživatele, předává View tyto události (kliknutí na tlačítko) Controlleru. (14)

3.4.3 Controller

Controller kontroluje chování aplikace. Zpracovává veškeré vstupy a události pocházející od uživatele. Na základě těchto vstupů volá příslušné procesy Modelu, které mění jeho stav apod. Controller vybírá vhodné View na základě výsledku předchozích procesů v Modelu a podle událostí přijatých od uživatele.

Pro webové aplikace jsou hlavním vstupem HTTP GET nebo POST požadavky odeslané uživatelovým prohlížečem. U stand-alone (offline) aplikací je takovou událostí například kliknutí myší nebo vybrání položky menu. (14)

3.4.4 Výhody MVC

Mezi hlavní výhody MVC patří:

- Snadné zpřístupnění pro různé druhy klientů. Pro podporu dalšího klienta je nutné pouze nadefinovat nové View. V případě, že vstupní události by byly zcela odlišné, i speciální Controller. Model jako hlavní část aplikace může zůstat stále stejný.
- Použitím MCV se minimalizuje duplicita kódu. Bez oddělení a zapouzdření Modelu by pro každé nové View musela být znovu naprogramována celá aplikační logika. V rámci jednoho typu klienta bývá jedna akce volána z několika různých míst aplikace.

- Možnost vyvíjet odděleně. Jednotlivé části mohou být programovány samostatně, ale je nutné mít dopředu nadefinované rozhraní mezi jednotlivými částmi. Minimalizuje se tím dopad modifikací, změny jsou většinou prováděny jen v dané vrstvě.

- Znuvupoužitelnost kódu. Jako Model lze ve vlastní aplikaci použít standardní knihovny či třídy. Existují také univerzálně pojaté Controllery.

- MVC návrh je vysoce komplexní, snadno rozšiřitelný a modulární.

- V případě centralizovaného Front Controlleru, který zpracovává všechny přicházející požadavky, výhody vyplývají z centralizace, například možnost jednotné kontroly přístupových práv, logování apod.

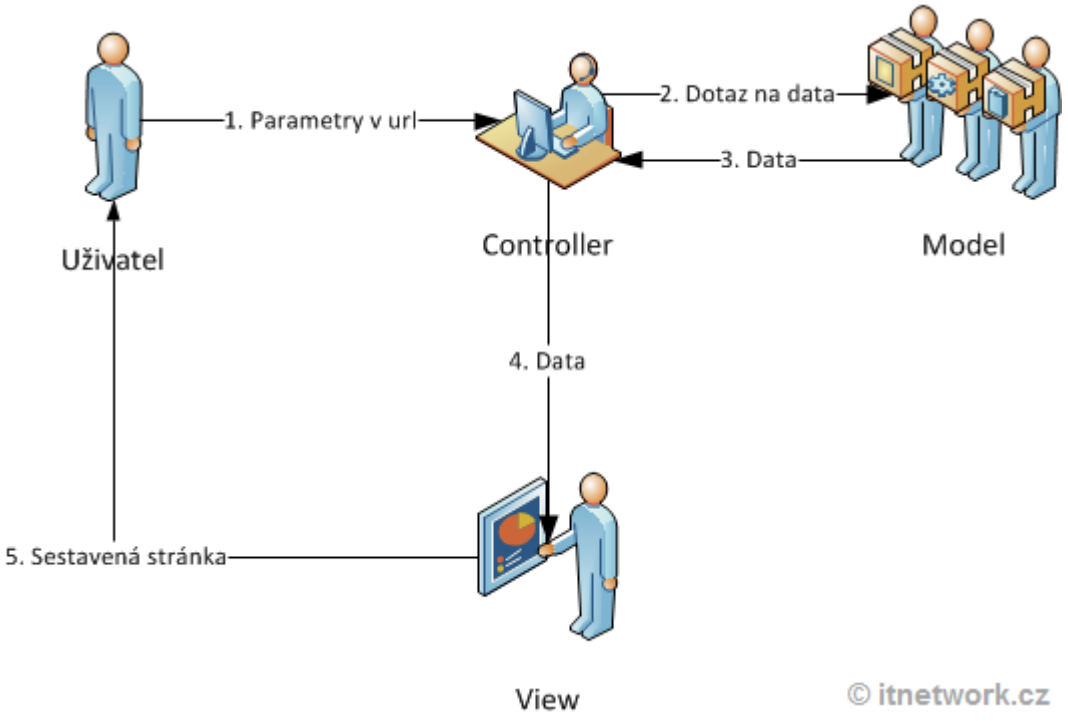
- Čistota designu, způsob přemýšlení nad aplikací, jejím návrhem a architekturou.

Postup při využití MVC rozlišuje mnoho strategií, jak výslednou aplikaci navrhnout. Jednotlivé strategie se mohou lišit například počtem Controllerů a rozdělením úkolů mezi nimi. Použitím či nepoužitím šablon v rámci View. Nebo zapouzdřeným voláním datových zdrojů apod. (13) (14)

3.4.5 MVC v praxi

Příkladem životního cyklu MVC je situace, kdy uživatel zadá do prohlížeče URL adresu a čeká na zobrazení stránky. Prohlížeč pošle požadavky na webový server, který určí, jaký controller bude požadavek vyřizovat na základně zaslaných parametrů.

Vybraný controller podle přijatých parametrů vykoná svou rutinu. Vyžádá si data z modelu. Dále může zavolat metody modelu, které si uloží jako proměnné. Nakonec vytvoří View (pohled), kterému předá příslušná data. View přijme data od controlleru, naplní jimi připravenou šablonu a vše odešle k zobrazení zpět prohlížeči. (14)



Obrázek 7 – MVC komunikace

[Zdroj: itnetwork.cz]

3.5 Přístupové metody

Většina atributů bývá programována jako soukromé private. Ke čtení nebo změně těchto atributů bude potřeba nadefinovat zvláštní přístupové metody. Tyto metody se dělí na čtecí, nebo také bývají označovány jako getter, a zapisovací, kterým se říká setter. Vytváření těchto metod bývá také považováno za návrhový vzor. (1)

Výhody tohoto přístupu:

- Je možné se postarat o to, aby hodnoty atributů bylo možné pouze číst, nikoliv měnit.

```
public String getNázev(){
      return název;
}[Zdroj: Autor]
```

- Bude-li třeba některé z atributů měnit, definuje se přístupová metoda označovaná jako setter.

```
public void setJméno(String novéJméno){
      staréJméno.set (novéJméno);
}[Zdroj: Autor]
```

Setter také umožňuje kontrolu správnosti vstupních parametrů.

```
public void setVěk(int novýVěk){
      if (novýVěk < 0)
            System.out.println("Neplatný věk.");
      else
            věk = novýVěk;
}[Zdroj: Autor]
```

- Při použití pro nastavování hodnot atributů lze provést doprovodné akce. Například překreslení plátna nebo aktualizace hodnot v tabulce.
- Přístupové metody je možné dát k dispozici nezávisle na tom, jestli jsou atributy v metodě dopočítány nebo jestli jsou to skutečné hodnoty.

Bude-li třeba později změnit přepočítávané atributy na originální nebo naopak celková funkčnost by neměla být vůbec ovlivněna, protože o vše se stará tělo metody. (1)

3.5.1 Konvence pro názvy přístupových metod

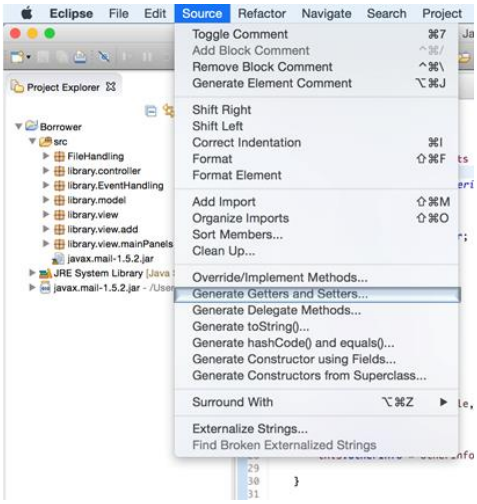
Přístupové metody mají svou zvláštní konvenci, kterou je vhodné dodržovat. Důvodem je jednodušší orientace programátorů, kteří budou program číst a na konvenci je postavena i funkčnost některých programů a technologií.

- Názvy metod jsou složené z předpony následované názvem atributu, přičemž název atributu začíná velkým písmenem i přesto, že je nadeklarovaný s písmenem malým.
- Nastavovací metody (settery) mají předponu **set.**
- Čtecí metody (gettery) používají předponu **get.** V případě, že vracejí logickou hodnotu (Boolean true/false), mohou mít předponu **is.** (1)

```
public boolean isPlnoletý(int věk){
	if(věk > 18)
		return true;
	else return false;
}[Zdroj: Autor]
```

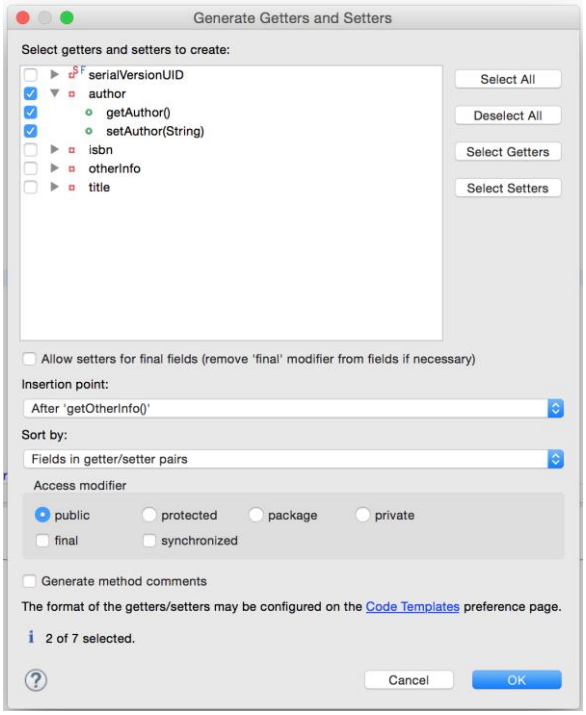
3.5.2 Využití vývojového prostředí Eclipse pro generování přístupových metod

Programátor nejprve musí nadeklarovat všechny potřebné atributy, aby mohl metody generovat.



Obrázek 8 - Generování přístupových metod v Eclipse

[Zdroj: Autor]



Obrázek 9 - Dialogové okno pro generování přístupových metod v Eclipse

[Zdroj: Autor]

Následně Eclipse zobrazí dialog pro nastavení generování přístupových metod. V horní části je obsažen seznam atributů deklarovaných v dané třídě. Každý atribut má zaškrtávací pole ke zvolení generování gettru, setteru nebo obou. Pod seznamem atributů lze dodatečně vybrat, kam budou metody vloženy, musí to však být v rámci zdrojového kódu jedné třídy. Dialogové okno je zobrazeno na obrázku 9.(8)

Po nastavení příslušných parametrů a potvrzení tlačítkem OK jsou metody vygenerovány.

```
public String getAuthor() {
	return author;
}

public void setAuthor(String author) {
	this.author = author;
}[Zdroj: Autor]
```

3.6 Black Box

Jedná se o způsob zapouzdření, které umožňuje některé metody a atributy skrýt tak, aby zůstaly viditelné a využitelné jen pro třídu zevnitř. Lze si poté objekt představit jako černou skříňku (Black Box), která má nadefinované určité rozhraní, přes které jí lze posílat zprávy, instrukce nebo data, které zpracovává.

Programu není známo, jak černá skříňka uvnitř funguje. Je pouze známo, jak se používá a jak se navenek chová. Není tedy možné způsobit nějakou chybu, protože je na venek vidět jen to, co programátor třídy zpřístupnil.

Toto zapouzdření tedy donucuje, aby byl objekt používán tím správným způsobem. Oddělení na veřejné funkce a vnitřní strukturu je prováděno modifikátory přístupnosti public a private.(15) (16)

Jako příklad lze nadefinovat třídu `Člověk`, která má nadefinovaný atribut `datumNarození`. Z tohoto atributu jsou odvozeny další atributy: `vek` a `plnoletost`. Pokusil-li by se někdo změnit `datumNarozeni` zvenčí, proměnné `plnoletost` a `vek` by přestali platit pro tuto třídu. Stav tohoto objektu se označuje za nekonzistentní. Běžný

případ ve strukturovaném programování. Nadefinuje-li se však atribut `datumNarozeni` jako private, bude pak zapouzdřený a viditelný pouze třídou `Člověk`. Poté je nadefinována metoda `setDatumNarození()`, která příjme nové datum a dosadí je do proměnné `datumNarozeni` a provede výpočet `věku` a vyhodnotí `plnoletost`. Použití objektu je pak mnohem stabilnější. (8) (15) (16)

3.6.1 Black Box a White Box testování

Black Box programovací architektura vychází z Black Box testování. Princip je prakticky stejný. Black Box testovací metoda považuje systém za černou skřínku a explicitně nevyužívá její vnitřní strukturu. Je prováděna speciálním testovacím týmem a nevyžaduje znalost programovacího jazyka. Je zaměřena na test vnější funkcionality a grafického rozhraní (GUI). Synonyma: behaviorální, funkcionální, neprůhledná (opaque-box) nebo zavřená (closed-box) skřínka.

White Box (bíla krabice) testování dovoluje „nahlédnout“ do krabice. Zaměřuje se převážně na testování vnitřní struktury softwaru a bezchybný průchod dat skrz strukturu. Tato metoda již vyžaduje jisté programátorské znalosti a je prováděna především vývojovým týmem aplikace. Je-li nalezena průchodem aplikace chyba, tým chybu opraví a provedou test znovu. Synonyma: strukturální, skleněná (glass-box) nebo čirá (clear-box) krabice. (15) (16)

Přestože Black Box a White Box testování jsou stále populární termíny, někteří programátoři dávají přednost termínům behaviorální nebo strukturální. I tak se definice behaviorálního testování poněkud liší od Black Box testování. Při behaviorálním testování není „nahlédnutí“ do krabice striktně zakázáno, jako je tomu u Black Box metody, ale přesto je to silně nedoporučováno. V praxi se neprokázalo výhodné používat pouze jednu testovací metodu. Je výhodnější použít více testovacích metod, aby testy nebyly omezeny charakteristikou jedné specifické metody. Tyto testy bývají nazývány jako šedé (grey-box) nebo průsvitné (translucent-box) krabice, ale vyvstává ohlas, že už je „překrabicováno“ a měl by se začít používat jiný termín.

Je důležité pochopit, že tyto testy bývají využívány v testovací fázi, a jejich viditelnosti s touto fází končí. Různé úrovně testování (jednotková, systémová, atd…)

vyžívají různé druhy testovacích metod. Jednotkové testování je většinou spojováno se strukturálním testováním, protože testeři většinou nemají dostatečně nadefinovanou úroveň požadavků na splnění testů. (15) (16)

3.7 Rozhraní

Rozhraní (interface) je množina metod, která může být implementována třídou. Rozhraní metody pouze popisuje, ale neobsahuje jejich vlastní implementace. Jako rozhraní třídy je chápána množina informací, které třída zveřejní. Čili vše co je ve třídě označené modifikátorem public. V rozhraní by měly být zahrnuty jenom ty konstruktory, atributy a metody, které jsou potřeba pro to, aby program opravdu viděl. Je-li veřejná viditelnost nežádoucí, nadefinuje se jako soukromá, a to modifikátorem private.

Obdobná definice říká, že rozhraní lze také chápat jako zvláštní druh třídy, která pouze odhaluje seznam veřejných nedefinovaných metod, na které budou jeho instance schopné odpovídat. (1) (17) (18)

3.7.1 Definice rozhraní

Hlavička je tvořena modifikátorem viditelnosti public, klíčového slova interface a jména rozhraní. Tělo rozhraní obsahuje pouze hlavičku metody ukončenou středníkem. Vlastní tělo metody už záleží na implementaci, o kterou se rozhraní nestará. Rozhraní tvoří Signatura a Kontrakt.

```
/*
 *  Zde se vyplňuje kontrakt.
 *  Podmínky a očekávané vstupy a výstupy
 */
public interface NázevRozhraní{
      public void metodaRozhraní();
}[Zdroj: Autor]
```

Signatura

Signaturu tvoří vlastnosti, které jsou kontrolovány překladačem (Názvy, typy, …).

Signaturu metody tvoří název a její parametry.

```
nazevMetody(String parametr1, int parametr2) [Zdroj: Autor]
```

Deklarace více metod se stejnou signaturou (stejným názvem a stejným počtem a typem parametrů) není možná. Kompilátor by nebyl schopen metody od sebe rozlišit. V jedné třídě však mohou existovat metody stejného jména jen v případě, že každá má jiný seznam parametrů.

Pojmenování metod

Jménem metody může být jakýkoli platný identifikátor. Konvence však doporučují vytvářet jména metody dle jistých pravidel. Názvem metody je malými písmeny psané sloveso nebo víceslovný název. Ve víceslovných názvech by mělo druhé a každé další slovo začínat velkým písmenem.

```
příkladVíceslovnéhoNázevuMetody() [Zdroj: Autor]
```

Metody v rámci jedné třídy mají většinou unikátní jméno. Nicméně třída může mít více metod se stejným názvem díky přetížení metod (parametrizace prázdné metody) tím, že metody budou mít odlišné parametry v signatuře. (1) (18)

Kontrakt

Za kontrakt se označuje ta část rozhraní, kterou překladač nedokáže zkontrolovat. Je popsán v dokumentačních komentářích a za jeho dodržení je zodpovědný programátor tím, že zajistí, aby při používání dané třídy či volání metody byly splněny jisté podmínky. Například můžeme vyžadovat minimální nebo maximální počet znaků v řetězci, jistý počet prvků v poli, nebo aby číselný parametr byl nezáporný a další.

Veškeré záležitosti týkající se kontraktu by měl uživatel uvést v dokumentaci třídy nebo metody, aby je uživatel nepoužil nesprávně. Nedodržení kontraktu může vést k nepředvídanému chování nebo k havárii celého programu. (1)

3.7.2 Implementace rozhraní

Implementace rozhraní způsob, jakým je rozhraní vytvořeno.
Provádí se v hlavičce třídy klíčovým slovem `implements`.

```
public class NázevTřídy implments NázevRozhraní{
}[Zdroj: Autor]
```

Třídy, které rozhraní implementovaly, se mohou vydávat za instance daného rozhraní. Avšak rozhraní instance mít nemůže, protože jeho metody jsou abstraktní a bez implementace.

Metody deklarované v rozhraní jsou abstraktní a jsou automaticky veřejné. Abstraktní proto, že neobsahují žádnou implementaci, obsahují pouze středník. Rozhraní jiné než veřejné metody obsahovat nesmí. Potíže by proto mohlo způsobit zapsání jiného modifikátoru než `public`. Rozhraní také dovoluje vynechání modifikátorů `abstract` a `public`.

Rozhraní kromě definic metod může také obsahovat statické konstanty. Chovají se pak stejně, jako by se jednalo o konstanty třídy, která rozhraní implementovala. Jsou také implicitně veřejné, statické a neměnné (`final`).

Jako lze dědičností rozšiřovat třídy je možné rozšiřovat i rozhraní. Vytvořené rozhraní, které dědí od jiného rozhraní, automaticky přebírá všechny jeho metody a konstanty. Dědičnost se zapisuje klíčovým slovem `extends` a je možné dědit od více rozhraní najednou. (1) (18)

3.7.3 Využití rozhraní

Zpracování parametrů různých typů lze dosáhnout definicí několika přetížených verzí jedné metody, a to takovým počtem, jako by bylo potřeba zpracovávat parametrů nebo definicí rozhraní deklarujícího objekt (parametr) tak, aby metoda mohla plnit svůj úkol.

Řešení přetíženými metodami lze použít pouze tehdy, pokud je dopředu znám počet typů parametrů, které má metoda zpracovávat. Řešení rozhraním je mnohem univerzálnější, protože metody lze použít na instance neomezeného počtu datových typu. Bude-li třeba zahrnout do seznamu spravovaný datových typů další třídu, je potřeba pouze naimplementovat správné rozhraní. Potom její instance začnou vydávat za parametry příslušných metod.

Další výhodou řešení použitím rozhraní je, že třída může naimplementovat libovolný počat rozhraní současně. Proto je vhodné nadefinovat pro různé účely různá rozhraní, která by třída naimplementovala jako služebníky (Návrhový vzor Služebník). (1) (11) (18)

3.8 Java EE

Představuje technologii Java Enterprise Edition, zkráceně se označuje jako JEE nebo J2EE. Používá se k tvorbě webových/podnikových aplikací v Javě. Pro práci s JEE se předpokládá dobrá znalost Javy SE (Standard Edition) a základní znalost HTML a CSS.

Java EE je v praxi nejrozšířenější podniková technologie, kterou používá mnoho velkých firem. Výsledná aplikace obsahuje velké množství různých pokročilých technologií a obsahuje spoustu hotových řešení použitelných pro rozsáhlé webové aplikace. Příkladem mohou být státní registry a různé bankovní aplikace.

Java EE je sada knihoven vložených do Javy SE, nejedná se o jinou verzi Javy. Programování v Java EE tedy probíhá naprosto stejně, jako v Javě SE.

Za podnikovou aplikaci je považována aplikace, která:

- Zvládne obsloužit velké množství uživatelů najednou
- Zpracovává velké množství dat v databázi
- Komunikuje s dalšími systémy
- Je bezpečná a spolehlivá

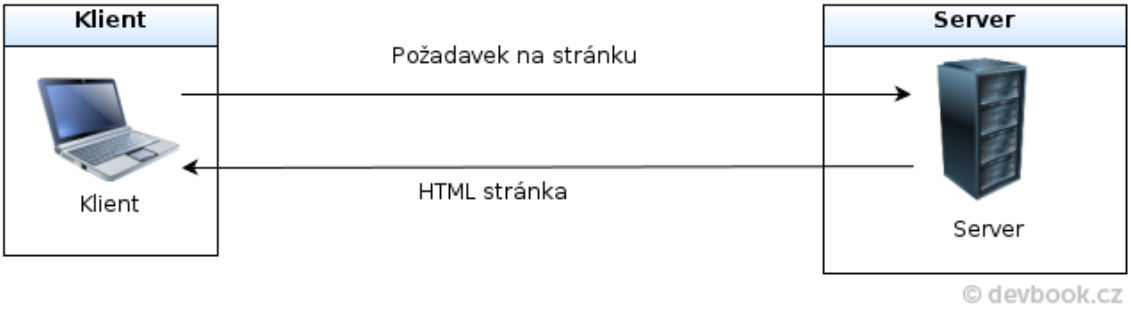
Java Enterprise Edition poskytuje velké množství kvalitních komponent pro usnadnění tvorby právě podnikových aplikací. Následuje třívrstvou architekturu, která rozděluje aplikaci na tři vrstvy. Vrstvu obsahující databázi, vrstvu s obchodní logikou a vrstvu prezentační. (22)

3.8.1 Webové aplikace JEE

Na rozdíl od Javy SE, kdy aplikace byla provozována na straně klienta. Java EE běží na straně serveru. Výstupem aplikace vytvořené v Java EE je tedy HTML stránka.

Statický web (bez použití Java EE)

Jedná se o HTML stánky, které jsou uloženy na serveru. Klient (uživatel s prohlížečem) zasílá požadavek na server, který klientovi vrací přesně ty stránky, které má uložené. (22)



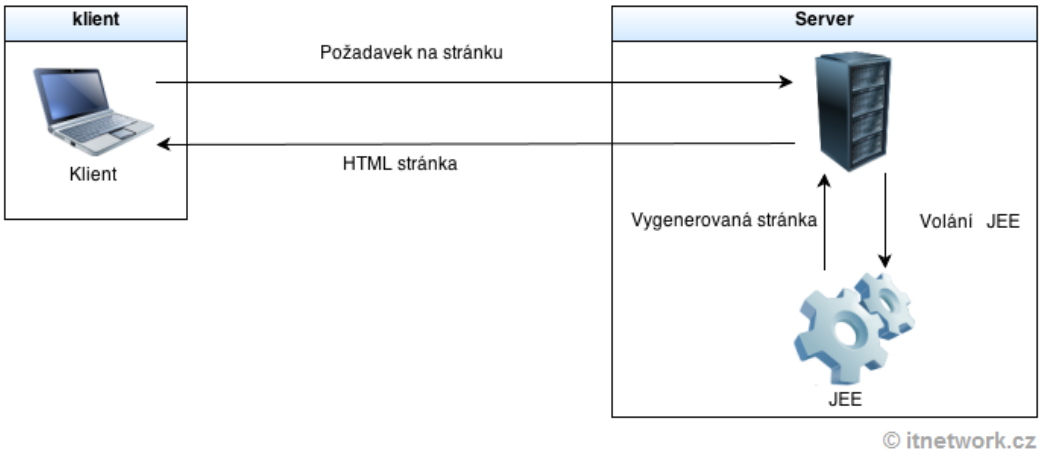
Obrázek 10 - Komunikace statického webu

[Zdroj: Devbook.cz]

Dynamický web (s použitím Java EE)

Statické weby se těžko spravují a jsou omezené. To zapříčilo vznik serverových jazyků, umožňujících pozměnit HTML stránku předtím, než je odeslána klientovi. Umožňuje tvorbu diskusních fór, nahrávání obsahu pomocí editoru, přihlašování uživatelů a další aktivní prvky.

Java EE je spuštěna na serveru, kde na základě požadavků klienta generuje obsah webové stránky. Generovaný obsah je načten z databáze, která běží také na serveru. Výsledné vygenerované HTML se zobrazí u klienta. (22)



Obrázek 11 - Komunikace statického webu

[Zdroj: ITnetwork.cz]

3.8.2 Aplikační servery

Java EE je provozována v rámci aplikačního serveru.

Aplikační server je fakticky Java proces, který běží na fyzickém serveru a zpracovává požadavky klientů (dle standardu JEE), odesílá emaily a zajišťuje spojení s databází a jiné. V prostředí aplikačního serveru je provozován Java EE, které zde provádí své operace. (22)

Některé aplikační servery:

- JBoss Application Server – Známější, opensource.
- GlassFish - Jednoduchý, od společnosti Oracle.
- WebSphere Application Server – Komerční, pro velké aplikace, od společnosti IBM.
- WebLogic Server – Komerční, od společnosti Oracle.

Komerční řešení poskytují maximální výkon, stabilitu a podporu. Používají i vlastní Java Virtual Machine[4]. Některé open source servery umožňují dokoupit komerční podporu.

[4] **Java Virtual Machine** (JVM) je sada počítačových programů, která využívá modul virtuálního stroje ke spuštění dalších počítačových programů a skriptů vytvořených v jazyce Java. (6)

3.8.3 Technologie obsažené v Java EE

Java EE obsahuje spoustu pokročilých technologií, některé z nich jsou:

- JSP (Java Server Pages) - Technologie umožňuje vkládat speciální direktivu do HTML kódu, která spustí Java kód. Na daná místa ve stránce se tak vloží data, která získala Java např. z databáze.
- JSF (Java Server Faces) - Konkurenční a modernější technologie k JSP. Celá webová stránka je reprezentována jako XML soubor. Web se skládá z již připravených komponent (formuláře, tabulky, seznamy), které lze jednoduše plnit daty z Javy.
- JDBC (Java DataBase Connectivity) - JDBC je standardní rozhraní pro práci s různými typy databází v jejich jazyce SQL.
- JPA (Java Persistence API) - JPA je rozhraní, umožňující objektovou práci s daty. S databází nekomunikuje přímo v SQL, ale pomocí mezivrstvy ORM. Pracuje tedy pouze s objekty. Konkrétní implementací je Hibernate[5].
- EJB (Enterprise Java Beans) - Komponenty obchodní logiky.
- Spring framework - Kromě standardních řešení z Javy Enterprise Edition se také uchytilo několik frameworků třetích stran. Mezi nejznámější patří Spring. Jedná se o konkurenci např. k JSF.

Je patrné, že různými přístupy lze dojít ke stejnému řešení, nehledě na to, která konkurenční technologie bude použita. Záleží na tom, kterou technologii vývojář používá nebo na tom, kterou si zákazník specifikuje v požadavcích. (19) (22)

5 **Hibernate** je Framework napsaný v jazyce Java, který umožňuje tzv. objektově-relační mapování (ORM). Usnadňuje řešení otázky zachování dat objektů i po ukončení běhu aplikace. Je jednou z implementací Java Persistence API (JPA). (6)

3.9 JDBC (Java Database Connectivity)

Poskytuje Java programu rozhraní pro přístup k relační databázi a dovoluje vykonávat SQL příkazy a komunikovat s databází. Řešení relačním způsobem JDBC není vždy dostačující, je-li potřeba přistupovat k datům jako k objektům, ORM je vhodným řešením. (20)

3.9.1 ORM (Object-Relational Mapping)

Jedná se o programovací techniku pro konvertování dat mezi relační databází a objektově orientovaným programovacím jazykem (Java, C# atd.).

Oproti běžnému JDBC má následující výhody:

- Business logika přistupuje k datům jako k objektům a ne jako DB tabulce.
- Skryje podrobnosti SQL dotazů pro objektově orientovanou logiku.
- Není třeba se zabývat implementací databáze.
- Entity založené na business konceptu než na databázové struktuře.
- Tansakční management a automatické generování klíčů.
- Rychlejší vývoj aplikace.

Využití ORM nabízí tyto služby:

- Poskytuje rozhraní pro provádění CRUD (Create, Read, Update and Delete) operací nad persistentními objekty.
- Jazyk nebo rozhraní, které specifikuje dotazy pro přístup k objektům nebo jejich obsahu.
- Konfigurovatelný nástroj pro mapování metadat.
- Techniky přístupu k transakčním objektům k provedení operací (dirty checking, lazy association fetching a další optimalizační funkce)

Persistance Framework je ORM servis, který ukládá a přenáší objekty do relační databáze. (20)

3.10 Spring Framework

Motivací pro vývoj Spring Frameworku je snaha o usnadnění v oblasti vývoje Java EE aplikací. Spring Framework integruje řadu existujících opensource nástrojů, které jsou funkční a prověřené. Umožňuje zaměřit se na architekturu místo na technologii, neinvazivnost a modulárnost. Nezabývá se tak řešením již vyřešených problémů. Spring umožňuje využít třeba jen část, která se zrovna hodí k řešení daného problému.

Spring Framework může používat libovolná Java aplikace. Stal se velice populární v komunitě Java. Může být alternativou k Enterprise Java Beans (EJB), nebo jako jeho nadstavbou. (21)

Obrázek 12 - Logo Spring
[Zdroj: Spring.io]

Usnadnění vývoje spočívá v:

- Pomoc při odstranění těsných programových vazeb jednotlivých POJO objektů a vrstev za pomoci návrhového vzoru Inversion of Control.
- Možnost volby implementace (EJB, POJO) business vrstvy pro aplikační architekturu a ne naopak (tedy aby architektura předepisovala implementaci).
- Řešení různých aplikačních domén bez nutnosti použití EJB, například transakční zpracování.
- Podpora implementace komponent pro přístup k datům, ať již formou přímého JDBC či ORM (object-relation mapping) technologií a nástrojů jako Hibernate.

- Abstrakce vedoucí ke zjednodušenému používání dalších části J2EE, jako například JMS, JavaMail, JDBC.
- Usnadnění používání a psaní unit testů.
- Správa a konfigurace business komponent. (21)

3.10.1 Vzor Inversion of Control

Jádro Springu je postaveno na využití návrhového vzoru Inversion of Control (zkratkou IoC). Vzor Inversion of Control funguje tak, že odpovědnost za vytvoření a provázání objektů je přesunuta z aplikace na Framework. Objekty mají pouze povinnost poskytnout rozhraní pro vložení potřebných objektů z vnějšku.

Dependency Injection

Dependency Injection je speciálním případem Inversion of Control a řeší tři způsoby vložení objektů.

- **Setter injection** - Vložení pomocí setteru, čili setovacích metod, které poskytuje Spring. Nevýhoda je, že je možné vytvořit instanci třídy bez nastavení závislostí.
- **Constructor injection** - Vložení přes konstruktor, používá ji framework PicoKontejner, které poskytuje Spring. Dovoluje vyžadovat, aby byla implementace rozhraní nastavena při konstrukci vlastního objektu.
- **Interface injection** - Vložení přes definované rozhraní, typickým představitelem je framework Avalon. (21)

3.10.2 POJO

Termín POJO je zkratkou pro Plain Old Java Objects a formulovali jej Martin Fowler, Rebbecca Parsons a Josh MacKenzie na konferenci v roce 2000.

Volný překlad z blogu Martina Flowera:

„Poukazovali jsme na mnoho výhod, které přináší zakódování business logiky do obyčejných objektů, namísto použití Entity Beans. Zajímalo nás, proč jsou lidé proti

používání běžných Java objektů v jejich systémech a dospěli jsme k tomu, že nemají žádné přepychové jméno. Tak jsme jim jedno dali a docela se chytlo.“ – [Martin Flower, -]

Označuje objekty, které nejsou svázané technologicky specifickým rozhraním. Jediné rozhraní těchto objektů je definováno business službami, které poskytují. (21)

3.11 Enterprise Java Beans

3.11.1 Co je to Java Bean?

Jedná se o třídu, která následuje určité standardy nebo konvence. Je třeba je dodržet při návrhu aplikace, protože na jejím dodržení závisí funkčnost spousta potřebných knihoven.

- Všechny instanční proměnné musí být `private` a přístup k nim musí zajišťovat přístupové metody tedy getty a settry.
- Musí mít veřejný bezstavový konstruktor.
- Musí implementovat Serializable.

Implementováním java.io.Serializable interface umožňuje instance serializovat (přeměnit je na stream bitů), lze je poté ukládat do souborů nebo objektových databází a posílat po síti. A poté zpětně deserializovat tedy načíst.

Mezi normální třídou a Java Bean není syntaktický rozdíl, ale třídu lze označit za Java bean pouze tehdy, pokud dodržuje výše definované konvence. (13) (19) (22)

3.11.2 Enterprise Bean

Je programovacím jazykem Java napsaná komponenta (Třída) na straně serveru, která zapouzdřuje business logiku aplikace. Business logika je jádrem aplikace.

Například aplikace kontrolující stav skladu. Enterprise Bean může implementovat business logiku v metodách `getStavSkladu` nebo `objednejZbozi`. Voláním těchto metod může klient využívat služby skladu prostřednictvím aplikace.

Z několika důvodů zjednodušují vývoj rozsáhlých distribuovaných aplikací. EJB kontejner poskytuje system-level služby (komunikace s klientem, management transakcí,

bezpečnostní funkce) pro Enterprise Bean. Tudíž se Enterprise Bean vývojář může zaměřit čistě na business logiku.

Další výhodou je, že business logiku neobsahuje klient ale Entrprise Bean, tudíž se vývojář může zaměřit pouze na prezentaci v klientovi. Klient developer se tedy nemusí zabývat implementací business pravidel nebo připojení k databázi. Výslednicí tedy je, že klient je mnohem „hubenější“. A to je velká a důležitá výhoda pro klienty, kteří běží na malých zařízeních.

Protože Enterprise Beans jsou přenositelné komponenty, je možné z nich seskládat novou aplikaci. Za předpokladu, že používají standardní rozhraní API, mohou tyto aplikace být spuštěny na jakémkoliv kompatibilním Java EE serveru. (19)

3.11.3 Využití Enterprise Beans

- Aplikace musí být škálovatelné. Je-li předpokládán rostoucí počet uživatelů, možná bude nezbytné distribuovat komponenty aplikace na více počítačích. Nejen, že mohou Enterprise Bean aplikace běžet na různých počítačích, ale také jejich umístění zůstane transparentní vůči klientům.
- Při transakcích musí být zajištěna integrita dat. Enterprise Bean transakce podporují. Mechanismy, které spravují souběžný přístup ke sdíleným objektům.
- Aplikace bude mít celou řadu různých klientů, kteří budou obsahovat jen pár řádků kódu. Vzdálení klienti mohou snadno najít Enterprise Bean, bez ohledu na jejich počet nebo druh.

Enterprise Beans se rozdělují na Session beans a Message-Driven beans. (19)

3.11.4 Session Beans

Zapouzdřuje business logiku tak, že může být vyvolána klientem vzdáleně nebo webovou službou připravující view pro klienta. Přístup k aplikaci nacházející se na serveru klient navazuje zavoláním metod obsažených v Session Beans. Session Bean vykonává práci pro svého klienta a chrání jej před složitostí provedením pracovních úkolů uvnitř

serveru. Session Bean není persistentní, to znamená, že jeho data nejsou uložena do databáze. (19)

Typy Session Beans

Rozlišujeme tři typy Session Beans:

- Stateless Session Beans (Bezstavové)
- Stateful Session Beans (Stavové)
- Singleton Session Beans (jedináček)

Stateless Session Beans (Bezstavové)

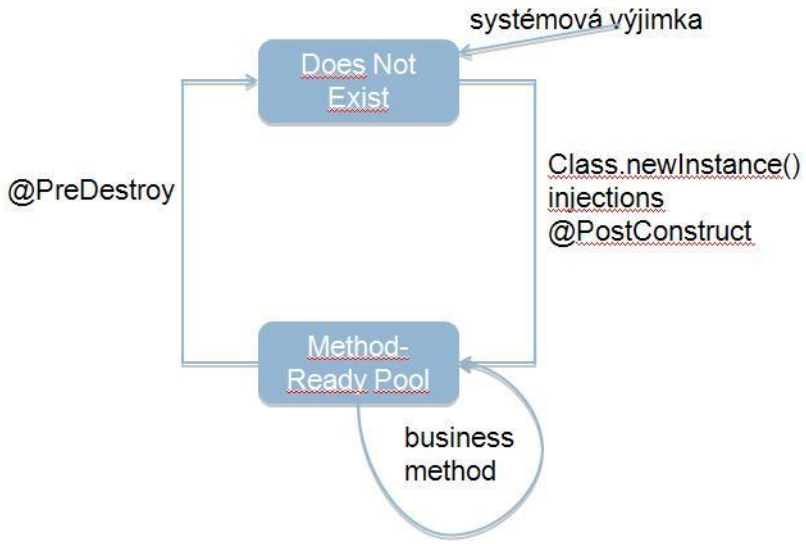
Ozanačují se anotací `@Stateless`.

Bezstavový bean neudržuje s klientem conversational state (konverzační stav) a většinou provádí jednorázové operace.

Když klient zavolá metodu bezstavového beanu, proměnné v beanu budou obsahovat stav, specifický ke klientovi v momentě, kdy byla metoda zavolána. Až metoda dokončí svou činnost, specifický stav klienta nebude zachován.

Jedna instance beanu zvládne obsluhovat více klientů. Obvykle ale existuje více instancí, které jsou uloženy v poolu (skupina instancí připravena provádět velké množství malých operací). Proto dovolují větší škálovatelnost aplikacím, které budou obsluhovat velké množství klientů. Typicky aplikace potřebuje menší množství bezstavových beanů než stavových pro obsloužení stejného množství klientů.

Bezstavový bean také dokáže implementovat web service, což stavový neokáže. (19)



Obrázek 13 - Stateless Session Beans

[Zdroj: http://docs.oracle.com/]

`@PostConstruct` Takto anotovaná metoda je zavolaná po vytvoření instance beanu a zařazení do poolu na serveru. Může zde být realizováno připojení k databázi.

Ve stavu Method-Ready Pool může provádět business logiku pro klienta. Je-li intance beanu ukončována běžným způsobem, zavolá se metoda anotovaná `@PreDestroy`. Zde může dojít k odpojení z databáze. Dojde-li k systémové výjimce tato metoda je přeskočena a instance beanu je odstraněna. (19)

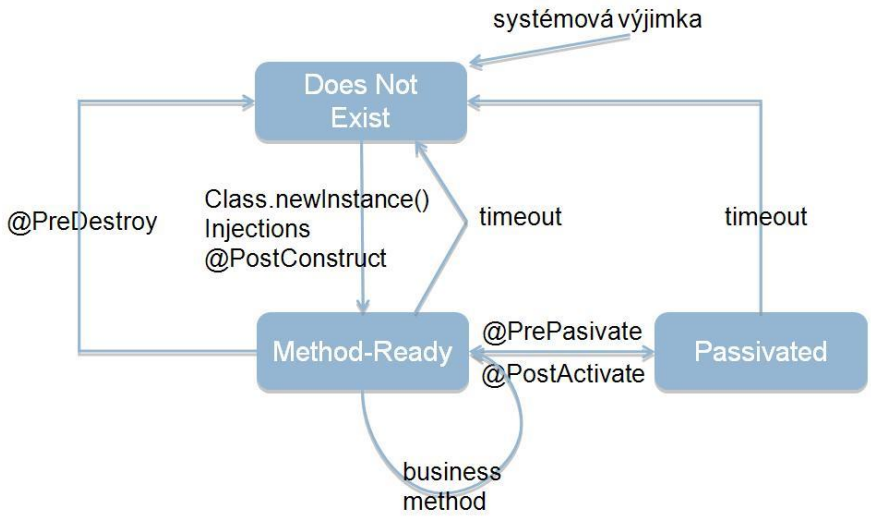
Stateful Session Beans (Stavové)

Ozanačují se anotací `@Stateful`.

Proměnné instance Stavového Session Beanu reprezentují unikátní stav clent/bean session a uchovávají si svůj stav mezi voláním metod v rámci jedné session. Stav komunikace Beanu s klientem je označován jako conversational state (konverzační stav).

Stavová Bean není sdílená, pro každého klienta je vytvořená unikátní bean instance. Při volání dalších metod se použijí data z předchozí interakce s klientem obsažené v instanci.

Když je klient přerušen, Session Bean se jeví jako ukončený a už se nijak nespojuje s klientem. V průběhu životního cyklu může dojít k pasivaci beanu a uložení jeho stavu pro uvolnění paměti na serveru. Stav je udržován po celou dobu spojení client/bean session, ale když klient spojení ukončí, stav zanikne.



Obrázek 14 - Stateful Session Beans

[Zdroj: http://docs.oracle.com/]

Cyklus probíhá téměř stejně jako u bezstavového beanu s tím rozdílem, že server může z nějakého důvodu bean uvést do pasivovaného stavu (uložení v serializované podobě). Pasivace probíhá v metodě anotované `@PrePasivated`. Bean může být ukončen běžným způsobem nebo expirací jeho životnosti na serveru (Timeout) nebo systémovou výjimkou. (19)

Singleton Session Beans

Tento bean je instancován jednou za aplikaci a je svázán s jejím životním cyklem. Jedináček je designován pro podmínky, kde jediná instance beanu je dostupná současně všem klientům. Nabízí podobné funkce jako bezstavový bean s tím rozdílem, že existuje pouze jedna instance beanu jedináček za aplikaci. Na rozdíl od poolu bezstavových beanů, kde na odpověď klientů může odpovědět kterýkoliv z nich, je jedináček jediný kdo odpovídá. Stejně jako bezstavový bean může implementovat web service endpointy.

Bean jedináček si udržuje stav mezi voláním klienta, ale nedokáže stav udržet po vypnutí nebo kolapsu serveru.

Aplikace vyžívající jedináčka by měly zajistit, aby byla vytvořena jeho instance hned po startu aplikace. Což dovolí jedináčkovi provést inicializační operace pro aplikaci. Stejně tak dokáže provést „uklízecí“ operace při vypínání aplikace, jelikož je vázán na celý její životní cyklus. Životní cyklus je stejný jako u bezstavového beanu, s tím, že `@PostConstruct` a `@PreDestroy` proběhnou při správném běhu beanu pouze jednou. (19)

Kdy použít který Session Bean

Stavový bean:

- Stav beanu reprezentuje interakci mezi beanem a specifickým klientem.
- Bean potřebuje udržet informace o klientovi v průběhu volání různých business metod.
- Bean je prostředníkem mezi klientem a další komponentou aplikace. Prezentuje zjednodušenou view klientovi.
- Bean v pozadí ovládá workflow několika Entreprise beanů.

Bezstavový bean:

- Bean neobsahuje žádná data pro specifického klienta.
- Po zavolání jedné metody, bean provede obecný úkol pro všechny klienty (poslat e-mail, který potvrzuje online objednávku).
- Bean implementuje web service.

Bean jedináček:

- Je-li potřeba sdílet stav napříč aplikací.
- Je-li potřeba přistupovat k jednomu Entrprise beanu více vlákny současně.
- Pokud aplikace potřebuje provést určité operace při startu nebo ukončení.

- Bean implementuje web service. (19)

3.11.5 Message-Driven Bean (Zprávami-řízený)

Jedná se o Enterprise bean, který dovoluje Java EE aplikacím zpracovávat zprávy asynchronně. Tento druh beanu se chová jako JMS[6] message listener[7], který je podobný jako event listener, ale na místo událostí dostává JMS zprávy.

Zpráva může být zaslána jakoukoliv Java EE komponentou (aplikačním klientem, jinou Enterprise beanou nebo webovou komponentou) nebo JMS aplikací nebo jinou aplikací, která nevyužívá Java EE technologie. (19)

Rozdíl Message-Driven Beans a Session Beans

Největším rozdílem od Session beans je, že klient nepřistupuje k zprávami řízeným beanům přes interface (rozhraní) nebo no-interface view, které definuje přístup klienta, ale obsahují pouze bean třídu.

Podobnosti s bezstavovým beanem:

- Instance zprávami řízeného beanu neobsahuje žádná data pro specifického klienta nebo neudržuje konverzační stav s klientem.
- Všechny instance jsou si rovny a mohou být řazeny do poolu. Zaslanou zprávu dokáže řešit kterákoliv instance zprávami řízeného beanu.
- Jeden zprávami řízený bean dokáže řešit zprávy od více klientů zároveň.

Komponenty klienta nekomunikují přímo s beanem. Na místo toho JMS posílá zprávy na místo, které je nadefinované při nasazení, a to použitím zdrojů GlassFish serveru. Toto místo bean odposlouchává nebo čeká na zprávu prostřednictvím `MessageListener`.

Charakteristiky zprávami řízených beanů:

- Vykonají se na základě jedné zprávy od klienta.

[6] Java Messaging Services
[7] Message listener – volně přeloženo jako posluchač zpráv. Nebo čekající na zprávu. Přesto se používá termín listener.

- Jsou volány asynchronně.
- Mají krátkou životnost.
- Nereprezentují přímo sdílená data v databázi, ale dokáží k nim přistupovat a měnit je.
- Nemají stavy.

Po příchodu zprávy je zavolána metoda `onMessage`, která zprávu zpracuje. Metoda `onMessage` zprávu přesměruje (cast) na jednu z pěti JMS message typů a zpracuje ji na základě business logiky aplikace. Metoda `onMessage` může zavolat pomocné metody nebo jiný session bean pro zpracování informací ve zprávě nebo k uložení informací do databáze. (19)

Použití Message-Driven Beans

Session beans dovolují posílat a přijímat JMS zprávy synchronně ne však asynchronně. Příjmem synchronních zpráv komponentou na straně serveru dochází k blokování prostředků serveru. JMS by všeobecně neměl přijímat ani posílat zprávy synchronně. Pro přijímání a posílání asynchronních zpráv se používají message-driven beans. (19)

3.11.6 EJB kontejner

Představuje dedikovaný virtuální prostor v aplikačním serveru, kam se nasazují EJB komponenty a řeší následující problematiku:

- Komunikace se vzdáleným klientem – zjednodušuje komunikaci klienta s aplikací.
- Dependency injection – plní proměnné například: JMS zprávami, datovými zdroji (SQL připojením) nebo jinými EJBeany.
- Řízení stavu – kontejner udržuje v paměti stavy jednotlivých stavových (stateful) beanů a tím i vzdálený stav u klienta, kterému se jeví stav jakoby uložen lokálně.
- Pooling – vytváření poolu instancí pro bezstavové beany a message-driven beany.

- Řízení životního cyklu – stará se o vytváření, inicializaci a destrukci instancí beanů a další události.
- Messaging – umožňuje MDB poslouchat na JMS destinacích a přijímat zprávy.
- Management transakcí – beany deklarují transakční vlastnosti metod, kontejner řeší commit a rollback.
- Bezpečnost – deklarace přístupů na úrovni tříd a metod
- Podpora souběžného zpracování – řeší problémy synchronizace souběžných přístupů ke sdíleným datům
- Správa interceptorů – komponenty umožňující odchytávat okamžik před a po volání metody
- Asynchronní volání metod

Embedded kontejner

Nabízí možnost spouštět EJB aplikace v prostředí Java SE. Usnadňuje testování a ladění, ale podporuje jen podmnožinu EJB Lite (bez MDB, vzdálených rozhraní, webových služeb, …). (13) (22)

3.12 Teoretická východiska

Při návrhu aplikace je potřeba určit a zvážit základní požadavky (pravděpodobný počet klientů, způsob jakým bude aplikace zpracovávat informace a požadavky na budoucí rozšíření aplikace atd.). Na základě analýzy počátečních požadavků se volí postup řešení. Od počátku informačních systémů vzniklo mnoho vývojových postupů s různou složitostí provedení (vodopádový, prototypový, inkrementální atd., a jejich různé kombinace).

Vrstvené řešení návrhu aplikací se jeví vhodné pro návrh aplikací využívajících síť. Kdy se funkčnost aplikace rozdělí do vrstev obstarávající jednotlivé úkoly. Není-li k chodu aplikace vyžadován přístup na síť, zůstává návrh v jedné vrstvě, to ale neznamená, že by se funkčnost nedala rozdělit do bloků (boxů).

Při konstrukci aplikace je vhodné následovat návrhové vzory. Mnoho již čelilo stejnému nebo podobnému problému při konstrukci aplikace. Ti, kterým se podařilo vyřešit jejich problém elegantním a úsporným způsobem, zaznamenali svůj úspěch v podobě návrhového vzoru, který může usnadnit spoustu zbytečných chyb při konstrukci dalších aplikací.

MVC architektura je vhodná pro konstrukci webových (vícevrstvých) aplikací, její logika se však dá také uplatnit jako návrhový vzor v jednovrstvé architektuře pro zpřehlednění celého návrhu.

Jednou z moderních technologií pro vývoj webových/podnikových (vícevrstvých) aplikací je Java Enterprise Edition, která je v praxi velice rozšířená. Nejedná se o jinou verzi Javy, je to pouze nadstavba Javy Standart Edition.

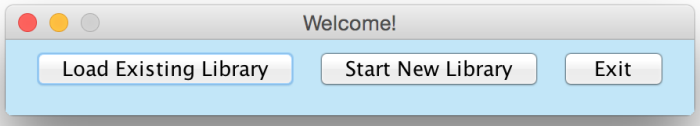
Java EE využívá k vývoji aplikací dalších nadstavbových komponent, které mohou pocházet od třetích stran a mohou být komerční. Databázové přístupy řešené pomocí JDBC nebo ORM. Frameworky usnadňující vývoj aplikací jako Spring nebo EJB, kdy každý poskytuje specifické funkce. Dají se využít k různým cílům, ale dají se i různě kombinovat.

4 Vlastní práce

Jako vlastní práci autor navrhnul a zkonstruoval jednovrstvou aplikaci s názvem Borrower pro jednoho uživatele, do které si uživatel může zaevidovat své předměty (převážně knihy), které by mohl někomu půjčit. Dále je uživatel schopen si zaevidovat osoby, kterým by tyto předměty mohl půjčit. Ve chvíli kdy má uživatel alespoň jeden předmět a jednu osobu, může provést proces vypůjčení. Vypůjčení obsahuje datum půjčení a přepokládaný datum návratu předmětu. Neuskuteční-li se tak, může uživatel osobu informovat prostřednictvím zaslání emailové zprávy.

4.1 Popis aplikace

Po spuštění aplikace je uživatel přivítán a může si vybrat, jestli chce začít novou knihovnu nebo načíst existující.



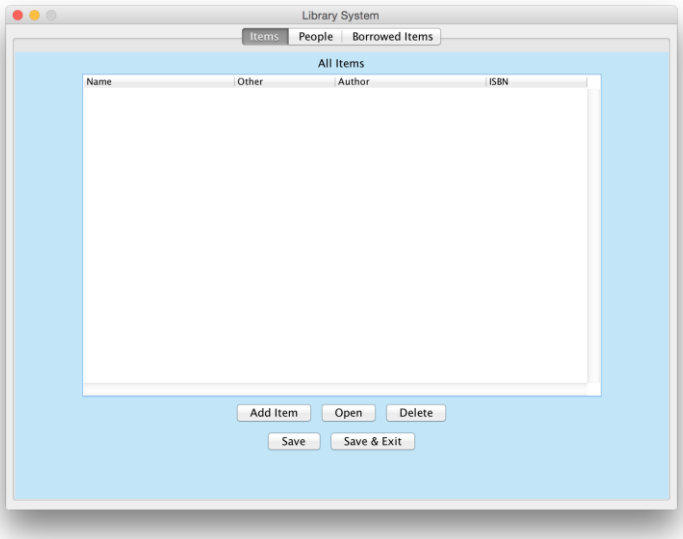
Obrázek 15 - Load Screen

[Zdroj: Autor]

Pro představení aplikace bude vybrána možnost založení nové knihovny. Po stisknutí tlačítka se zobrazí nové okno aplikace, přepnuté na záložku Items (předměty). Aplikace dále nabízí záložky People (lidé) a Borrowed Items (zapůjčené přeměty). Přepínání mezi záložkami je v horní části okna. 

4.1.1 Items (Předměty)

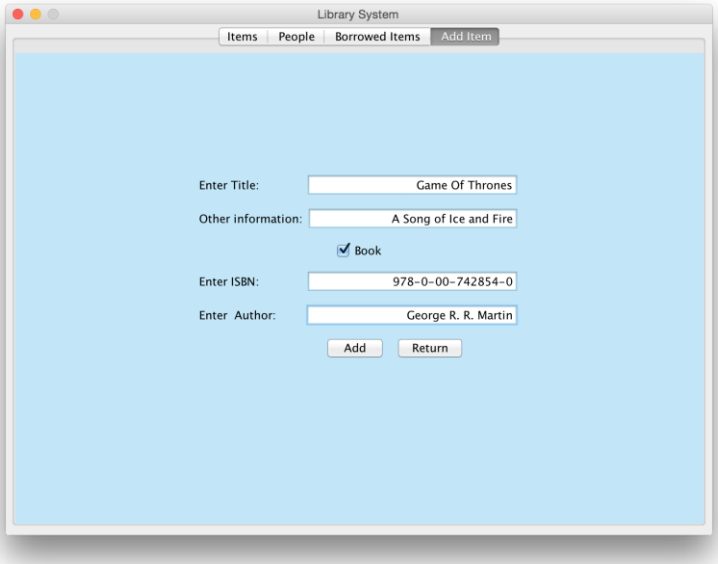
Aplikace přepnutá na záložku předmětů obsahuje v hlavní části tabulku uživatelových existujících předmětů. Pod tabulkou jsou tlačítka pro manipulaci s předměty (přidat, upravit, odstranit) a tlačítka pro uložení knihovny a ukončení aplikace.



Obrázek 16 - Okno aplikace - předměty

[Zdroj: Autor]

Na obrázku je zřejmé, že uživatel nemá vytvořené žádné předměty. Vytvoření nového předmětu je zobrazeno na následujícím obrázku.



Obrázek 17 - Okno aplikace - nový přemět

[Zdroj: Autor]

Po kliknutí na tlačítko Add Item (přidat předmět) se zobrazí nová záložka, kde uživatel zadá následující informace:

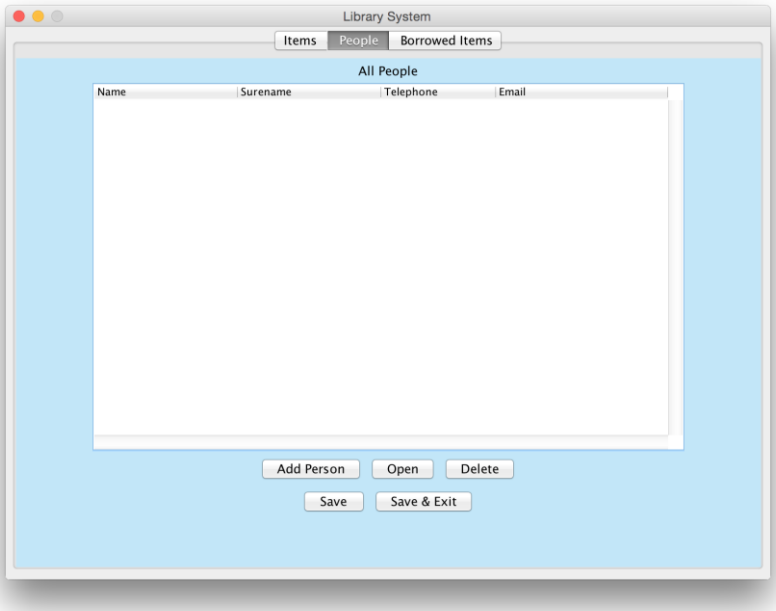
- Enter Title (zadej název) – je povinné pole a bez jeho vyplnění nebude uživatel moci pokračovat. Název je typu String.

- Other Information (ostatní informace) – je pole nepovinné, tudíž může zůstat nevyplněno. Je typu String.
- Book (kniha) – je zaškrtávací box. Je-li aktivní, zpřístupní se další dvě doplňková pole týkající se knihy.
- Enter ISBN (zadej ISBN) – pole je zpřístupněné pouze pokud je aktivní zaškrtávací box Book. Pole je povinné a lze do něj zadat pouze číslice. Pole validuje formát ISBN-10 a ISBN-13 a je typu Long.
- Enter Author (zadej autora) – pole je zpřístupněné pouze pokud je aktivní zaškrtávací box Book. Pole je povinné a typu String.

Jsou-li všechna pole správně vyplněna, tak po stisknutí tlačítka Add se pohled přepne na záložku předmětů a nový předmět bude zobrazen v tabulce. Pokud některá pole nebudou validní, musí je uživatel opravit, dokud tak neučiní, nebude moci dále pokračovat v aplikaci.

4.1.2 People (Lidé)

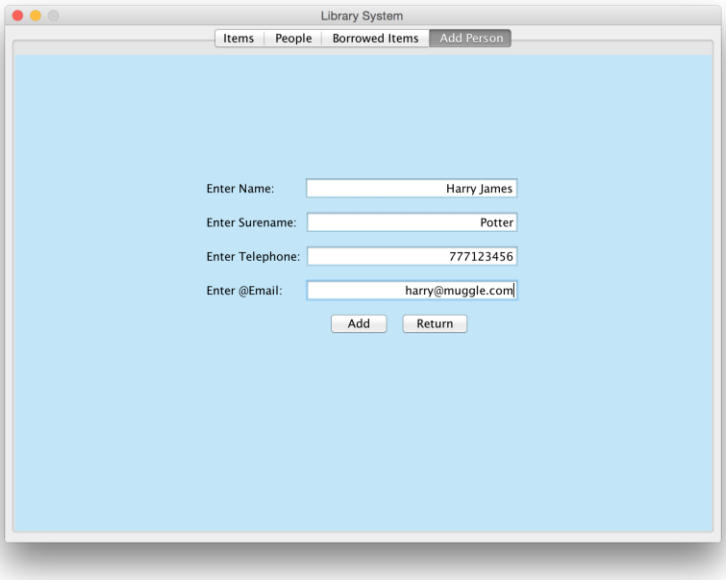
Záložka People (lidé) má velice podobné zobrazení a ovládací prvky jako záložka přemetů.



Obrázek 18 - Okno aplikace - lidé

[Zdroj: Autor]

Pod tabulkou obsahující přidané osoby jsou ovládací tlačítka (přidat, upravit, odstranit) a tlačítka pro uložení knihovny a ukončení aplikace. Přidávání lidí je popsáno na následujícím obrázku.



Obrázek 19 - Okno aplikace - přidat osobu

[Zdroj: Autor]

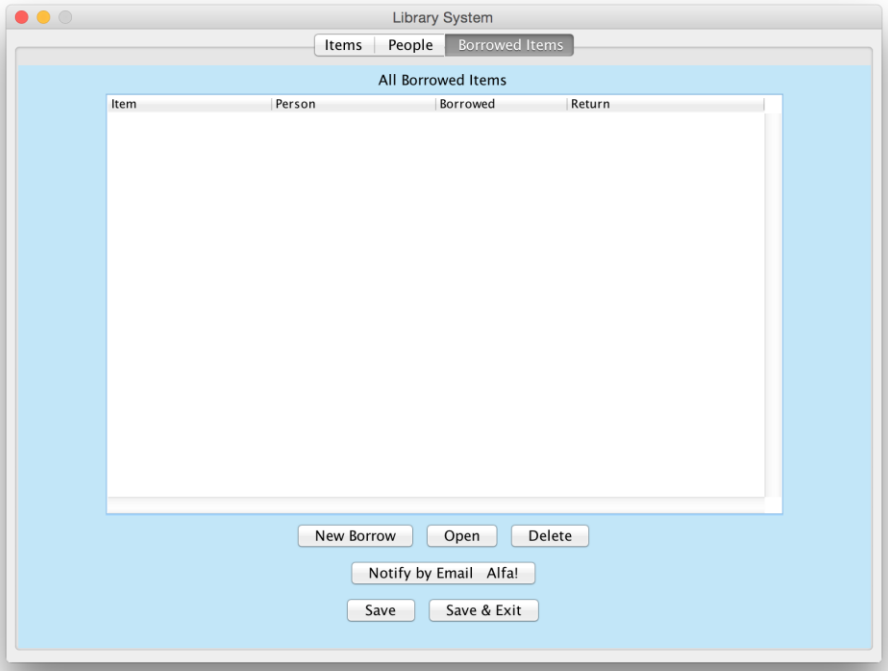
Po kliknutí na tlačítko Add Person (přidat osobu) se zobrazení přepne na nový panel, kde uživatel vyplní příslušná pole. Všechna pole jsou povinná, aby uživatel získal dostatek informací o osobě, které hodlá předměty půjčovat.

- Enter Name (zadej jméno) – je pole typu String.
- Enter Surename (zadej přijmění) – je pole typu String.
- Enter Telephone (zadej telefon) – do pole lze zadat pouze číslo a pole je typu Integer.
- Enter @Email (zadej email) – do pole lze zadat pouze validní emailovou adresu. Pole je typu String.

Jsou-li všechna pole správně vyplněna, tak po stisknutí tlačítka Add se pohled přepne na záložku lidí a nová osoba bude zobrazena v tabulce. Pokud některá pole nebudou validní, musí je uživatel opravit, dokud tak neučiní, nebude moci dále pokračovat v aplikaci.

4.1.3 Borrowed Items (Zapůjčené přeměty)

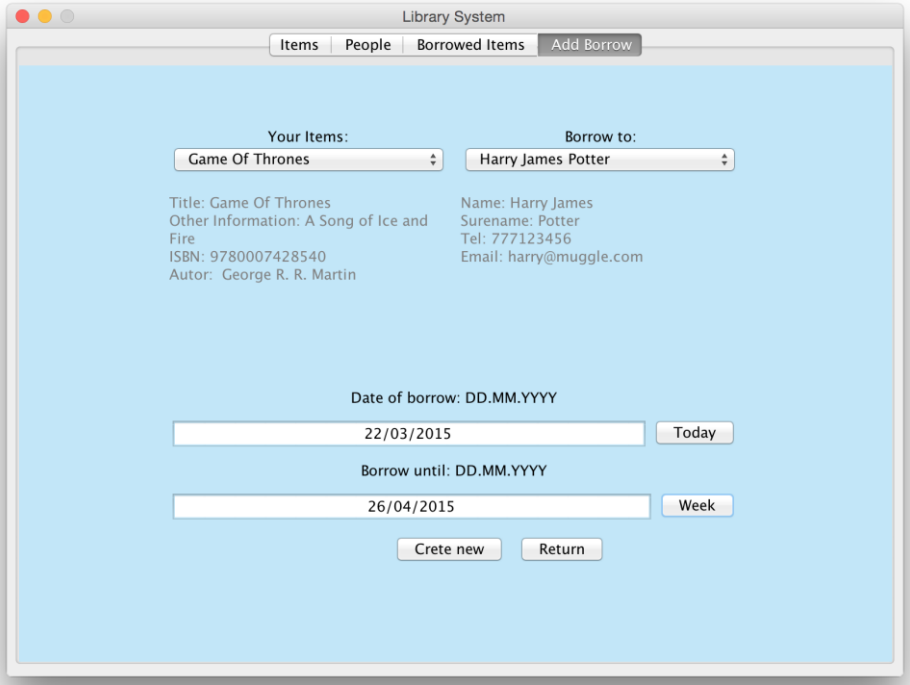
Záložka aplikace s názvem Borrowed Items (zapůjčené předměty) také obsahuje tabulku, ve které se zobrazují informace o zapůjčených předmětech. Dále také ovládací tlačítka (přidat, upravit, odstranit) a tlačítka pro uložení knihovny a ukončení aplikace. Oproti ostatním záložkám také obsahuje tlačítko Notify by Email (upozornit emailem). Funkce bude popsána v další kapitole. Záložka Borrowed Items je zobrazena na následujícím obrázku.



Obrázek 20 - Okno aplikace - zapůjčené přeměty

[Zdroj: Autor]

Je-li v aplikaci vytvořen alespoň jeden předmět a alespoň jedna osoba, je možné provést vypůjčení předmětu vybrané osobě. Proces je popsán na následujícím obrázku.



Obrázek 21 - Okno aplikace - vypůjčení
[Zdroj: Autor]

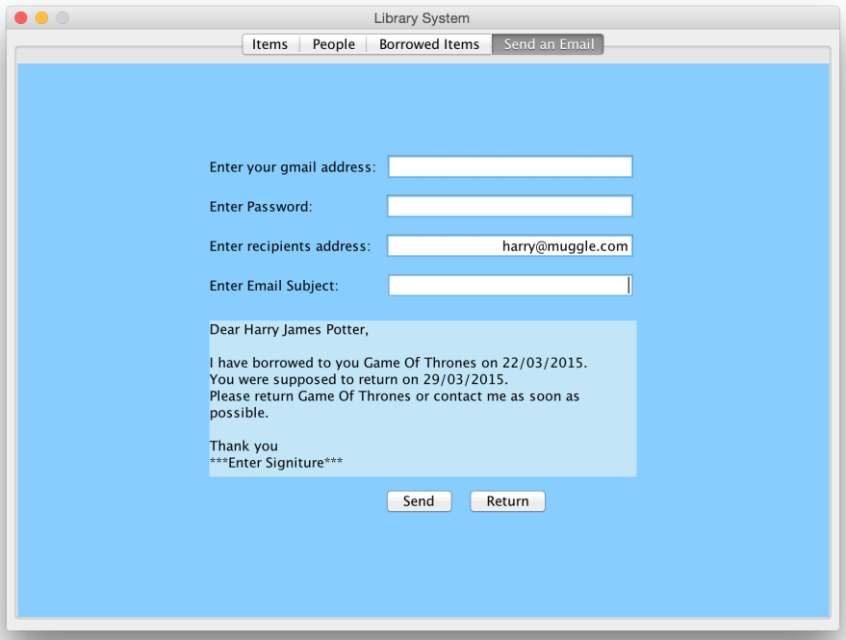
Po kliknutí na tlačítko New Borrow (nové vypůjčení) se aplikace přepne na nový panel. V levé části je vysouvací nabídka nadepsaná Your Items (vaše předměty). Tato nabídka zobrazuje pouze filtrované předměty. Pokud žádné předměty v aplikaci nejsou, nabídka uživatele upozorní. V nabídce se zobrazí pouze předměty, které nejsou půjčeny. V pravé části je vysouvací nabídka nadepsaná Borrow to (půjčit komu). V nabídce se zobrazí seznam osob, kterým lze předmět půjčit. Pod každou z nabídek je needitovatelná oblast, která poskytuje náhled odpovídající položce vybrané v nabídce výše.

Ve spodní části jsou pole prezentující data. Date of borrow (datum vypůjčení) a Borrow until (vypůjčit do). Do obou polí lze zadávat pouze číslice a jsou ve formátu DD.MM.YYYY. Napravo se nacházejí tlačítka pro usnadnění zadávání data. Today (dnes) vyplní pole datem, které je nastavené na zařízení, na kterém je aplikace spuštěna. Week (týden) vyplní pole datem, které je o sedm dní vyšší, než je datum na lokálním stroji. Při každém dalším stisknutí se datum zvýší o týden. Do obou polí lze zadávat data až do roku 2099.

Jsou-li všechna pole správně vyplněna, tak po stisknutí tlačítka Create new (vytvořit nový) se pohled přepne na záložku vypůjčených předmětů a nové vypůjčení bude zobrazena v tabulce. Pokud některé z polí nebude validní, musí je uživatel opravit, dokud tak neučiní, nebude moci dále pokračovat v aplikaci.

Funkce Notify by Email (Upozornit emailem)

Tato funkce je omezena na poštovní server Google. A k jejímu provozování je potřeba mít vytvořený Gmail účet. Po stisknutí tlačítka se zobrazí následující panel.



Obrázek 22 - Okno aplikace - upozornit emailem

[Zdroj: Autor]

Pro správnou funkci musí uživatel zadat do polí *Enter your gmail address* a *Enter Pasword* validní Gmailovou adresu a heslo.

Pole *Enter recipients address* je vyplněno automaticky na základě informace z tabulky osob.

Uživatel si může zvolit pole předmětu (*Enter Email Subject*) sám nebo je může ponechat nevyplněno.

Vlastní obsah zprávy se generuje automaticky na základě informací z tabulek svázaných s výpůjčkou. Překlad zprávy zní:

„Drahý jméno a příjmení osoby,

Zapůjčil jsem Vám název předmětu dne datum zapůjčení.

Předmět jste měl navrátit dne datum navrácení.

Prosím vraťte mi název předmětu nebo mne kontaktujte jakmile to bude možné.

Děkuji,

Podpis uživatele.“

Obsah zprávy je editovatelný a uživatel může celou zprávu smazat a napsat jakýkoliv jiný email na kteroukoliv emailovou adresu.

Po stisknutí tlačítka Send (odeslat) se funkce pokusí navázat spojení se serverem. Pokud uživatel zadal všech informace správně, zobrazí se hláška, označující úspěšné odeslání zprávy. Pokud ne bude uživatel vyzván ke kontrole zadaných údajů.

4.2 Vlastní implementace návrhového vzoru Singleton (Jedináček)

Autor použil lehce odlišnou implementaci návrhové vzhoru Singleton než je v příloze, ale výsledek je stejný. Vzoru bylo využito při tvorbě nového okna panelu (vytvoření nového předmětu). Není žádoucí, aby byl uživatel schopen otevřít více oken najednou, použitím vzoru byl tento problém vyřešen.

Po stisknutí tlačítka na vytvoření nového předmětu se provede následující kód.

```
screen.jtp.setEnabledAt(0, false);
new AddItemPanel(this, screen, null,constructISBNlist());
```

První řádek říká, že přístup do panelu s indexem 0, bude znemožněn.

Druhý řádek je přetížený konstruktor nového panelu pro přidání předmětu. Po vytvoření jeho instance se pohled přepne na pozici, kde je nově vytvořený panel. Přepnutí provede následující kód:

```
screen.getTabbedPane().setSelectedComponent(this);
```

A jelikož panel, odkud byl konstruktor zavolán, je deaktivovaný, není možné zkonstruovat další instanci panelu. Tedy instance třídy `AddItemPanel` je pouze jedna. Po kliknutí na tlačítko přidat (Add), se zkontroluje obsah hodnot, zadaných v panelu, pokud je vše v pořádku, tak se vytvoří nový objekt (předmět) a panel se smaže. Vytvořený objekt je odeslán zpět do systémové třídy, která s ním dále pracuje. Dále přepne zpět na panel s předměty, odkud je možné vytvářet novou instanci třídy `AddItemPanel`.

4.3 MVC architektura v projektu

Projekt je sice jednovrstvá aplikace, ale její třídy jsou rozděleny do package (balíčků) podle architektury MVC. Rozdělení mnohem zjednodušuje orientaci ve zdrojovém kódu aplikace.

Obrázek Package Explorer v příloze zobrazuje rozložení tříd projektu do balíčků.

- library.controller – obsahuje systémovou třídu aplikace a třídy poskytující další funkce (ověřování vstupů, posílání emailů).
- library.model – obsahuje logické třídy aplikace
- library.view.add – obsahuje třídy, které vytváří instance panelu pouze s dočasným zobrazením. V aplikaci se vyskytují pouze omezenou dobu, po dokončení svého úkolu zaniknou. Instance vytvořených panelů jsou jedináčci.
- library.view.mainPanels – obsahuje třídy, které vytváří instance panelu se stálým zobrazením. Instance panelu jsou zobrazeny od spuštění a jsou pouze dočasně deaktivovány.
- library.view – obsahuje třídy pro vytvoření prvního zobrazení po startu aplikace a třídu vytvářející zobrazení pro odesílání emailů.

4.4 Využití Interface v projektu

Projekt obsahuje rozhraní ICallBackEvent, který reprezentuje služebníka pro předávání vytvořených nebo upravených objektů zpět do systémové třídy nebo vrací zprávu o ukončení akce (Return).

ICallBackEvent rozhraní obsahuje následující metody:

- návrat nového objektu

```
public void addCallBack(Object o);
```

- návrat upraveného objektu

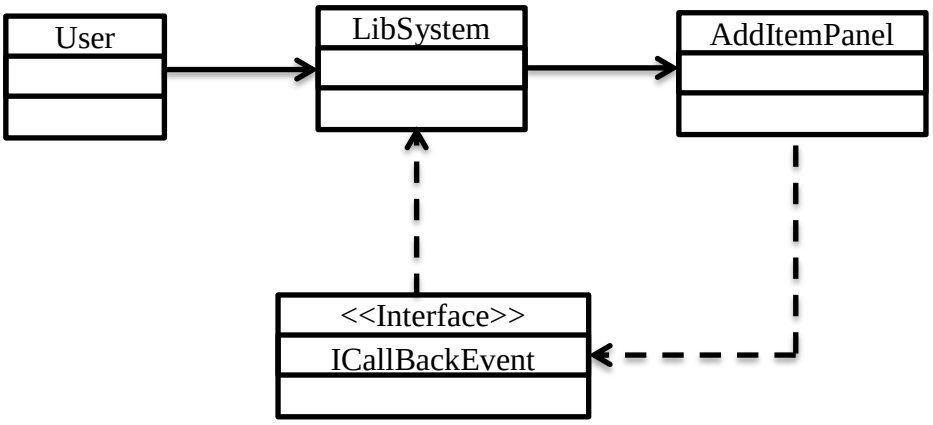
```
public void editCallBack(Object o);
```

- návrat načtené knihovny

```
public void libraryCallBack(Object o);
```

- ohlášení ukončení operace

```
public void cancelCallBack(int index);
```



Obrázek 23 - Interface v projektu

[Zdroj: Autor]

Vlastní implementaci metod obsahuje systémová třída LibSystem, která rozhraní implementuje.

Příklad implementace metody rozhraní `addCallBack` s návratovým objektem typu Item.

```
public void addCallBack(Object o) {

        if (o instanceof ItemClass){
              ItemClass item = (o instanceof ItemClass) ?
              (ItemClass)o : null;
              if(item == null)
                    return;
```

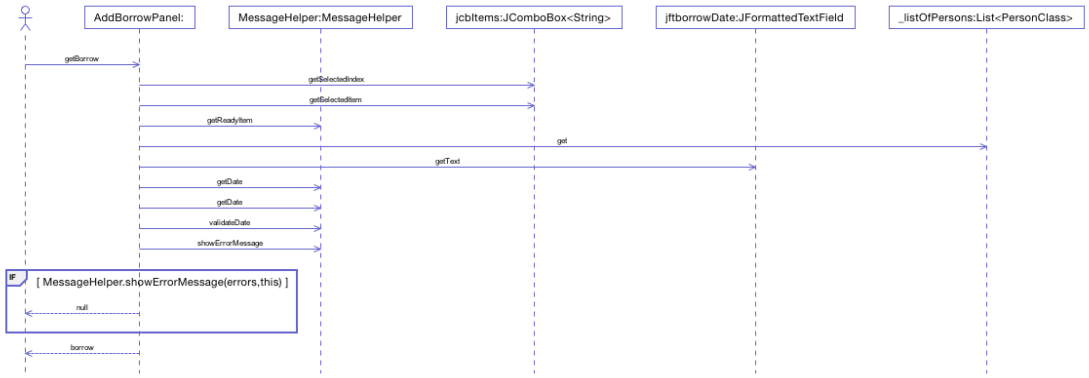
```
                lib.addItemtoItemsCollection(item);
                reloadTables();
                afterItemAction();
            }
} [Zdroj: Autor]
```

Metoda nejprve ověří, zda se jedná o objekt typu Item a pokud ano, tak ho přiřadí do kolekce v knihovně. Dále aktualizuje zobrazení tabulek a provede další operace po návratu (přepne zpět na panel s předměty v aplikaci). Pokud ne, vrátí se z metody a pokračuje v programu.

4.5 Sequence diagram

Autor k vygenerování diagramu použil plug-in *ModelGoon UML4Java*.

Na následujícím obrázku je zobrazen sekvenční diagram metody getBorrow(). Diagram popisuje sekvenci metod pro získání a validaci informací, ze kterých je poté vytvořena instance BorrowClass. Stejná sekvence se opakuje při editování instance BorrowClass.



Obrázek 24 - Diagram - getBorrow

[Zdroj: Autor]

4.6 Persistence dat v projektu

Persistenci dat v projektu zajišťuje třída FileHelper, které je posílána celá knihovna aplikace. Třída FileHelper poté uloží knihovnu do souboru na lokálním stroji. Při dalším spuštění může uživatel načíst existující knihovnu. Knihovna pak existuje v aplikaci a změny se v souboru neprojeví, dokud není znovu uložena.

Knihovna je ukládána do souboru s příponou „ *.SER “. Tato přípona je nejčastěji spojována s Javou.

4.6.1 Soubor .SER

Soubor .SER obsahuje serializovaná data objektu. Serializace je vestavěný mechanismus pro manipulaci s objektem jako s proudem bitů. Serializace objektu poskytuje základy pro vzdálené volání metod Javy (RMI – remote method invocation) a umožňuje Java objektům rozmístěným na síti, aby dokázali volat své vzdálené metody. RMI je často používán v distribuovaných podnikových aplikací, které jsou postaveny na Java EE. (23)

Volba souboru .SER se tedy jeví jako výhodná v případě budoucího rozšíření aplikace s využitím síťového připojení.

4.7 Black Box a White Box testování

Autor v projektu nesčetněkrát použil obou testovacích metod.

Metoda Black Box byla použita pro testování Grafického rozhraní GUI. Reakce na tlačítka, vyplňování polí a rozevíracích nabídek atd.

White Box metoda byla použita při testování funkčnosti jednotlivých bloků a odstraňování poruch v programu. White Box testování proběhlo prostřednictvím debuggeru ve vývojovém prostředí Eclipse. Konkrétně při plnění rozevírací nabídky v panelu New Borrow, kdy metoda pro plnění nabídky byla jednou z logicky nejobtížnějších v projektu.

4.8 Podobnost Java Beans a Black boxů

Black Box pojem byl definován jako třída, která na venek zobrazuje jenom metody, které programátor nadefinuje jako veřejné. Vnitřní logika třídy zůstává pro ostatní části programu skryta.

Java Bean byl definovaný jako třída, jejíž proměnné jsou privátní a přístup k nim je zajištěn pomocí přístupových metod. Dále má bezparametrický konstruktor a implementuje knihovnu Serializable.

Zatím co Black Box je spíše obecná technika, se širokou škálou použitelnosti v programovacím prostředí.

Java Bean se používá v oblasti konstrukce vícevrstvých enterprise aplikací. Konkrétně v aplikační vrstvě na straně servu. A stanovuje konvence dodržované při sestavování tříd v aplikaci.

Dalo by se tedy vyvodit, že Java Bean je speciálním případem Black Boxu.

4.9 Závěrečná diskuse

Autor by rád poukázal na nedostatek nástrojů při práci se SWING komponentou JTable. Bylo by vhodné rozšířit tuto komponentu o další funkčnost, která by práci s ní značně usnadnila. Příkladem může být vymazání obsahu tabulky, které muselo být prováděno cyklem.

Další rozšíření funkčnosti by autor navrhl u kontejnerů (konkrétně List), kde by bylo vhodné přidat metodu, která by přijímala na vstupu proměnné dvou listů. Položky listů by poté vzájemně porovnala a výstupem by byl třetí list, který by obsahoval buď shodné, nebo rozdílné položky. Autorova implementace metody je vnoření dvou cyklů a použití iterátoru při porovnávání položek. S potřebou této funkce už se autor několikrát setkal.

Práce s komponentou JComboBox se také jeví jako méně efektivní, je to kvůli nepraktickému přístupu k zobrazeným položkám v rozbalovacím menu. Komponenta je naplněna objekty typu String a přistupuje se k nim metodou getSelectedIndex(), což vrátí index vybrané položky nikoliv odkaz na objekt. Tudíž naplní-li se JComboBox objekty se shodným atributem String, který je zobrazen. Tak i přesto, že ostatní atributy mohou být rozdílné JComboBox, si neví rady a při označení jedno z atributů, označí všechny se stejným Stringem, protože se mu jeví jako totožné. Řešením tohoto problému by bylo ošetřit předměty, aby neobsahovaly duplicitní názvy. Což se jeví jako nepraktické, protože uživatel může vlastnit více totožných knih. Dalším možným řešením je zabalit hodnoty do třídy a přiřadit k instancím klíče. Což se opět jeví jako nadbytečná práce pro programátora.

Nejoptimálnějším řešením by podle autora bylo, přidat do knihovny JComboBox metodu, která by dokázala naplnit JComboBox objekty a zobrazila vybraný String atribut. Případně komponentu JComboBox rozšířit tak, aby tuto funkčnost zvládla.

```
jComboBox.addNamedObject(Object,String); [Zdroj: Autor]
```

Object reprezentuje objekt schovaný pod indexy jednotlivých položek nabídky a String by byla textová hodnota, která se v nabídce zobrazí. Praktická implementace by mohla vypadat následovně.

```
jComboBox.addNamedObject(item,items.getTitle());[Zdroj: Autor]
```

Tudíž když se provede metoda getSelectedObject() vrátí se celý objekt a ne jenom String hodnota, které je zobrazena v nabídce. Nepraktické by to bylo v případě, že by objekty byli rozměrné a bylo jich mnoho. V tom případě by se jevila metoda zabalování do tříd efektivnější.

Při vývoji aplikace autor narazil na diskusní fórum *stackoverflow.com* zabývající se různými programovacími jazyky a technikami. Autor zde nalezl mnoho užitečných rad a odpovědi na své problémy při vývoji aplikace. Podání řešení bylo na profesionální úrovni a řešené problémy byly zpracované několika různými způsoby a metodami, aby bylo řešení jasné a optimální.

5 Závěr

Java sama o sobě se jeví jako složitý programovací nástroj, ale díky různým nadstavbám a opensource knihovnám se jeho využití značně zjednodušuje. Vytváření složitějších aplikací v Javě by bylo značně složitou a dlouhodobou záležitostí pro jednoho člověka, ale díky opensource nadstavbám (ODB, Spring Framework, Enterprise Java Beans) a zdrojového kódu lidí, kteří je veřejně vystavili pro širokou veřejnost, se celý proces tvorby aplikace značně zjednodušuje.

Nástroj Eclipse se pro programování v jazyce Java velmi osvědčil. Je vyvíjen pod licencí Common Public Licence (CPL) a umožňuje přidávat a distribuovat moduly v jazyce Java. Eclipse je příjemné vývojové prostředí a nabízí spoustu základních funkcí, které jsou k základnímu programování zapotřebí.

Cílem diplomové práce bylo představení architekturou řízeného vývoje aplikací v jazyce Java. Popsání základních architektur softwaru. Popsání pojmů jako Interface (rozhraní), návrhový vzor Model View Controller, architektura Black Box.

V praktické části měl autor vytvořit jednovrstvou aplikaci v programovacím prostředí Eclipse. Za použití MVC návrhového vzoru a architektury Black Box.

V práci bylo dosaženo všech stanovených cílů. V teoretické časti, byla popsána architektura rozdělující aplikaci na funkční vrstvy (jednovrstvá, dvouvrstvá a vícevrstvá). Dále byl vysvětlen pojem návrhových vzorů a jejich využití s příkladem konkrétních implementací. V další části práce byla popsána architektura MVC a její druhy. Definice a použití přístupových metod a rozhraní. Byl definován pojem Back Box programming. Velká část práce se věnuje programování v Java Enterprise Edition a nadstavbám ve třívrstvé architektuře, což nebylo konkrétně stanoveným cílem ale hluboce souvisí s tematikou architektonické tvorby aplikací v Javě.

V praktické části byla popsána studentem vytvořená jednovrstvá aplikace. A ukázky implementace architektur při konstrukci aplikace. Aplikace byla při vývoji testována metodami Black Box a White Box testing.

Rozložení aplikace podle architektury MVC se jeví jako efektivní pro funkčnost i zpřehlednění celé aplikace. Použití návrhových vzorů při konstrukci aplikace je také vysoce efektivním postupem.

Rozšíření funkcí aplikace by se dalo uskutečnit prostřednictvím customizace služby *upozornit emailem*, aby si klient mohl připojit svůj emailový účet a nebyl omezen pouze na Google klienta. Další funkcí, která by aplikaci prospěla, je možnost vyhledávání nebo řízení tabulek. Funkce třízení tabulek je deaktivována, protože by byla narušena funkcionalita programu.

Možné rozšíření aplikace na více vrstev za použití architektury MVC. Kdy by uživatel měl na svém zařízení klienta aplikace, který by si po přihlášení aktualizoval knihovnu ze serveru, a před ukončením by se změny přenesly na server. V případě komercializace aplikace by bylo třeba přidat doplňující pole s atributy k lidem a předmětům. Nebo vnořit další JTabbedPane do panelu s předměty.

Autor se domnívá, že vyřešení problematiky zmíněné v diskusi, by usnadnilo práci mnohým programátorům při vývoji všech aplikací.

6 Seznamy

6.1 Seznam použité literatury

1. PECINOVSKÝ, Rudolf. *Java 8: úvod do objektové architektury pro mírně pokročilé*. 1. vyd. Praha: Grada, 2014, 655 s. Knihovna programátora (Grada). ISBN 978-80-247-4638-8.

2. ČERMÁK, Miroslav. CLEVERANDSMART. *Vícevrstvá architektura* [online]. 2012 [cit. 2015-03-29]. Dostupné z: www.cleverandsmart.cz.

3. BRUCKNER, Tomáš. *Tvorba informačních systémů: principy, metodiky, architektury*. 1. vyd. Praha: Grada, 2012, 357 s. Management v informační společnosti. ISBN 978-80-247-4153-6.

4. GAMMA, Erich. *Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů*. 1. vyd. Praha: Grada, 2003, 386 s. Moderní programování. ISBN 80-247-0302-5.

5. PECINOVSKÝ, Rudolf. *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.

6. TECHTHERMS.COM. *TechTherms* [online]. 2015 [cit. 2015-03-29]. Dostupné z: http://techterms.com/.

7. ALEXANDER, Christopher, Sara ISHIKAWA a Murray SILVERSTEIN. *A pattern language: towns, buildings, construction*. New York: Oxford University Press, 1977, xliv, 1171 p. ISBN 0195019199.

8. ČÁPKA, David. ITNETWORK.CZ. *Objektově orientované programování v Javě* [online]. 2013. vyd. 2013 [cit. 2015-03-29]. Dostupné z: http://www.itnetwork.cz/java-objektove-orientovane-programovani-navody-a-tutorialy.

9. BOUDA, Radek. PROGRAMUJTE.COM. *Seriál návrhových vzorů* [online]. 2012. vyd. 2012 [cit. 2015-03-29]. Dostupné z: http://programujte.com/.

10. PECINOVSKÝ, Rudolf. *OOP: naučte se myslet a programovat objektově*. Vyd. 1. Brno: Computer Press, 2010, 576 s. ISBN 978-80-251-2126-9.

11. PECINOVSKÝ, Rudolf. *Myslíme objektově v jazyku Java: kompletní učebnice pro začátečníky*. 2., aktualiz. a rozš. vyd. Praha: Grada, 2009, 570 s. Myslíme v--. ISBN 978-80-247-2653-3.

12. PECINOVSKÝ, Rudolf, Jarmila PAVLÍČKOVÁ a Luboš PAVLÍČEK. UNIVERSITY OF BOLOGNA. *Let's Modify the Objects First Approach into Design Patterns First:*

Eleventh Annual Conference on Innovation and Technology in Computer Science Education. 2006. vyd. University of Bologna, 2006. Dostupné z: http://www.iticse06.cs.unibo.it/.

13. ORACLE. *Oracle* [online]. 2015 [cit. 2015-03-29]. Dostupné z: http://www.oracle.com/technetwork/java/index.html.

14. ČÁPKA, David. ITNETWORK.CZ. *MVC architektura* [online]. 2013. vyd. 2013 [cit. 2015-03-29]. Dostupné z: http://www.itnetwork.cz/.

15. FAQS.ORG. *Black boxand white box testing* [online]. 2014. vyd. 2014 [cit. 2015-03-29]. Dostupné z: http://www.faqs.org/faqs/software-eng/testing-faq/section-13.html.

16. ALLINTERVIEW.COM. *Differences Between Whitebox testing and Blackbox testing* [online]. 2015. vyd. 2015 [cit. 2015-03-29]. Dostupné z: http://www.allinterview.com/showanswers/33254/differences-between-whitebox-testing-and-blackbox-testing.html.

17. ZICHA, Vojtěch. PROGRAMUJTE.COM. *Java* [online]. 2007. vyd. 2007 [cit. 2015-03-29]. Dostupné z: programujte.com.

18. ORACLE. *The Java Tutorials: Defining an Interface* [online]. 2012 [cit. 2015-03-29]. Dostupné z: http://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html.

19. ČÁPKA, David. ITNETWORK.CZ. *Java Enterprise Edition (JEE)* [online]. 2014. vyd. 2014 [cit. 2015-03-29]. Dostupné z: http://www.itnetwork.cz/.

20. TUTORIALSPOINT. *ORM Overview* [online]. 2014. vyd. 2014 [cit. 2015-03-29]. Dostupné z: http://www.tutorialspoint.com/hibernate/orm_overview.htm.

21. PICHLÍK, Roman. INTERVAL.CZ. *Spring Framework* [online]. 2005. vyd. 2005 [cit. 2015-03-29]. Dostupné z: https://www.interval.cz/clanky/spring-framework-predstaveni-j2ee-lightweight-kontejneru/.

22. ORACLE A/NEBO JEJÍ DCEŘINÉ SPOLEČNOSTI. *The Java EE 6 Tutorial* [online]. 2013. vyd. 2013 [cit. 2015-03-29]. Dostupné z: http://docs.oracle.com/javaee/6/tutorial/doc/gipjf.html.

23. FILE-EXTENSIONS.ORG. *SER file extension - Java serialized object file* [online]. 2000. vyd. 2000 [cit. 2015-03-29]. Dostupné z: http://www.file-extensions.org/ser-file-extension.

6.2 Seznam zkratek

API - Application Programming Interface

SQL - Structured Query Language

JRE - Java Runtime Environment

Java SE - Java Platform, Standard Edition

Java EE - Java Platform, Enterprise Edition

JDK - Java Development Kit

JMS - Java Messaging Services

JSP - Java Server Pages

JSF - Java Server Faces

JPA - Java Persistence API

JDBC - Java DataBase Connectivity

ORM - Object-relational mapping

EJB - Enterprise Java Beans

URL - uniform resource locator

(X)HTML - Extensible Hypertext Markup Language

CSS - Cascading Style Sheets

ODBMS - object-oriented database management system

RDBMS - elational database management system

GUI - Graphical user interface

MVC - Model – View - Controller

HTTP - Hypertext Transfer Protocol

HTTPS - HTTP Secure

CSV - Comma-separated values

ASC - .asc is computer filename extension used for some text files

XML - Extensible Markup Language

6.3 Seznam obrázků

7 Přílohy

7.1 Zdrojový kód vzoru Singleton

```
public class Singleton {

     private static Singleton instance;

     //Vytvoření soukromého konstruktoru
     private Singleton() { }

     //Metoda pro vytvoření objektu jedináček
     public static Singleton getInstance() {
     //Je-li proměnná instance null, tak se vytvoří objekt
         if (instance == null) {
             instance = new Singleton();
         }
         //Vrátíme jedináčka
         return instance;
     }

     //Použiti
     public static void main(String[] args) {
         Singleton objekt = Singleton.getInstance();
     }
 }[Zdroj: programujte.com]
```

7.2 Zdrojový kód vzoru Messenger

```
class Puntik{ // přepravka
    public final int x;
    public final int y;

    public Puntik(int x, int y){
        this.x = x;
        this.y = y;
    }
}

class Platno{
    /**
     * metody, definice, konstruktory
     */

    public nakresliPuntik(Puntik p){
        /* implementace kreslení */
    }

    public nakresliPuntik(int x, int y){
```

```
        /* implementace kreslení */
    }
}

public class Main{
                    public static void main(String[] args) {
                // vytvoříme instanci plátna
                Platno pl = new Platno();
                // budeme kreslit na souřadnice 5,5
                Puntik puntik = new Puntik(5,5);

                // s využitím přepravky
                pl.nakresliPuntik(puntik);
                //s jiným využitím přepravky
                pl.nakresliPuntik(puntik.x,puntik.y);
                // špatně, bez využití přepravky
                pl.nakresliPuntik(5,5);
                    }
} [Zdroj: programujte.com]
```

7.3 Zdrojový kód vzoru Utility Class

```
        final class KnihovniTrida{
            private KnihovniTrida(){} // privátní konstruktor

            public static int secti(int a, int b){
                return a + b;
            }

            public static double umocniOdmocni(double a){
                return Math.sqrt(a*a);
            }
        }

        public class Main{
            public static void main(String[] args) {
                KnihovniTrida.secti(1, 1);
                KnihovniTrida.umocniOdmocni(-5.0);
            }
        } [Zdroj: programujte.com]
```

7.4 Zdrojový kód vzoru Static factory method

```
class Dum {
   /*
    * Definice různých atributů.
    */
    private final int pocetPoschodi;

    private Dum(int pocetPoschodi){
```

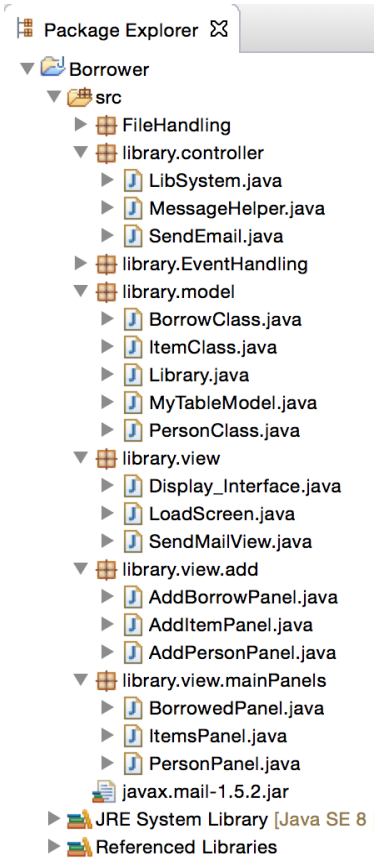
```
        this.pocetPoschodi = pocetPoschodi;
    }

    public static Dum getJednoposchodovyDum(){
        return new Dum(1);
    }

    public static Dum getDvouposchodovyDum(){
        return new Dum(2);
    }
}

public class Domy {

    public static void main(String[] args) {
        Dum prvni = Dum.getJednoposchodovyDum();
        /*
         * Další práce s jednoposchoďovým domem.
         */
    }
}
```
[Zdroj: programujte.com]

7.5 Zdrojový kód vzory Servant

```
interface IMeritelny {

    public double getVyska();

    public double getVaha();
}

class Zena implements IMeritelny {

    private double vaha;
    private double vyska;

    @Override
    public double getVaha() {
        return vaha;
    }

    @Override
    public double getVyska() {
        return vyska;
    }
}

class Muz implements IMeritelny {

    private double vaha;
    private double vyska;
```

```
    @Override
    public double getVaha() {
        return vaha;
    }

    @Override
    public double getVyska() {
        return vyska;
    }
}

class Meric {

    public double BMI(IMeritelny objekt) {
        double bmi = objekt.getVaha() / (objekt.getVyska() *
objekt.getVyska());
        return bmi;
    }

}

public class UkazkaServant {

    public static void main(String[] args) {
        Zena z = new Zena();
        Muz m = new Muz();

        Meric meric = new Meric();
        double bmiZeny = meric.BMI(z);
        double bmiMuze = meric.BMI(m);
    }
}[Zdroj:programujte.com]
```

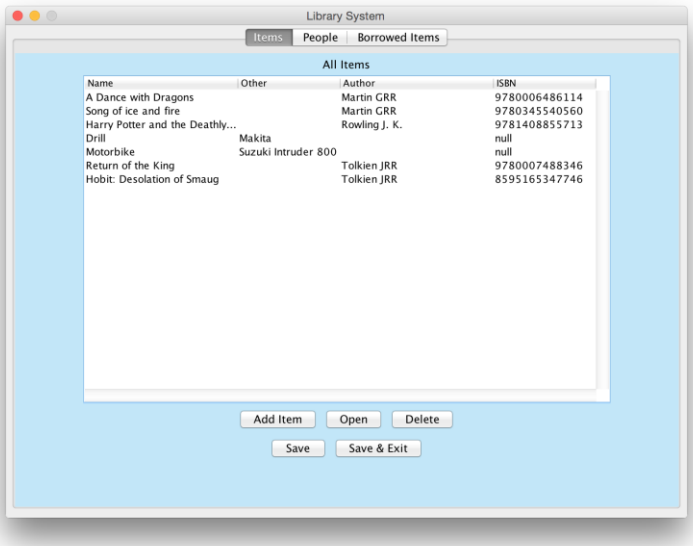
7.6 Package Explorer projektu



[Zdroj: Autor]

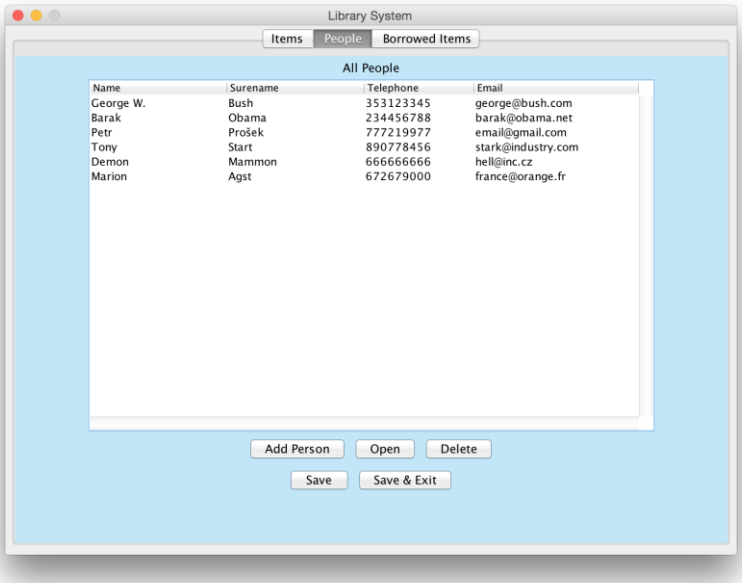
7.7 Ukázka aplikace v chodu

7.7.1 Items



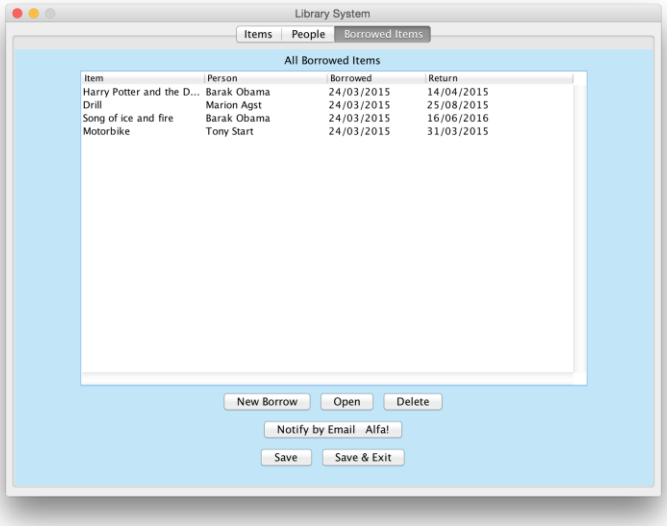
[Zdroj: Autor]

7.7.2 People



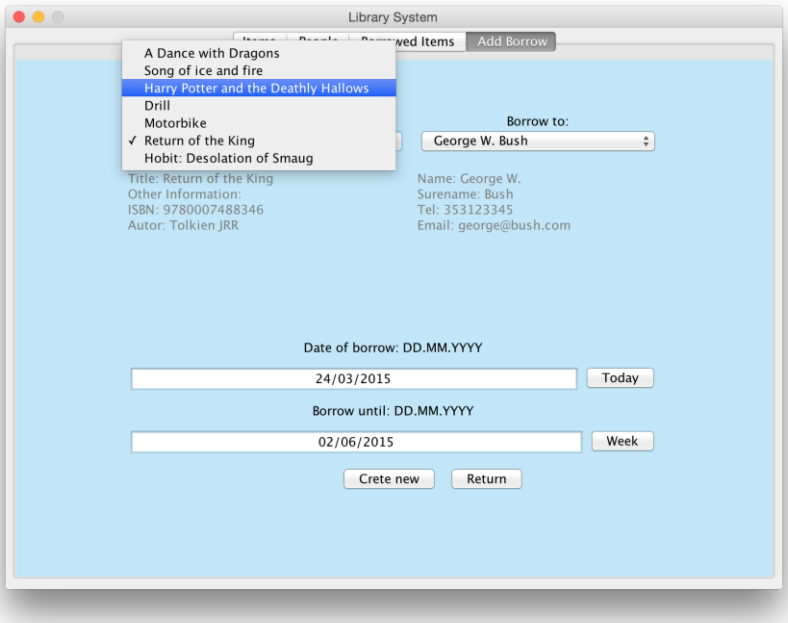
[Zdroj: Autor]

7.7.3 Borrowed Irems



[Zdroj: Autor]

7.7.4 Add Borrow



[Zdroj: Autor]