VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

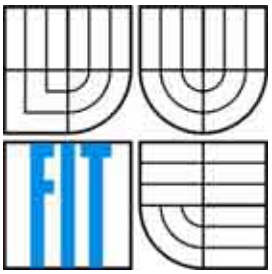
NÁVRH A ANALÝZA REZERVAČNÍHO SYSTÉMU PRO VÝUKU

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE MILAN JIŘÍČEK
AUTHOR

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRH A ANALÝZA REZERVAČNÍHO SYSTÉMU PRO VÝUKU

DESIGN AND ANALYSIS OF RESERVATION TEACHING SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — MILAN JIŘÍČEK
AUTHOR

VEDOUCÍ PRÁCE — ING. JIŘÍ KRAJÍČEK
SUPERVISOR

BRNO 2008

Abstrakt

V této bakalářské práci je detailně popsáno, jakým způsobem se analyzují a následně navrhují informační systémy. V menší míře je také věnována pozornost implementaci takového systému. Na konkrétním případu rezervačního systému pro výuku jsou znázorněny techniky postupu při návrhu pomocí jazyka UML. Samotný informační systém potom slouží pro evidenci událostí, ať už pravidelné výuky či jednorázových akcí. Jednotlivý uživatelé mohou provádět činnosti, které jsou spjaty se správou těchto událostí, ale i např. osob nebo místností.

Klíčová slova

Informační systém, databázový systém, databáze, rezervační systém, rozvrh, návrh informačního systému, analýza, návrh, OOP, UML, MVC, MDA, CASE systém, softwarové inženýrství, návrhové vzory, diagram, pohledy na systém, objekt, třída, polymorfismus, dědičnost, PHP, SQL, PostgreSQL, XHTML, CSS, Javascript.

Abstract

In this bachelor's thesis there is a detailed description of an information systems analysis and design. An attention paid to implementation of such system is also specified in further details. Techniques of procedure designing of these systems using UML are described on a specific case of reservation teaching system. This information system works with specific events, either a regular class or a one-time event. Particular users of this system are able to carry out actions, which are linked with event management as well as with subjects or classrooms.

Keywords

Information system, database system, database, reservation system, schedule, information systems design, analysis, design, OOP, UML, MVC, MDA, CASE system, software engineering, design patterns, diagram, system model views, object, class, polymorphism, inheritance, PHP, SQL, PostgreSQL, XHTML, CSS, Javascript.

Citace

Milan Jiříček: Návrh a analýza rezervačního systému pro výuku. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Návrh a analýza rezervačního systému pro výuku

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením ing. Jiřího Krajíčka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Milan Jiříček
12. května 2008

Poděkování

Na tomto místě bych rád poděkoval panu ing. Jiřímu Krajíčkovi za pomoc a rady při tvorbě této práce.

Obsah

Úvod

Informační technologie v současnosti zažívají obrovský rozmach, který má důsledek i na informační systémy, které zákazníkům, potažmo uživatelům, usnadňují práci prováděnou v nejrůznějších odvětvích průmyslu, bankovnictví, státní správy, apod. Především díky snadné manipulaci s daty, kterou zajišťují databázové servery, dochází k radikálním změnám ve firmách a institucích, které stále více využívají těchto možností nabízených na trhu s informačními systémy. Vedení takových společností si stále více uvědomují důležitost role, kterou informační systémy v byznyse hrají a budou hrát. Není žádným tajemstvím, že právě investice a přechod na modernější technologie často ušetří firmám značné časové, ale i humanitní prostředky, a tím ve své podstatě i finanční náklady.

S již zmíněným rozšiřováním informačních systémů však přichází nutnost tyto systémy stále zdokonalovat a pracovat na jejich rozšiřování a přizpůsobování pro stávající a budoucí uživatele. Ti jsou se stále se zvyšujícím procentem počítačově gramotné populace čím dál více náročni na kvalitu a snadnou obsluhovatelnost informačních systémů.

Právě proto se v nepříliš vzdálené minulosti začaly postupně objevovat určité standardy usměrňující vývoj těchto systémů do již zaběhnutých kolejí, a tím unifikovat návrh, který bude jednak srozumitelný a také patřičně dokumentovatelný. Tento proces má za výsledek snadnější implementaci výsledného produktu, především díky detailní analýze a propracovanosti zadaných problémů.

1.1 Přechod na OOP

Jak již bylo zmíněno, zvyšující se požadavky na software s sebou přinesly přechod na nové, v některých případech i experimentální, metody návrhu a implementace výsledného kódu. Tou zřejmě nejradikálnější změnou v doposud povětšinou agilním a funkcionálním přístupům k tvorbě softwaru se zdá být objektově orientované programování, kde je kvalitní a správně zpracovaný návrh jednou z nejdůležitějších částí práce na projektu.

Tato práce má tedy za úkol srozumitelně přiblížit, jak takový systém analyzovat, posléze navrhnou pomocí moderních modelovacích nástrojů a nakonec také implementovat. Jednotlivé stádia tvorby informačního systému jsou rozděleny do jednotlivých kapitol, kde je každá z těchto částí detailně popsána a vysvětlen je význam dané úrovně. Na konkrétním případu rezervačního systému pro výuku je předvedeno, které aspekty jsou při tvorbě software důležité, na jaké věci je dobré si dát pozor a jak postupovat při problémech s návrhem, popř. s následnou implementací.

2 Požadavky informačního systému

Velmi důležitou část práce na informačním systému představuje analýza požadavků na takový systém. Umožňuje nám předem se připravit na problémy a překážky, které se mohou vyskytnout při samotném návrhu a implementaci aplikace ve zvoleném programovacím jazyce. Tím samozřejmě velmi usnadní již zmíněné procedury a proto je vhodné tuto část nepodcenit.

2.1 Analýza systému

Rezervační systém pro výuku, jak sám název napovídá, je systém určený především pro použití v institucích, kde je zapotřebí určitý systém evidence událostí spojených s vyučováním jakékoliv látky, ať už se jedná o jednorázové akce nebo pravidelné kurzy. Bude tedy nutné tyto informace rozlišovat a posléze také ukládat do databáze. Důležité je především uchovat záznamy o délce dané události, ale také o jejím začátku.

2.1.1 Uživatelé systému

Do informačního systému budou mít přístup kromě administrátorů také učitelé, popř. přednášející a studenti. Každé ze zmíněných skupin přísluší určitá práva, která určují, jaké operace může daný uživatel v systému provádět. Konkrétněji se jedná o přidávání dat, jejich následnou úpravu, a v neposlední řadě také mazání existujících údajů, především těch nevyhovujících.

2.1.1.1 Administrátoři

Administrátor má oprávnění zcela neomezená, může tedy přidávat/upravovat/mazat všechny údaje, které jsou uložené v databázi. Ať už se jedná o kurzy, místnosti, studenty, ale i např. učitele. Mají tedy plnou kontrolu nad celým systémem.

2.1.1.2 Učitelé

Omezenější pravomoci mají potom učitelé. Ti mohou přidávat/upravovat/mazat kurzy a jednotlivé události, nikoliv však již zmíněné další přednášející, studenty a místnosti.

2.1.1.3 Studenti

Nejméně zásahů do systému mohou logicky provádět studenti. Ti se registrují na jednotlivé události, např. pokud se chce zúčastnit jednorázové akce typu zkouška či pravidelně docházet na přednášky. V tomto případě tedy dochází k velmi omezené úpravě dat v podobě událostí. Zde práva studenta na změny v systému končí.

2.2 Případy užití

Především díky zpracované analýze systému a jeho uživatelů můžeme dále zpracovávat požadavky kladené na rezervační aplikaci. K tomu je vhodné využít modelovacího jazyka UML. Bližší informace o tomto jazyce jsou uvedeny v další kapitole.

2.2.1 Diagram případů užití

První úkol, tj. vymezení hranic systému, je v UML podpořeno dynamickým pohledem modelovaným prostřednictvím diagramů případů užití, který je znám pod celou řadou dalších názvů, jako například diagram typových činností, jednání atp. Je jedním z nejčastěji uváděných diagramů při návrhu informačního systému, v anglickém jazyce označovaný jako „use case diagram“. Ten zobrazuje přesné akce a funkce, které budou následně implementovány do systému, a můžeme si tak udělat představu o způsobu manipulace s daty ve finálním produktu. Tento diagram, obecně model, značí základní vztah systému k jeho okolí, tzv. **účastníkům** (actors). Účastníci jsou nejčastěji samotní uživatelé systému, nicméně mohou jimi být obecně libovolní externí činitelé stojící mimo modelovaný systém. Může se tedy jednat například o další hardware se softwarovými prostředky, jiné informační systémy, s kterými má nový systém spolupracovat, instituce, které se systémem komunikují atd. Každý případ užití pak představuje jeden z možných způsobů použití systému, jednu z možných cest komunikace účastník - systém. Konkrétním případem užití může být například použití systému za účelem zjištění aktuálně volného termínu přednášky, úprava názvu kurzu, výpis všech studentů atp. Jednoduše řečeno, případy užití simulují použití reálného systému externími účastníky. Všechny tyto případy jsou naznačeny na obrázku. (1.1) V rámci tvorby těchto diagramů modelujeme vztah systému a jeho okolí, nikoliv vzájemnou interakci, tj. způsob, jakými jsou jednotlivé případy užití zajištěny interními funkcemi systému.

2.2.2 Úskalí modelování případů užití

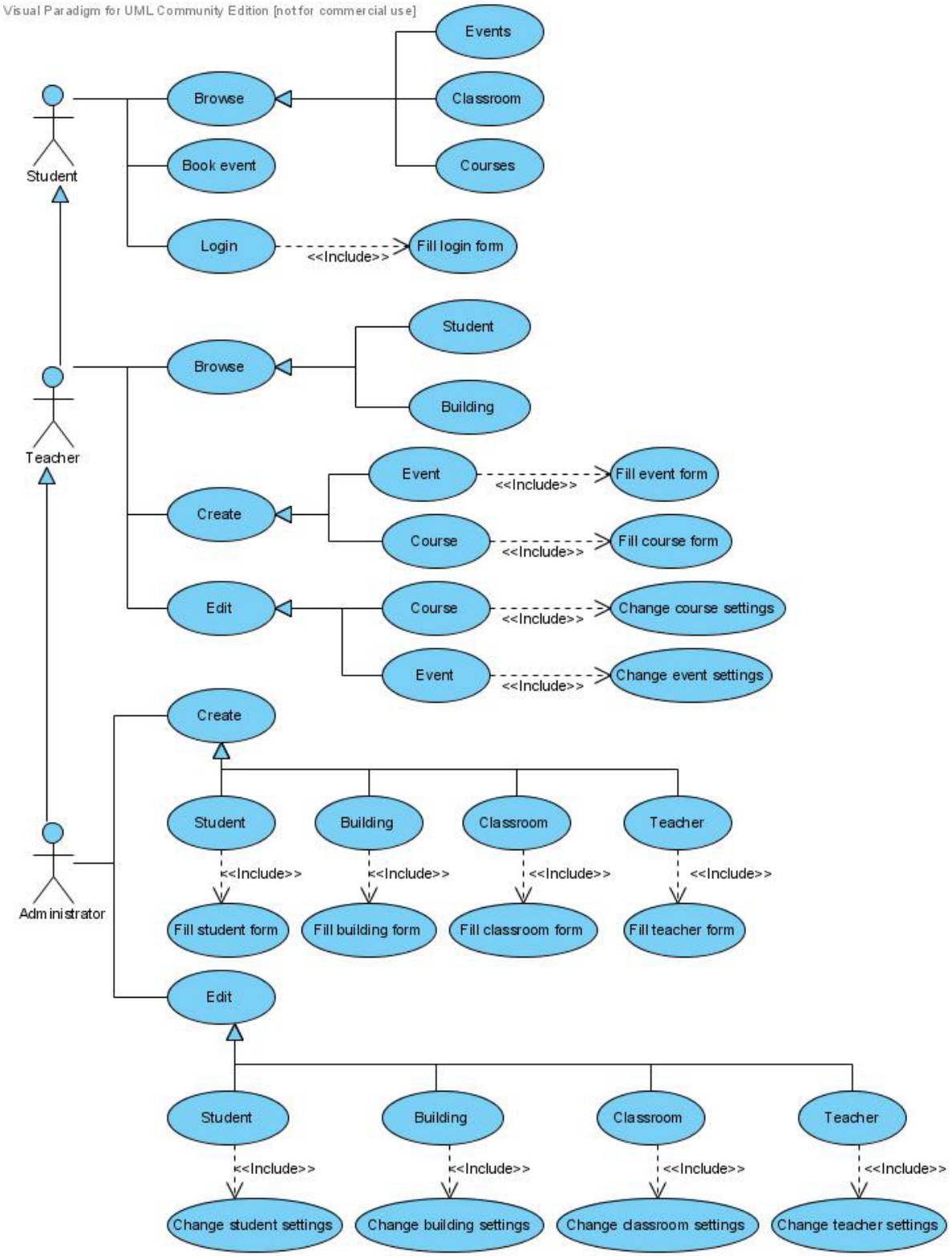
Největší úskalí při používání tohoto prostředku spočívá v odhadnutí „správné úrovně abstrakce“. Občas se může stát, že jsou případy užití moc obecné a nelze s nimi bez patřičného popsání efektivně pracovat. Na druhou stranu je nutné vyvarovat se druhého extrému, kterým je zneužívání tohoto nástroje pro funkcionální rozklad, který nemá s „use case“ modelováním nic společného. Historie praví, že systém postavený na funkcích, jež má zajišťovat, spíše než na objektech, které jej tvoří, je mnohdy nestabilní z hlediska měnících se uživatelských požadavků. Ačkoliv toto riziko a problémy s ním spojené objektový přístup neeliminuje bezezbytku, je přeci jenom možné nežádoucí efekty vyplývající ze změny požadavků částečně potlačit. V ideálním případě se změna promítne v místě, kterého se bezprostředně týká, v horším případě ovlivní dílčí změna mnoho součástí systému. S

rostoucím počtem následných úprav pak logicky roste i riziko z nejnepříjemnějších, čímž je nekonzistence modelů.

Praxe ukazuje, že z hlediska používání UML je zvládnutí tvorby těchto modelů jedním z nejobtížnějších úkolů, který navíc díky své povaze výraznou měrou ovlivňuje podobu výsledného produktu. Přestože je pomocí tohoto prostředku možné rozdělit systém na dílčí oblasti, není tento prostředek určený na komplexní modelování architektury. Snad více než kde jinde je v tomto typu modelu nutno zvolit správný soulad mezi úplností a přehledností. Nepříliš adekvátním použitím mohou diagramy případů užití velice rychle narůstat na objemu a stanou se z nich nezvladatelné kolosy, které již neslouží svému původnímu účelu, tzn. pro základní členění systému.

Ve spojení s dalšími prostředky pro dynamické modelování je tvorba diagramů případů užití nezbytným prostředkem pro nalezení objektů participujících v modelovaném systému. Rozdělení celého systému na jednotlivé případy užití přináší kromě vymezení hranic systému i možnost zpracovávat jednotlivé případy užití odděleně a částečně tak realizovat iterativní přírůstkový životní cyklus. Skrze tvorbu případů užití jsou samozřejmě definovány i základní uživatelské požadavky.

Použitá literatura v této podkapitole: [7], [11] a [12].



Obrázek 2.1: Diagram případů užití

3 Návrhová metodologie OOP

V poslední době se pozornost obrací především k metodám postaveným na doporučené notaci UML (Unified Modeling Language), která slouží jako sjednocující standard mezi analytiky, návrháři a programátory používajícími objektové technologie. Bezezbytku všechny ze soudobých metod objektové analýzy a designu se honosí označením „UML compliant“. Tento standard nabývá se stále rostoucím počtem informačních systémů na důležitosti a jistě za sebou ponechává doby, kdy se jednalo o pomíjivou záležitost. Zkrátka, v moderním procesu tvorby informačního systému má UML již své pevné místo.

3.1 Historie

Nejzazší historie UML sahá do listopadu 1994, kdy své síly spojili dva tehdejší objektoví guru Grady Booch a James Rumbaugh. V roce 1995 se k oběma tvůrcům připojila další výrazná osobnost, Ivar Jacobson, autor vcelku úspěšné metodiky Objectory/OOSE. Rychle došlo ke zcela zásadní verzi 1.0, která později sehrála klíčovou roli. Bleskovým přejmenováním Unified Method na Unified Modeling Language byla definitivně stvrzena úloha UML jakožto prostředku pro grafickou tvorbu objektových modelů a nikoli jako ucelené metodiky definující i odpovídající proces tvorby informačních systémů. Notace UML 1.0 byla rovněž předložena konsorciu OMG, které ji ještě téhož roku adoptovalo jako svůj doporučený standard. Cesta k akceptování UML širokou odbornou veřejností se tímto zdála být otevřena. UML verze 1.3 spatřilo světlo světa roku 1999.

Dnes je UML nejrozšířenější používanou objektovou notací, na jejíchž základech stojí většina CASE systémů. Za pár měsíců své existence doznalo UML několik rozšíření a modifikací jako například zavedení OCL (Object Constraint Language) a prostředků pro business modelování. Zatím poslední oficiální verzí je OMG UML 2.0 z roku 2005.

Použitá literatura v této podkapitole: [3].

3.2 Prostředky UML

Jak již bylo patřičně zdůrazněno, UML není ucelená metodika definující i nezbytný proces tvorby informačních systémů, ale pouhá množina nástrojů, které se v tomto procesu používají. Teprve nasazením odpovídající objektové metodiky, která definuje kdy, kým a jak jednotlivé nástroje používat, můžeme vytěžit z UML maximum. Z objektových metodik vytvořených přímo nad nástroji UML jmenujme například Unified Process či Catalysis, vhodnou pro tvorbu systémů stojících na principech CBD (Component-Based Development). Jak je vcelku zaběhnutým zvykem (otázkou je jestli dobrým či nikoliv), odlišuje UML nástroje pro modelování statické a dynamické stránky

systému. Zjednodušeně řečeno, statický pohled charakterizuje, jak systém vypadá, zatímco dynamický definuje, jak se systém chová v čase a jak reaguje na nejrůznější podněty. Souhrnně hovoříme o architektuře systému. Jednotlivé nástroje a prostředky obsažené v UML nám při respektování tohoto pravidla dovolí modelovat vytvářený systém ze všech možných stran a na všech možných úrovních. Vhodnou kombinací těchto nástrojů, a to je na zvládnutí objektového modelování to nejtěžší, lze dosáhnout vcelku přehledných modelů dílčí i celkové architektury systému.

Začneme-li etapou analýzy, o jejíž nutnosti soudný člověk nemůže pochybovat, pak je prvním z úkolů vymezení hranic modelovaného systému. Tento postup je samozřejmě analogický s procesem zkoumání jakéhokoliv předmětu. Zpravidla první a určující činností je vždy tento zkoumaný předmět poznat, čímž se zabývá obecně analýza, a následně se dají na základě těchto poznatků dělat další rozhodnutí, která mohou mít naopak povahu syntézy (etapy designu a implementace).

Po úspěšném vymezení hranic systému se již víceméně pouštíme do obtížného úkolu nalezení vhodných objektů a vztahů mezi nimi, které tvoří danou oblast. Takto vytvořený objektový model je poté různě zpracováván až do podoby předloh pro implementaci, kterou nám může automatickým generováním kódu usnadnit použitý CASE nástroj. V následujících odstavcích je naznačeno, jak nám může UML pomoci při naplňování tohoto zjednodušeného schématu.

3.3 UML v podnikovém prostředí

Největší koncepční výtkou na adresu UML je přílišná orientace na použití při tvorbě software, což se někomu může sice zdát jako logický důsledek předešlého vývoje, někomu jinému však jako poměrně výrazná slabina a nevýhoda. Tvorba informačního systému není zdaleka pouhá tvorba softwaru.

Při prvním seznámení může UML působit vcelku kompaktním dojmem, ovšem při hlubším studiu a především dlouhodobějším používání narazí jeho uživatel na pár nekonzistencí. Na druhou stranu nesporně platí, že UML hraje roli sjednocujícího komunikačního prostředku mezi objektovými modeláři, kde sehrává opravdu zásadní úlohu. Nástroje obsažené v balíku UML tvoří vcelku ucelenou množinu efektivních prostředků dobře sloužících svému účelu, jejichž zvládnutí však vyžaduje dostatek snahy a trpělivosti. Po úspěšném osvojení UML dobře poslouží jako balík základních nástrojů pro tvorbu informačních systémů.

Objektové metodiky jsou účinnými zbraněmi především proti nadměrné komplexnosti a jejím důsledkům. Bohužel, samotné UML jako notace pro tvorbu a dokumentaci objektových modelů má občas s komplexností problémy. To je v některých případech dáno především přístupem jeho tvůrců, který je někdy zbytečně maximalistický, což je mnohdy spíše ke škodě věci. Na druhou stranu poslední instancí je uživatel UML (analytik, návrhář, apod.), který by se měl rozhodnout jaké nástroje a k čemu používat.

Asi vůbec nejdůležitějším trendem je používání objektových technologií jak pro tvorbu informačních systémů, tak i pro aktivity spojené s BPR (Business Process Reengineeringu).

Informační technologie dnes sehrávají v podnikovém prostředí zásadní roli a je více než zřejmé, že nelze tvorbu informačních systémů od BPR absolutně oddělit. Naskýtá se zde pak aktuální otázka, proč nepoužívat jednotný nástroj, který by byl schopen vhodně podpořit oba dva pohledy na stejný problém. Tímto nástrojem se však sotva stane UML v dnešní podobě tak, jak ho známe. Pravděpodobně by bylo nutné přehodnotit směr, kterým se standard UML před pár lety vydal. Jedno je však přesto jisté, objektové modelování zaujímá v informačních technologiích a informačních systémech své pevné místo a jen stěží lze proti tomu něco namítat.

3.4 Základy OOP

Termín OOP znamená objektově orientované programování (dále už jen OOP), což je způsob programování, kdy chápeme procesy jako entity. Tedy zkráceně řečeno, nepotřebujeme vědět, jak daný program funguje, ale musíme vědět, jak tento program používat.

3.4.1 Objekty a třídy

Na rozdíl od procedurálního programování, v OOP jsou data a funkce navzájem svázány do struktury, nazývané objekt. Třída je šablonou objektu, volání třídy se pak označuje jako instance. Pojmy instance a objekt jsou v podstatě synonyma. Často však mluvíme o konkrétním objektu jako o instanci. Termín objekt se používá, pokud mluvíme o objektech obecně. Rozdíl mezi objektem a třídou lze tedy popsat tak, že objekt je vyjádřením třídy v reálném světě.

3.4.2 Viditelnost

Občas chceme definovat vlastnostem i metodám viditelnost pomocí zvláštních klíčových slov. Atributy (vlastnosti) a metody tedy dělíme na „public“, „protected“ a „private“. Vlastnost nebo metoda určená pomocí slova „public“ je přístupná všem uživatelům třídy. Klíčové slovo „protected“ (v češtině chráněný) před metodou nebo vlastností třídy znamená, že je vlastnost přístupná uživatelům třídy (rodiče) a podtřídy, která tuto třídu dědí. Soukromá metoda nebo vlastnost předznačená klíčovým slovem „private“ je přístupná pouze uvnitř třídy, čímž dosahujeme oddělenosti, neboli bezpečnosti, částí kódu.

3.4.3 Dědičnost

V některých případech potřebujeme vytvořit třídu podobnou té, kterou už máme definovánu. Bylo by zcela zbytečné, kdybychom vše přepisovali. Stačí totiž pouze pozměnit stávající třídu. Proces, kdy vytváříme novou třídu, která přebírá libovolné metody a atributy, se právě nazývá dědičnost.

3.4.4 Polymorfismus

Slovo polymorfismus znamená něco jako „mnohotvarost“. Polymorfismus je v programování velmi obecný pojem. V souvislosti s OOP se jedná o to, že instance různých tříd na stejný podnět (na vyvolání stejné metody) reagují různě. Instance více tříd poskytují svému okolí stejnou službu, ale každá instance na vyžádání této služby provede něco jiného.

3.5 Stavba UML

V jazyce UML rozlišujeme tři základní stavební bloky (tzv. „building blocks“), pomocí nichž vytváříme výsledné modely.

3.5.1 Předměty

Název vznikl z anglického slova „things“. Do předmětů jsou řazeny samotné elementy modelu. Oblast předmětů je dále členěna na podkategorie.

3.5.1.1 Strukturní předměty

Mezi ně patří strukturní předměty (structural things), které obsahují subjekty modelu. Subjekt modelu nese statické informace o softwarovém artefaktu a je většinou vyjádřen podstatným jménem (třída, komponenta, uzel).

3.5.1.2 Dynamické předměty

V podkategorii pro dynamické předměty (behavioural things) se nalézají předměty, které zachycují dynamické chování systému (aktivita, stav).

3.5.1.3 Seskupení

K sdružování sémanticky příbuzných předmětů se používají takzvané balíčky, pro něž je vytvořena samostatná kategorie předmětů vyjadřujících seskupení (grouping things).

3.5.1.4 Anotační předměty

Ke každé části modelu je možné zadat libovolné množství poznámek. Poznámky patří do kategorie anotačních předmětů (Annotational Things).

3.5.2 Relace

Z anglického „relations“ potom vznikly české relace. Předměty neleží vedle sebe jako chaotické shluky nesouvisejících informací, ale jsou vzájemně propojeny komplexním poutem vztahů. Vztahy v modelu ukazují, jak spolu jednotlivé předměty souvisí. UML rozeznává několik typů vztahů.

3.5.2.1 Asociace

Asociace, neboli „association“, je obecná souvislost předmětů. UML přesně definuje asociaci jako „měnící se populaci vztahů daných propojením objektů“. Speciálními variantami asociace jsou kompozice a agregace.

3.5.2.2 Závislost

Změna v jednom předmětu způsobí změnu v jiném předmětu nebo mu poskytne požadovanou informaci. Například když třída pro odesílání emailu v aplikaci získá šablonu emailu z třídy globálních konfiguračních parametrů, měli bychom tento vztah zachytit jako závislost. Dobře zmapované závislosti nás neustále informují o celkové míře provázanosti jednotlivých částí modelu. Pokud závislosti v modelu začnou připomínat propletené křižovatky, je na čase vztahy mezi třídami přehodnotit, protože jde o jasný signál blížícího se krachu implementovaného systému při prvním pokusu o jeho další rozšíření.

3.5.2.3 Generalizace

Generalizace, neboli „generalisation“, značí, že jeden předmět je specializací jiného předmětu. Vztahy generalizace a specializace jsou v objektově orientovaném modelování velmi důležité, protože jejich neadekvátní použití předznamenává většinou krizi celého návrhu i jeho implementace.

3.5.2.4 Realizace

Realizace, neboli „realization“, je druh vztahu, při němž jeden předmět představuje dohodu, za jejíž plnění je odpovědný jiný předmět. Například z jazyků JAVA, C# a dalších známe rozhraní, která reprezentují dohodu, jejíž plnění je požadováno po třídách, jež rozhraní implementují.

3.5.3 Diagramy

Model v UML se skládá z různých diagramů, jež představují průhledy na různé části sémantického základu navrhované aplikace. Sémantickým základem je souhrn specifikací aplikace, který vymezuje teritorium, v němž se může návrh pohybovat. Diagram ve vizuální formě prezentuje právě jednu konkrétní událost o aplikaci. Žádný dvourozměrný diagram nemůže zachytit komplexní aplikaci v celku, ale soustředí se vždy právě na jeden důležitý aspekt.

3.5.4 UML pohledy na systém

Jazyk UML specifikuje pět základních pohledů na vytvářený systém. Pohledy jsou konkretizovány v různých typech diagramů, jejichž detailní popis bude předmětem další kapitoly.

3.5.4.1 Pohled případů užití

Tento pohled byl podrobněji probrán již v předcházející kapitole. V případech užití jsou vyjádřeny základní požadavky kladené na systém. Veškeré další pohledy se pohybují v oblasti vymezené pohledem případů užití.

3.5.4.2 Logický pohled

Logický pohled se zabývá zejména pojmy z problémové domény zadavatele a jejich vzájemnými statickými vztahy.

3.5.4.3 Procesní pohled

Procesní pohled se soustřeďuje na chování systému, které musí splňovat požadavky a omezení z případů užití, jež jsou kladeny na průběh procesů. Jinak řečeno, jedná se o procesně orientovaný doplněk logického pohledu.

3.5.4.4 Implementační pohled

Implementační pohled zachycuje fyzické rozdělení aplikace na samostatné komponenty a jejich závislosti.

3.5.4.5 Pohled nasazení

Pohled nasazení mapuje komponenty na množinu fyzických výpočetních uzlů v cílovém prostředí.

3.6 Řízená rozšíření jazyka UML

Při navrhování různých typů aplikací se neustále potýkáme s požadavkem na vyjádření netriviálních faktů, pro jejichž sémantiku není v jazyce UML přímá podpora. Řízená rozšíření překlenují propast mezi všeobecně známými standardními elementy jazyka UML a naší snahou přiměřeným způsobem zohlednit ve vytvářeném modelu charakteristické a specifické aspekty problémové domény.

Samotné výrazové prostředky UML nejsou dostatečně silné na to, aby bylo možné s jejich pomocí zachytit zásadní vztahy a především pravidla, která jsou nezbytnou součástí modelovaného systému. Diagramy tříd například nenesou informace o pravidlech, které musí platit pro zajištění konzistence celkového modelu.

3.6.1 Stereotypy

Stereotypy jsou založeny na existujících elementech jazyka UML a upravují jejich význam. Každý stereotyp se vytváří přidáním dvojice lomených závorek s názvem stereotypu k původnímu elementu. Často používané stereotypy jsou integrální součástí jazyka UML. Úpravou významu elementu se rozumí zúžení významu nebo úplná redefinice jeho role v diagramech.

Zajímavější jsou stereotypy měnící radikálně sémantiku elementu, na nějž jsou aplikovány. Například často používaný stereotyp <<interface>> využívá grafickou notaci tříd pro vyjádření rozhraní. Změněný význam vyjádřený stereotypem můžeme zvýraznit použitím jiné barvy nebo dokonce změnou tvaru grafického symbolu. Stereotypy se změněnými grafickými symboly jsou často využívány v diagramu nasazení, kde fyzické uzly systému místo neforemných „kvádrů“ reprezentují symboly počítačů, laptopů či tiskáren.

3.6.2 Omezení

Omezení, neboli „constraints“, se v UML používají pro vyjádření podmínek, které musí být v daném kontextu vždy pravdivé. Omezení se zapisují do složených závorek a jejich syntaxe by měla být srozumitelná pro všechny členy týmu.

Všechna omezení v návrhu musí být v kódu transformována na explicitní předpoklady, jejichž neočekávané porušení musí být vyhodnoceno jako přechod aplikace do nestabilního stavu, spojený se zapsáním chyby do aplikačního „logu“ a okamžitým ukončením aplikace.

3.6.3 Označené hodnoty

Poslední mechanismus, jak upravovat elementy v jazyce UML, představují označené hodnoty (anglicky „tagged values“). Označené hodnoty se shodně s omezeními zapisují do složených závorek, ale jejich syntaxe vyžaduje použití názvu atributu s přidruženou hodnotou. Pokud k elementu přidáváme více označených hodnot, musíme je oddělit čárkou. Označené hodnoty vyjadřují další podstatné informace o elementu a také mohou zavádět nové vlastnosti u stereotypu.

3.7 Architektura MDA

Architektura řízená modelem (Model Driven Architecture – MDA) je rámec životního cyklu vytvořený skupinou Object Management Group (OMG), což je organizace, která si klade za cíl standardizaci v oblasti objektově orientovaného přístupu. Jí schválené standardy jsou akceptovány jako průmyslové standardy. Jedním z nich je i jazyk UML. MDA je snahou o použití jazyka UML jako spustitelné (executable) specifikace, tj. specifikace, ze které je možné vygenerovat softwarový systém, který reprezentuje daný model v UML. Koncept MDA sice úzce souvisí s UML, avšak není na tento modelovací standard bezprostředně vázaný, neboť se dá aplikovat i jiným způsobem modelování, než je UML.

MDA je tedy především o zjednodušení, a tím i zlevnění údržby a rozvoje aplikací. Vzhledem k tomu, že právě sem směřuje větší část investic související s vývojem softwaru, je tato koncepce nakonec zajímavá i po finanční stránce.

Použitá literatura v této kapitole: [3], [5], [6], [8], [10] a [13].

4 Návrh systému

Metoda objektově orientovaného návrhu spočívá v konstrukci univerzálně využitelných prvků různého typu a v zajištění požadované funkce programu skládáním služeb poskytovaných těmito prvky. Objektově orientovaný návrh je postup, který vede k programovým architekturám založených na objektech, s kterými se manipuluje, spíše než na funkcích, které má zajistit.

4.1 Návrhové vzory

Návrhové vzory, neboli „design patterns", můžeme označit za způsob, jak řešit určitý problém nejen při návrhu objektově orientovaného (OO) systému. Existuje 23 základních vzorů, od kterých se odvozují další. Návrhové vzory byly poprvé popsány v knize „Design Patterns" od 4 autorů (Gama, Helm, Johnson, Vlissides - označováni jako „Gang of Four") [1]

4.1.1 Dělení vzorů

Vzory dělíme podle jejich účelu do tří základních kategorií.

4.1.1.1 Tvořivé vzory

Tyto vzory popisují způsob, jak skrýt implementaci objektů, aby nebylo nutné při jejich změně měnit systém.

4.1.1.2 Strukturální změny

Strukturální změny se zabývají propojením objektů. Jak propojení omezit nebo předejít změnám propojení.

4.1.1.3 Vzory chování

Vzory chování zapouzdřují určité procesy prováděné v systému.

4.1.2 Smysl návrhových vzorů

Objektově orientované vzory znázorňují vztahy mezi jednotlivými objekty a třídami, ovšem bez specifikace finálních aplikačních tříd a objektů. Jsou vytvářeny tak, aby co nejlépe sloužily k popisu, popřípadě jako šablona pro řešení problémů v mnoha různých situacích, nikoliv aby byly ihned převáděny na kód programu.

Návrhové vzory omezují složitost, poskytují hotové abstrakce a v neposlední řadě není nutné je znovu vymýšlet – jsou totiž obecným řešením obvyklých problémů, se kterými se v softwarovém inženýrství můžeme setkat. Navíc omezují možnost vniku chyb, standardizují detaily běžných řešení

a představují znalosti akumulované při řešení podobných situací. Mezi další výhody je možné zařadit např. zrychlenou komunikaci při návrhu, která navíc probíhá na vyšší úrovni abstrakce nebo nepotřebnost vymýšlení specifických návrhových alternativ. Samozřejmostí je, aby všechny zúčastněné osoby znaly běžné návrhové vzory.

Čím více času navíc se stráví návrhem, tím větší je šance, že návrh bude nejen dostatečný, ale také přiměřeně kvalitní. Zkušenosti dále praví, že nejvíce chyb se naskytne většinou v oblastech, kde máte pocit, že detail je dostačující.

4.1.3 Často používané vzory

Návrhové vzory se rozdělují na tři základní kategorie, které jsou stále uznávané.

4.1.3.1 Vytvářející vzory

Vytvářející vzory (anglicky „creational patterns“) řeší problémy související s tvorbou objektů v systému. Cílem těchto návrhových vzorů je popsat postup výběru třídy nového objektu a zajištění správného počtu těchto objektů. Z velké většiny se jedná o dynamická rozhodnutí učiněná za běhu programu. Zde je výčet vzorů zařazených do této kategorie: Factory Method Pattern (umožňuje vytvářet instance tříd odvozené od základní třídy bez nutnosti starat se o jednotlivé třídy jinde v kódu), Singleton Pattern (globální přístup ke třídě, která má pouze jednu instanci), Abstract Factory Method Pattern, Builder Pattern a Prototype Pattern.

4.1.3.2 Strukturální vzory

Strukturální vzory (structural patterns) představují skupinu návrhových vzorů, které se zaměřují na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Snahou je zpřehlednit systém a využít možností strukturalizace kódu. Mezi tyto návrhové vzory patří: Adapter Pattern (převádí jedno rozhraní na druhé), Bridge Pattern (vytváří most mezi rozhraním a implementací tak, aby bylo možné měnit jedno bez nutnosti zásahu do druhého), Composite Pattern, Decorator Pattern (umožňuje dynamicky přidávat objektu povinnosti bez nutnosti vytvářet podtřídy pro specifické konfigurace povinností), Facade Pattern (poskytuje konsistentní rozhraní ke kódu, který by jinak takové rozhraní nenabízel), Flyweight Pattern a Proxy Pattern.

4.1.3.3 Vzory chování

Vzory chování (behavioral patterns) se zajímají, jak již název napovídá, o chování systému. 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, která zajišťuje dosažení požadovaného výsledku. Mezi tento typ vzorů můžeme zařadit: Observer Pattern (synchronizuje objekty z určité množiny pomocí speciálního objektu zodpovědného za hlášení změn

objektů v množině), Strategy Pattern (definuje množinu algoritmů nebo chování, které je možné dynamicky zaměňovat), Chain Of Responsibility Pattern, Command Pattern, Interpreter Pattern, Iterator Pattern, Mediator Pattern, Memento Pattern, State Pattern, Template Pattern a Visitor Pattern.

Použitá literatura v této podkapitole: [1], [4], [9] a [14].

4.2 Architektura Model-View-Controller

Model-View-Controller (zkráceně MVC) je softwarová architektura, která rozděluje datový model samotné aplikace, uživatelské rozhraní a řídicí logiku do tří nezávislých komponent takovým způsobem, že modifikace některé z nich má minimální vliv na ostatní. V českém překladu reprezentuje tuto architekturu sousloví model-pohled-řadič. Tato architektura je ve velké míře využívána při vytváření prezentačně orientovaného systémů a již je publikována jako návrhový vzor.

MVC architektura je částečně postavena na principu návrhového vzoru „Observer“. Model představuje pozorovaný objekt, ke kterému se jako pozorovatelé přihlašují objekty z prezentační vrstvy a řadiče. Jestliže nastane změna v modelu (například změna dat), je o této skutečnosti informován řadič i zaregistrované prezentační objekty pomocí metody update(), která je definována v rozhraní „Observer“. Řadič může vlastnit referenci na více komponent modelu, ale i na více objektů z prezentační vrstvy. Návrhový vzor MVC byl vytvořen pro technologii „client-server“. Problém tohoto vzoru v internetové technologii je nemožnost okamžité notifikace prezentační logiky. Tato technologie pracuje na principu požadavek-odpověď, kdy zahájení komunikace vždy vychází z GUI (graphic user interface). Proto je tato vrstva informována o proběhlých změnách pouze pomocí odpovědí na její požadavky.

4.2.1 Model

Část model zodpovídá za správu vnitřní struktury, což je doménově specifická reprezentace informací, s nimiž aplikace pracuje. Model se však může lišit podle typu architektury, který daná aplikace používá. Jelikož se většinou jedná o nejsložitější vrstvu systému, která obsahuje množství komponent, je vhodné poskytnout přehledný přístup k této vrstvě pomocí ucelené sady aplikačních rozhraní (API). Z návrhových vzorů, které se hodí pro tento účel, můžeme použít „Facade“ ve spolupráci s „Command“ vzorem.

4.2.2 Pohled

Pohled, neboli „view“, převádí data reprezentovaná modelem do podoby vhodné k interaktivní prezentaci uživateli. V případě webových aplikací se pohled obvykle skládá z HTML a PHP/ASP/JSP popř. podobných technologií. HTML stránky vytvářejí statický obsah, zatímco ostatní zmíněné jazyky mohou vytvářet obsah i dynamický. Některé aplikace vyžadují JavaScript na straně klienta.

Tím se však neporušuje ani nepřekračuje koncepce MVC. Kromě HTML je vhodnou alternativou např. WML. Jelikož je pohled oddělen od modelu, může aplikace podporovat hned několik pohledů, každý pro jiný typ klienta, a přitom používat tytéž modelové komponenty.

4.2.3 Řadič

Řadič, neboli „controller“, reaguje na události (typicky pocházející od uživatele) a zajišťuje změny v modelu nebo v pohledu. Jelikož všechny požadavky klientů i odpovědi procházejí skrze řadič, nachází se zde centrální kontrolní bod celé aplikace. To je užitečné při zavádění nových funkcí. Řadič také pomáhá oddělit prezentační komponenty (pohledy) od vnitřních operací, a tím podporuje další vývoj.

Použitá literatura v této podkapitole: [2].

4.3 Diagramy

Diagramy jsou nejznámější a nejpoužívanější částí standardu. Následuje přehled diagramů jazyka UML 2.0 včetně jejich detailního popisu.

4.3.1 Sekvenční diagram (sequence diagram)

Sekvenční diagramy se vytvářejí většinou přímo z diagramů případů užití. K jednomu případu užití může existovat několik sekvenčních diagramů, které modelují interakci objektů v rámci komunikace účastníka se systémem.

Jak je docela zřejmé, sekvenční diagramy jsou zaměřeny výhradně na dynamickou stránku systému. Jsou vhodné pro nalezení jednotlivých objektů a zobrazení komunikace mezi nimi. S tvorbou sekvenčního diagramů se postupně vynořují jednotlivé objekty (resp. kandidáti na objekty), které jsou rovnou zapracovávány do diagramů tříd. Je vhodné zdůraznit, že objekty nalezené v počátečních fázích modelování systému se zřídkakdy v původní podobě uplatní také jako softwarové objekty implementované v cílovém prostředí. Zprvu se jedná spíše o koncepty, pojmy, které se postupem času, zejména ve fázi designu, vyhodnocují a dochází zpravidla k částečné redukci a konsolidaci objektových modelů.

Výsledný systém může mít nakonec podobu logické třívrstvé architektury, kdy jsou od sebe odděleny objekty prezentační vrstvy, objekty samotné problémové oblasti nalezené v analýze, rozpracované v designu a nakonec například objekty zajišťující funkce perzistentní vrstvy. Ignorujeme-li analýzu problémové oblasti, můžeme ve výsledném systému nakonec onu prostřední vrstvu představující tzv. obchodní logiku úplně postrádat. Jsme-li prozřetelnější, snažíme se obchodní logiku systému navrhovat přímo z analytických modelů, což nám mimo jiné pomůže zajistit lepší návaznost s uživatelskými požadavky.

S rozpracováváním sekvenčního diagramu může poměrně rychle vzrůstat jeho celková komplexnost, což je vcelku spolehlivým příznakem příliš obecného případu užití, ke kterému je sekvenční diagram vytvářen. Namísto tohoto prostředku je někdy možno zvolit prostředek jemu významově velice blízký – diagram spolupráce.

4.3.2 Diagram spolupráce (collaboration diagram)

Chceme-li v jednom diagramu znázornit jak strukturu objektů (například skládání objektů), tak i jejich dynamické chování, použijeme s výhodou diagramů spolupráce, které jsou vedle sekvenčních diagramů dalším prostředkem, jehož těžiště tkví v modelování dynamiky. Na rozdíl od sekvenčních diagramů je však znatelně obtížnější vysledovat návaznost jednotlivých posílaných zpráv zajišťujících samotnou funkcionalitu systému. Zatímco v sekvenčních diagramech je tato návaznost zřejmá z vertikálního uspořádání celého diagramu, v diagramech spolupráce je následnost zobrazena pořadovým číslem, kterým jsou zprávy uvozeny.

Tvorba diagramu spolupráce by měla být v souladu s diagramem tříd. Pro zachování vzájemné konzistence modelů je nezbytné dodržovat základní pravidla, která mezi těmito modely platí. Komunikace mezi objekty musí být například podmíněna existencí odpovídající vazby mezi jejich třídami, o existenci patřičných tříd nemluvě. Nemusíme asi ani zdůrazňovat, jak nevděčnou prací je „ruční“ kontrolování obdobných pravidel. Je jasné, že zde mohou opět pomoci CASE nástroje.

Stávají-li se sekvenční diagramy občas nepřehlednými, pak to u diagramů spolupráce platí dvojnásob. Na druhou stranu vzhledem k tomu, že se pořadí zasílání zpráv určuje pořadovým číslem, dá se lépe modelovat komplexní chování včetně větvení a cyklů.

4.3.3 Diagram tříd (class diagram)

Diagramy tříd patří bezesporu k nejčastěji používaným nástrojům UML a tvoří vůbec jakýsi základ všech prostředků objektové analýzy a designu. Tyto diagramy jsou většinou vytvářeny již ve fázi analýzy často jako pojmové modely, jejichž úkolem je formálněji definovat jednotlivé termíny používané ve studované problémové oblasti. Takové modely jsou maximálně přínosné a užitečné v okamžiku, kdy je nutno zpětně vstřebat terminologii oboru. S postupným přibližováním k fázi implementace jsou diagramy tříd poměrně zásadně přehodnocovány a v ideálním případě zpracovávány až do podoby grafického modelu ekvivalentního zdrojového kódu.

Diagramy tříd zobrazují statickou stránku systému, především vztahy mezi třídami. UML explicitně rozlišuje několik druhů tříd, stejně jako rozličné množství vztahů, které jednotlivé třídy navzájem pojí (asociace, agregace, kompozice, specializace/generalizace, atd.).

Tvorba diagramů tříd patří mezi klíčové problémy a je zřejmé, že zvládnout objektový přístup často znamená správně využívat právě tento typ diagramů používaný průběžně napříč celým životním

cyklem tvorby informačního systému. Zvláště při tvorbě těchto diagramů se doporučuje dodržovat nepsané pravidlo 7 + 2, které říká, že v jednom diagramu je vhodné zobrazit 7, maximálně až 9 tříd. Při respektování tohoto doporučení se většinou výraznou měrou zpřehlední výsledné modely, nicméně na druhou stranu vyvstává problém vhodného rozdělení systému na dílčí oblasti (v terminologii UML jsou jimi tzv. „packages“). Asi nejčastějšími chybami při tvorbě diagramů tříd (a celkově při objektovém modelování) je zaměňování pojmů objekt a třída, čímž může dojít k zavlečení poměrně závažných nekonzistencí do vytvářeného modelu.

Značně ovlivněna svými přímými předchůdci je notace diagramu tříd v UML vzdáleně podobná klasickým ER diagramům obohaceným o některé objektové rysy. Atributy tříd jsou zobrazeny ve střední části prvku třídy, metody pak v části spodní. Podle specifikace UML mohou být atributy pouze základních typů (integer, real, atd.), všechny ostatní by měly být zobrazeny jako vztahy k patřičným třídám. Je zde jasně vidět polovičatost s jakou se UML staví k objektovému přístupu. Ve skutečnosti samozřejmě neexistuje objektivní důvod pro odlišování jednotlivých atributů podle typu, v čistě objektových jazycích jsou i základní typy reprezentovány prostřednictvím patřičných tříd. Rozlišování způsobu notace atributů podle typu se může zezačátku zdát velice matoucí a nelogické. V podobném duchu je možné určit pro jednotlivé složky (atributy, metody) tříd jejich viditelnost vzhledem k ostatním třídám atd.

Značná orientace na jedno konkrétní prostředí je jednou z méně šťastných vlastností UML. Na druhou stranu je nutné si opět uvědomit, že tyto prostředky spadají až do pozdního designu a dříve než v designu o nich nemá smysl uvažovat.

Vzhledem k velkému množství modelovacích prvků použitelných v tomto typu diagramu může být poměrně lehké uchýlit se k tvorbě sice formálně přesných, ale těžko srozumitelných modelů, což se opět negativně projeví při snaze diskutovat řešení s člověkem, který se objektovým modelováním neživí.

4.3.4 Stavový diagram (state transition diagram)

Stavový diagram patří mezi klasické a osvědčené nástroje objektového modelování. V UML je tento prostředek přítomen v lehce modifikované podobě „Harelových“ diagramů. Diagram stavů a přechodů, jak je někdy tento prostředek nazýván, slouží pro modelování životního cyklu části systému svým rozsahem odpovídající jednomu objektu. Přechod mezi jednotlivými stavy bývá vyvolán podnětem z vnějšího okolí, nejčastěji ve formě zprávy zaslané příslušnému objektu nebo jinou externí událostí. Validní stav objektu je, zjednodušeně řečeno, definován přípustnými hodnotami jeho atributů, přechod mezi stavy je pak vlastně vyjádřením změny hodnot těchto atributů.

Životní cyklus objektu nějak začíná a zpravidla i nějak končí. Ovšem nemusí být již zcela samozřejmé, že tzv. „start“ stav by měl být v diagramu vždy pouze jeden, zatímco „stop“ stavů, tj.

stavů ukončujících životní cyklus sledovaného objektu může být více, pochopitelně většinou alespoň jeden.

Obdobně jako mnohé jiné notace umožňuje i UML definovat v těle stavu jednoduchou exekutivu vykonávanou v rámci uvažovaného stavu. V neposlední řadě je možné specifikovat podmínky, které musí platit při přechodu do stavu a při odchodu z něho. Tak je možné zachytit důležitá pravidla hlídající konzistenci a validitu celého modelu.

4.3.5 Diagramy aktivit (activity diagram)

Jako jistou obdobu stavových diagramů jsou chápány i diagramy aktivit, kde jednotlivými stavy rozumíme aktivity, a přechod mezi aktivitami je vyvolán dokončením aktivity stávající. Diagram aktivit se zpravidla vztahuje k jednomu případu užití, případně k jedné metodě objektu. Pomocí diagramu aktivit modelujeme tentokrát dynamický tok řízený nikoliv vnějšími událostmi, ale interními podněty. Při troše dobré vůle lze tento nástroj používat i pro modelování procesů a pracovních toků (work-flow), kdy se nezřídka využívá možnost modelovat paralelismus a větvení v rámci konkrétního zpracovávaného procesu.

Poměrně elegantním způsobem je možné přiřazovat k jednotlivým aktivitám osoby (resp. aktory), které jsou za provedení patřičné aktivity zodpovědné. Stejně dobře jako pro procesní modelování je možné používat diagramy aktivit i pro základní schematickou tvorbu grafického uživatelského prostředí, což může pomoci odhalit případná nedorozumění ohledně návaznosti jednotlivých obrazovek GUI.

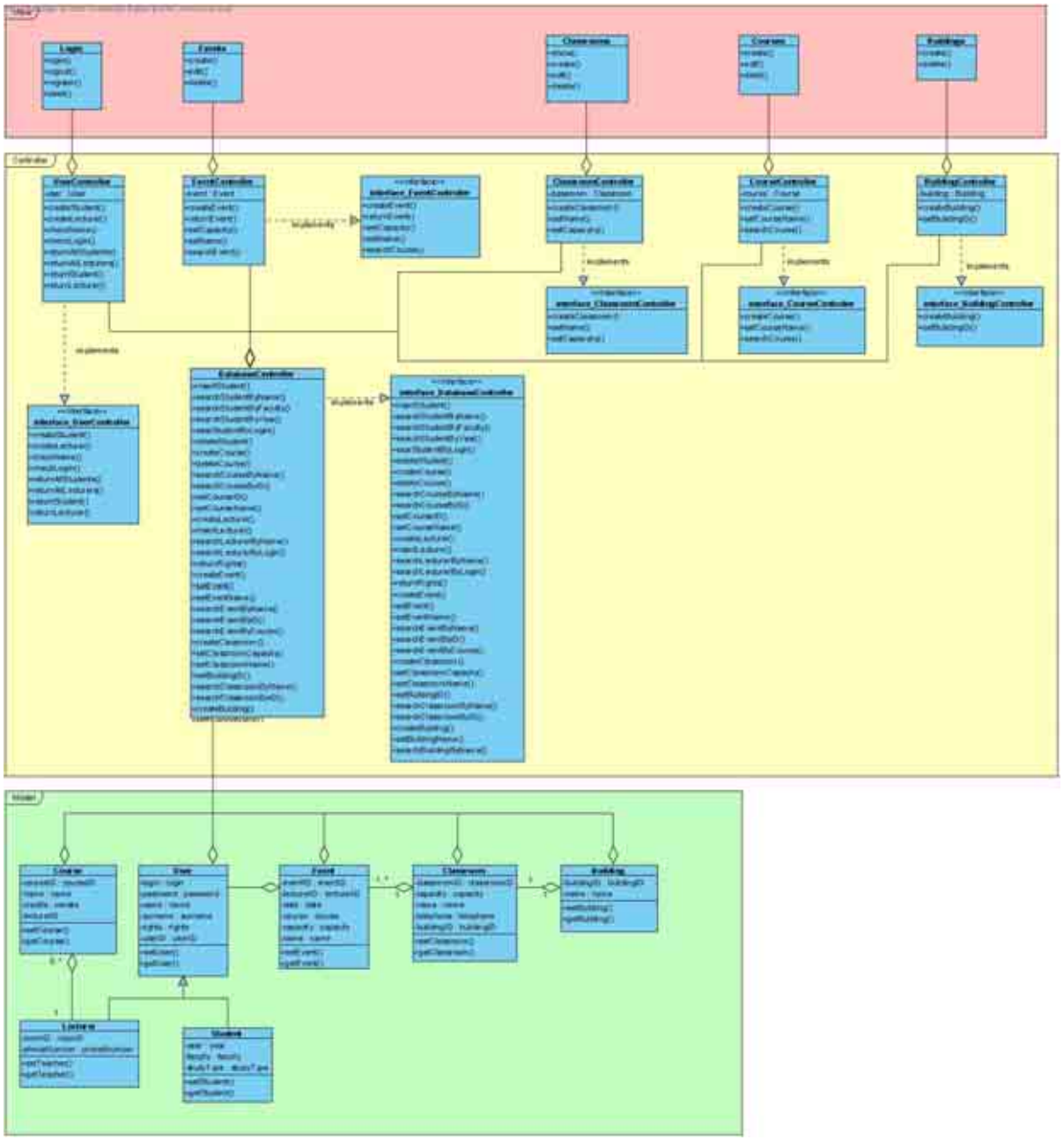
4.3.6 Diagram nasazení (deployment diagram)

Diagram nasazení je druhým typem diagramů určených pro implementační fáze. Jeho úkolem je především zobrazit vztahy mezi částmi systému tak, jak vypadají v době samotného vykonávání. Diagramy nasazení zobrazují rozložení jednotlivých SW komponent na HW zdrojích a jejich spolupráci, rozmístění HW a SW prostředků v lokalitách, topologie používaných sítí, druhy a využití komunikačních prostředků atp.

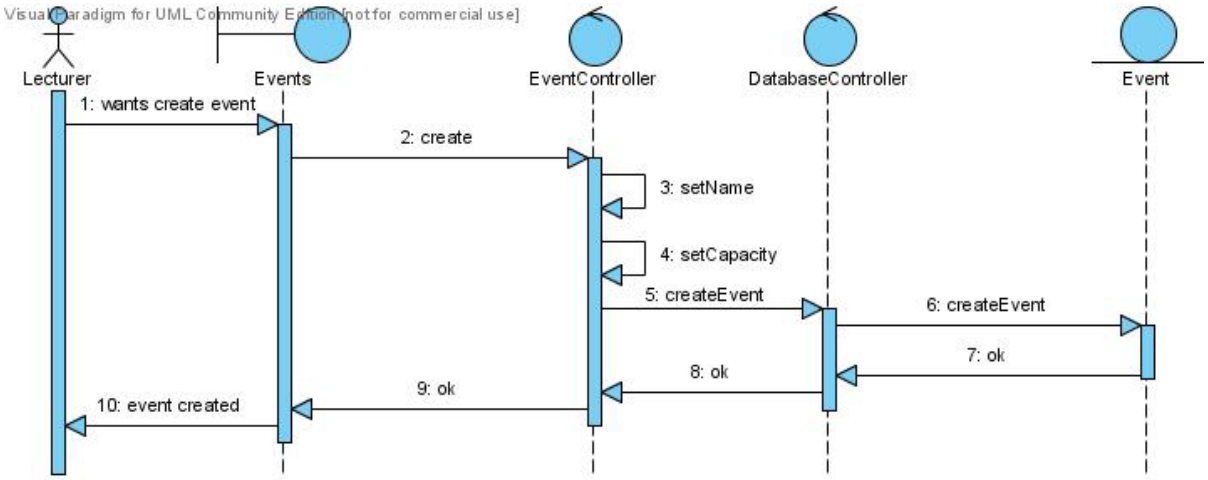
4.3.7 Vlastní návrh diagramů

Po důkladném studiu všech zmíněných diagramů v předchozích podkapitolách byly zvoleny a následně vytvořeny následující diagramy.

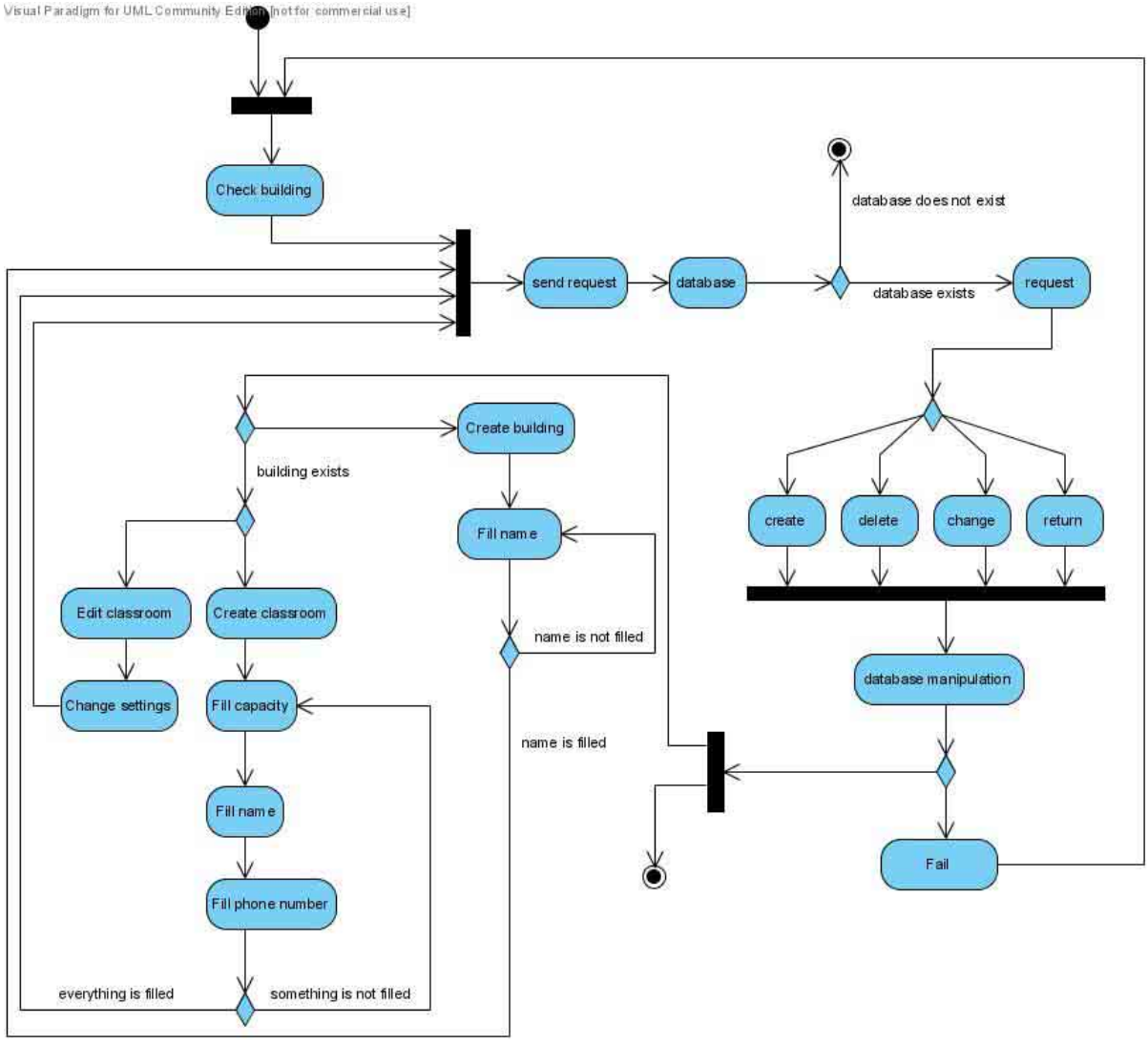
Diagram tříd na obrázku (3.1) jasně naznačuje zvolenou třívrstvovou architekturu a nechybí v něm ani názvy procedur a funkcí. Sekvenční diagram na obrázku (3.2) znázorňuje akci, kdy učitel přidává do systému novou událost. Na obrázku (3.3) je diagram aktivit zobrazující manipulaci s budovami a místnostmi. Obrázek (3.4) ukazuje nasazení systému do provozu.



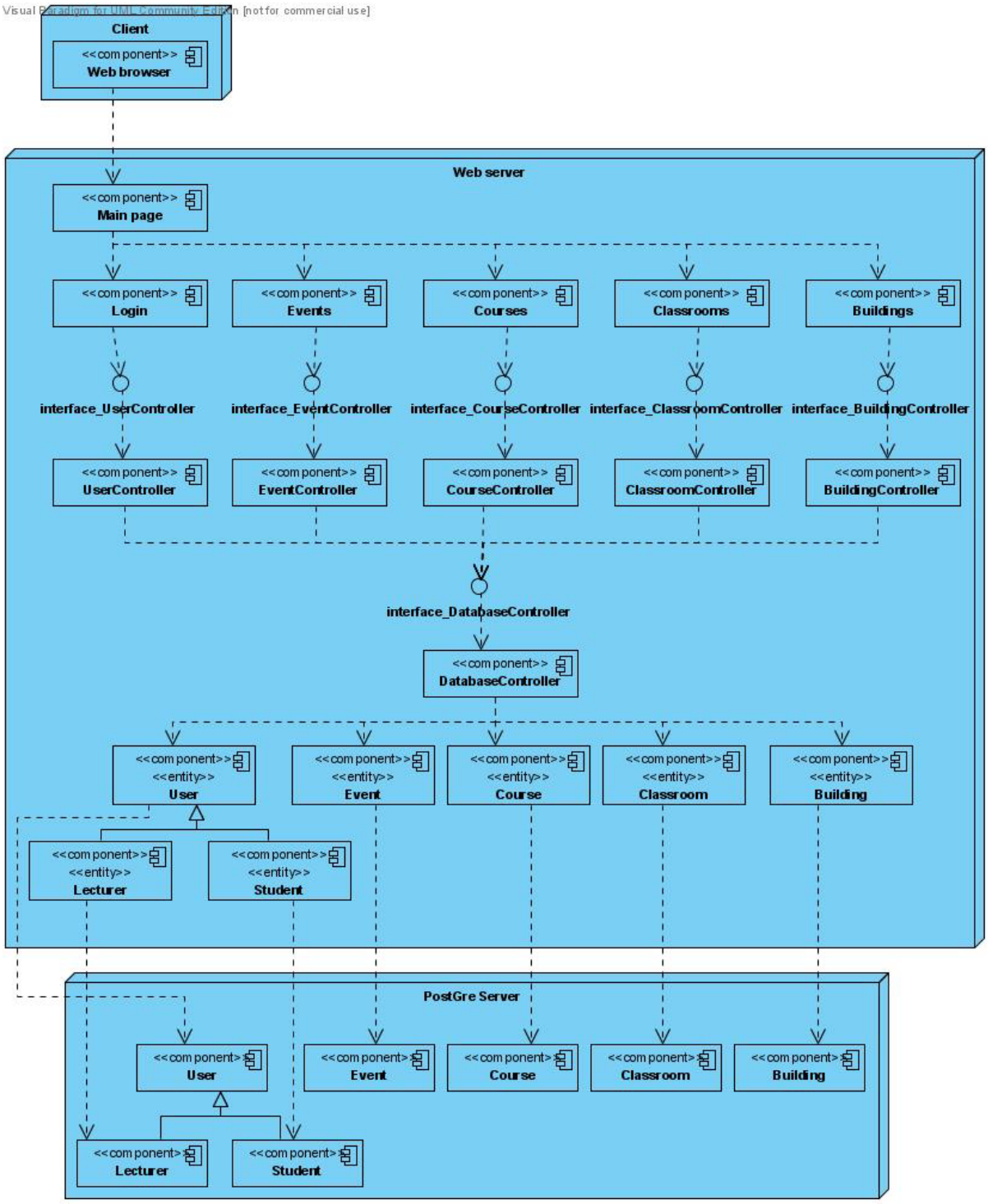
Obrázek 4.1: Diagram tříd



Obrázek 4.2: Sekvenční diagram



Obrázek 4.3: Diagram aktivit



Obrázek 4.4: Diagram nasazení

5 Implementace systému

Implementace znamená především programování. Neznamená ale pouhé kódování návrhu do příkazů programovacího jazyka. Návrh algoritmů není vždy součástí návrhu systému. Řadu algoritmů ve skutečnosti navrhuje až programátor v rámci psaní programu. Podobně zápis složitých dotazů nad databází v jazyce SQL může být intelektuálně náročný. Tím se vlastně programátor podílí i na návrhu. Moderní programování spočívá nejen v zápisu programu v daném programovacím jazyce, ale je založeno na využití existujících či dostupných komponent. To vyžaduje od programátora nejen znalost programovacího jazyka, ale i znalost či schopnost použitelné komponenty nalézt, pokud existují.

5.1 Jazyk PHP

PHP je serverový skriptovací jazyk (server-side) navržený pro potřeby webových stránek. To znamená, že vše, co PHP provádí, neprobíhá na straně klienta jako například u Javascriptu, ale interpretuje se na straně serveru a generuje HTML (či jiný) výstup, který vidí koncový uživatel. PHP je „Open Source“, tedy volně šiřitelná technologie. PHP není závislé na platformě a není vázané s žádným konkrétním serverem, může tedy běžet téměř kdekoliv.

5.1.1 Historie PHP

Počátky vývoje PHP sahají do roku 1994, kdy Rasmus Lerdorf vytvořil jednoduchý systém evidence přístupů ke svému webu, nejdříve v jazyce Perl, poté v C. Ten se rozšířil mezi další uživatele, kteří přicházeli s požadavky na vylepšení. Vznikl tak systém Personal Home Page Tools, později Personal Home Page Construction Kit.

Rasmus Lerdorf vytvořil i nástroj umožňující začleňování SQL dotazů a tím zpřístupnění databází na serveru – Forms Interpreter (FI). V roce 1997 bylo PHP/FI 2.0 oficiálně uvolněno, brzy potom vznikla verze PHP 3.0.

5.1.1.1 PHP 3

PHP 3.0 vytvořil Andi Gutmans a Zeev Suraski. Původní zkratka dostává nový význam – PHP: Hypertext Preprocessor.

5.1.1.2 PHP 4

PHP 4.0, oficiálně uvolněné v roce 2000 je výkonnější verzí PHP 3, přidává například podporu pro mnoho WWW serverů, HTTP sessions, buffering výstupu, bezpečnější způsoby zpracování vstupů uživatele a nové jazykové konstrukty.

5.1.1.3 PHP 5

V červnu 2003 byla oficiálně uvolněna betaverze PHP 5. Největší změna je v objektovém modelu, PHP se přibližuje ostatním jazykům podporujícím OOP. Snazší je také obsluha chyb.

5.1.2 Další aspekty

Je zajímavé si povšimnout odlišností od skriptů psaných v jiných jazycích, jako Perl nebo C – místo psaní programu s množstvím příkazů pro výstup HTML vytvoříte HTML soubor s vloženým kódem, který něco vykoná. PHP kód je uzavřen mezi zvláštními uvozujícími a koncovými „tagy“, které umožňují vstoupit a opustit „PHP mód“.

PHP se liší od jazyků, jako je Javascript na straně klienta, tím, že je vykonáván na straně serveru. Je dokonce možné nakonfigurovat webový server tak, aby zpracovával veškeré HTML soubory.

Tento jazyk je extrémně jednoduchý pro začátečníka, ale poskytuje mnoho pokročilých vlastností i profesionálnímu programátorovi. Ačkoli je programování v PHP zaměřeno na skripty na straně serveru, můžete v něm vytvořit mnohem více.

Použitá literatura v této podkapitole: [15], [16] a [17].

5.2 Databáze PostgreSQL

Databáze a databázové servery patří na třetí místo v prodejnosti software za hry a kancelářské aplikace. Vývoj jednoznačně směřuje k jejich potřebě v jakékoliv formě, ať už té nejjednodušší souborové (například účetnictví), nebo sofistikovanější SQL serveru (např. webové aplikace).

5.2.1 Historie databází

Databázi můžeme přirovnat ke kartotéce, kde každý záznam je jedinečný a organizovaně nalezitelný. V dobách, kdy se počítače začaly šířit masívněji, vládla poměrně anarchie, protože takřka každý výrobce software, měl vlastní, z jeho hlediska nejlepší, systém, jak záznamy ukládat. Třeba použitím jednoho oddělovače pro věty a druhého pro položky ve větách.

V 60. letech dvacátého století firma IBM vytvořila systém hierarchického modelu uspořádání dat. Data byla uspořádána ve stromech. Později se začaly propojovat data mezi stromy, ale složitost tohoto uspořádání dat neumožňovala vystihnout všechny možné vazby, které byly třeba. Proto se v 70. letech začali objevovat systémy, které byly založeny na relační algebře. Od 80. let se rozvíjí naplno relační databáze a jazyk SQL.

5.2.2 Jazyk SQL

SQL se překládá jako Structured Query Language, některé mutace hovoří i o Simple Query Language. Hlavní síla nezávisí na tomto jazyce, ale v celkové organizaci dat a administraci serveru.

5.2.3 Historie PostgreSQL

Původně byl server vyvíjen v Berkeley na kalifornské univerzitě v letech 1986 až 1993. Vývoj se rozpadl komerčním směrem (sloučení s Informixem) i nekomerčním směrem, jako Postgres95. V roce 1996 je projekt přejmenován na PostgreSQL, je uvedena verze 6.0. V současnosti je k dispozici nejnovější verze s pořadovým číslem 8.

5.2.4 Aspekty PostgreSQL

Většina tvůrců databázových a informačních systémů si velmi oblíbila spojení PHP a MySQL. Pro tento projekt však byl zvolen databázový server PostgreSQL, a hned ze dvou důvodů. Prvním je fakt, že tento server nabízí spoustu zajímavých funkcí a vylepšení a druhý důvod je potom velmi prostý – možnost nějak se odlišit od ostatních tvůrců a přinést tak nějaké zpestření do tohoto projektu.

Jak již bylo zmíněno, PostgreSQL vyniká hned v několika aspektech, např. stabilitou, rychlostí, dobrou podporou a dobrou integrací pokročilých technologií. Některé web hostingy jej nabízejí vedle MySQL, potažmo i aplikace ve finančním světě, které se o něj opírají. Primárně je určen pro použití v UNIX systémech, donedávna existovala jen alfa verze pro Windows. PostgreSQL je relační databáze podporující SQL92 a SQL99 normu a mnoho dalších moderních rysů: komplexní dotazy, cizí klíče (foreign keys), spouště (triggery), pohledy (views), transakce nebo vlastní datové typy. Server je vysoce rozšiřitelný (právě možnostmi definovat si vlastní datové typy a uložené procedury) a objektově-relační. Klíčovou vlastností je také výborná podpora na straně jazyka PHP.

Použitá literatura v této podkapitole: [18] a [19].

5.3 XHTML

První definici HTML vytvořil Tim Berners-Lee v roce 1991. Tato verze umožňovala vkládat do textu obrázky, hypertextové odkazy, vytvořit několik logických úrovní a několik druhů zvýraznění. Byla označena jako HTML 0.9. Požadavky uživatelů se zvyšovaly a producenti prohlížečů začali vytvářet nové HTML prvky. V roce 1997 byla do světa vypuštěna HTML 4.0 a o dva roky později opravná verze HTML 4.01, která je také poslední vývojovou verzí HTML.

HTML je tedy jazyk pro publikování hypertextu na WWW. Je to nepatentovaný formát založený na SGML. HTML využívá „tagy“ ke strukturování textu do nadpisů, odstavců atd. Vývoj HTML sice již skončil, ale HTML má samozřejmě své následníky.

5.3.1 Aspekty XHTML

XHTML je nástupce HTML založený na XML. Rozlišujeme 3 druhy XHTML: XHTML 1.0 Strict (čistě strukturální značkování, neobsahuje žádné značky spojené s formátováním vzhledu), XHTML 1.0 Transitional (povoluje atributy pro formátování textu a odkazů v elementu body a některé další atributy) a konečně XHTML 1.0 Frameset (používá se při použití rámců pro rozdělení okna prohlížeče na dvě nebo více částí).

Použitá literatura v této podkapitole: [20].

5.4 CSS

CSS (Cascading Style Sheets) neboli kaskádové styly vznikly jako souhrn metod pro úpravu vzhledu stránek. První návrh normy byl zveřejněn v roce 1994, v roce 1996 byla pak vydána specifikace CSS1, v roce 1998 CSS2, nyní se pracuje na verzi CSS3.

5.4.1 Aspekty CSS

CSS se využívá k formátování obsahu HTML, XHTML a XML dokumentů. Ve srovnání s formátováním pomocí atributů v HTML formátovací schopnosti rozšiřuje. Styly umožňují přesně určit, jak bude který element vypadat. Na rozdíl od atributů stylem můžeme definovat jednotný vzhled elementu pro celý dokument a to jediným zápisem pro příslušný element. Stejně tak můžeme pomocí stylu určit odlišné formátování pro jen jediný výskyt určitého elementu. Tím se jednak zbavíme velkého množství kódu, jednak se tento kód stane mnohem přehlednější. Navíc pokud se jednou rozhodneme změnit například barvu písma všech odstavců, bude to pro nás otázka několika málo vteřin, měnit každý atribut u každého elementu v HTML by byla katastrofa. Jeden styl můžeme snadno použít pro libovolné množství stránek.

Styl se skládá z pravidel pro jednotlivé elementy, které mají být formátovány. Každé takové pravidlo má dvě části, selektor (název elementu, pro který má toto pravidlo platit) a deklaraci (co pro něj má platit). V deklaraci určujeme vlastnost a její hodnotu, deklarace je uzavřena do složených závorek.

5.4.2 Dědičnost

Většina vlastností se dědí. To znamená, že element, který nemá vlastnost definovanou, ji dědí po nadřazeném elementu. Týká se to především vlastností písma – barvy, velikosti, stylu atd. Pokud tedy chceme definovat nějakou vlastnost, kterou budou mít všechny elementy společnou, definujeme ji pro element „body“.

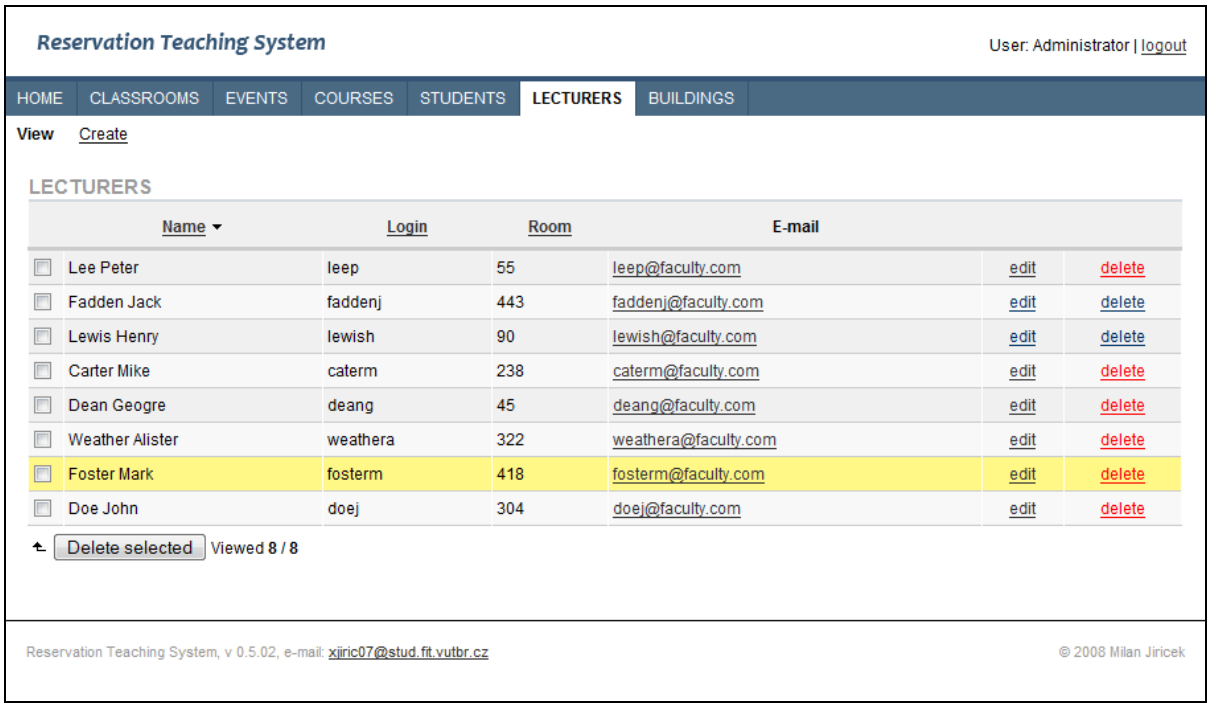
5.4.3 Váha stylů

Pokud ve stylu definujeme pro stejný element stejnou vlastnost dvakrát, vyšší váhu má ta deklarace, která byla definovaná později (myšleno na pozdějším řádku) a ta se také provede. Pokud bychom chtěli některé deklaraci přiřadit větší důležitost, použijeme klíčové slovo „!important“.

Použitá literatura v této podkapitole: [21] a [22].

6 Prezentace systému

Jako vhodný vzhled systému byla zvolena nenásilná světlá barva v kombinaci s modrými prvky. Menu je horizontální, což v dnešní době, kdy se stále více upřednostňují širokoúhlé monitory, jistě správná volba. Hlavní nabídku je možno spatřit na obrázku (6.1), kde je také zobrazena tabulka s výpisem učitelů, kteří jsou uloženi v systému. Tento styl je použit u všech takovýchto přehledových tabulek. HTML a CSS kód byl tvořen podle pravidel přístupnosti, je tedy plně přizpůsoben všem standardům a aspektům spojených s přístupnými web stránkami.

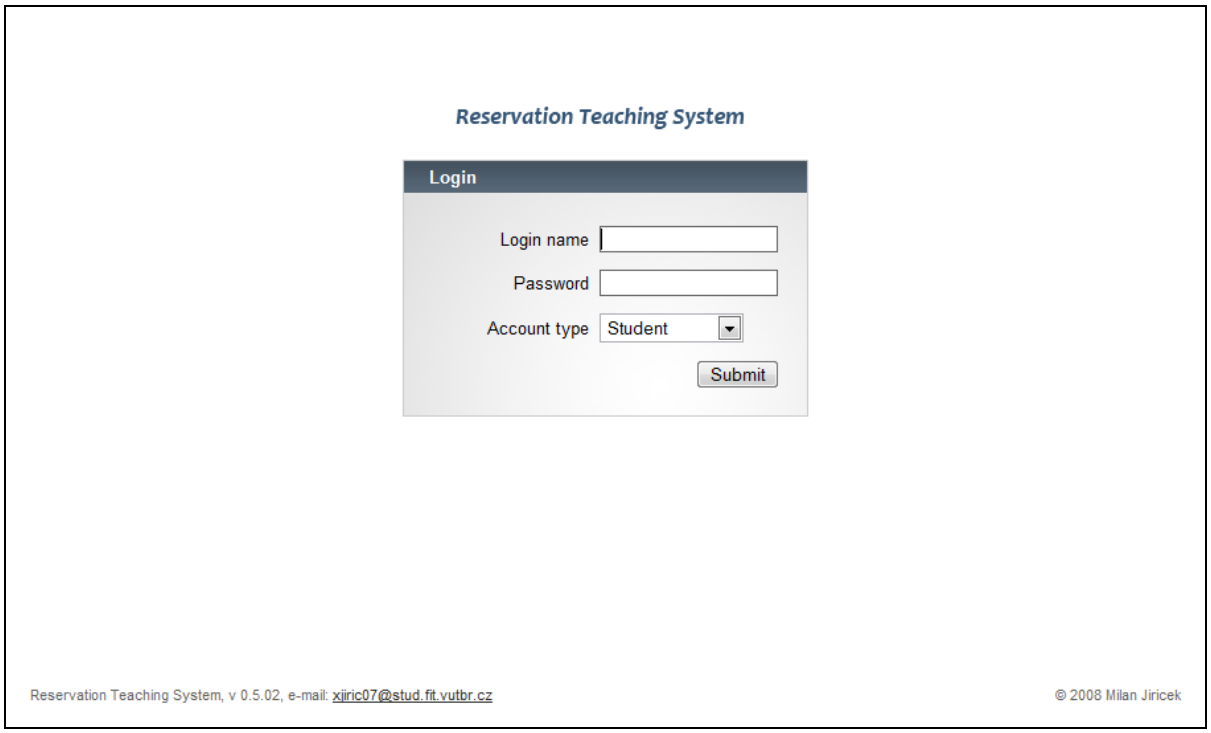


Obrázek 6.1: Sekce výpis učitelů

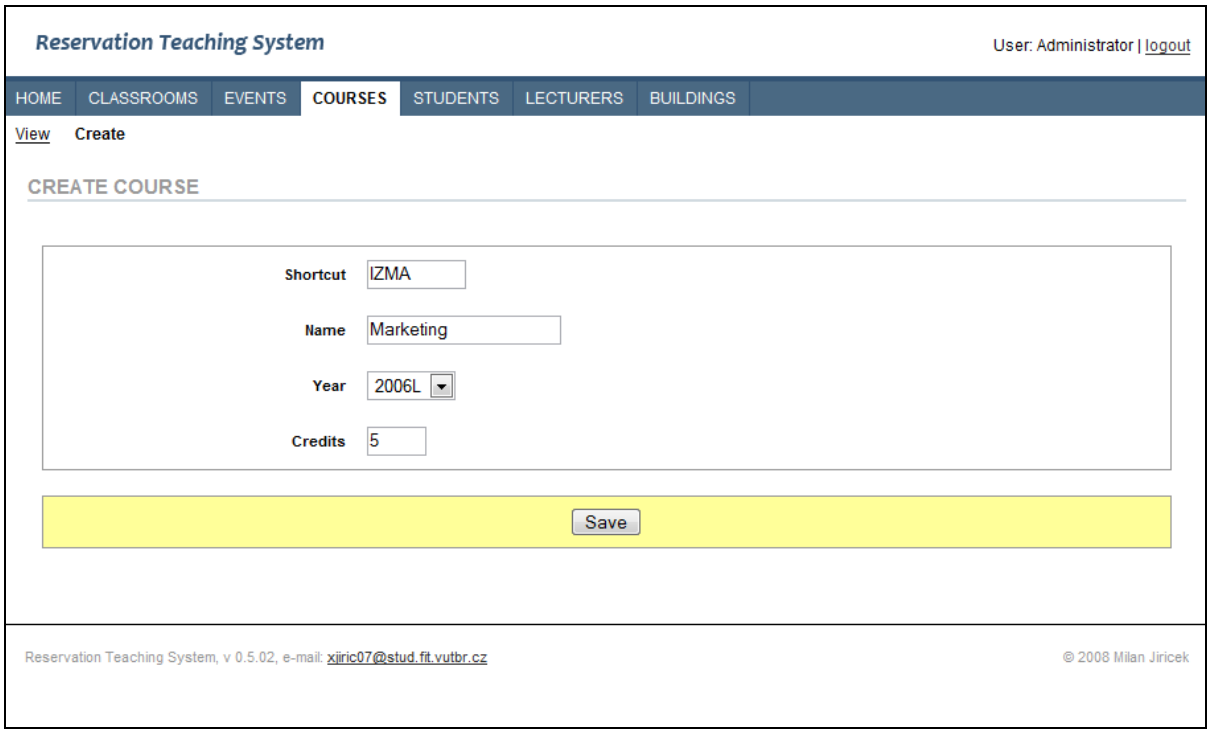
V podobném nenáročném duchu je vytvořena také přihlašovací obrazovka. Ta je zobrazena na obrázku (6.2). Společná je jak pro studenty a učitele, tak i pro administrátory. Ti si pro přihlášení do příslušné sekce vybírají z nabídky různých typů účtů. Na obrázku (6.3) lze spatřit formulář, pomocí kterého administrátor vytváří nový kurz. Jako poslední ukázka (6.4) je zde prezentován kalendář akcí, pomocí kterého si učitel nebo administrátor vybere vhodný termín pro danou událost.

Na obrázku (6.5) je nakonec ukázka kódu pro přihlášení studentů a učitelů. Hesla pro tyto uživatele jsou v databázi kvůli bezpečnosti uložena pomocí šifrovacího algoritmu sha1.

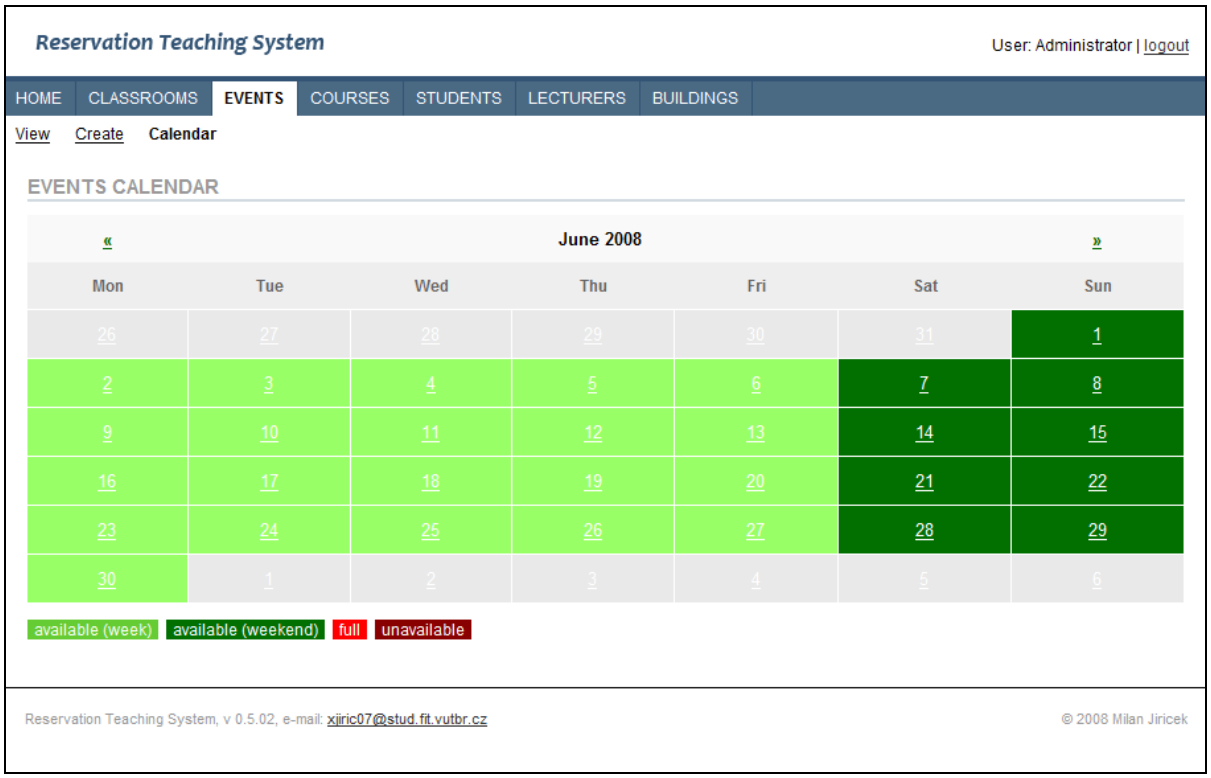
Implementace nebyla dokončena v plném rozsahu. Je to způsobeno tím, že takový systém je velmi rozsáhlý, a na jeho vývoj je potřeba daleko více časových prostředků, než bylo v konečném účtování k dispozici. Více se o tomto problému lze dozvědět v závěrečné kapitole.



Obrázek 6.2: Přihlašovací obrazovka



Obrázek 6.3: Vytvoření nového kurzu



Obrázek 6.4: Kalendář akcí

```
case '1': //student
{
 $result=$db->get_result("*", "students", "login='".$_POST['login']."' AND password='".sha1($_POST['passwd'])."'");
 $row=pg_fetch_assoc($result);

 if(pg_num_rows($result)==1)
 {
  $_SESSION['login']=$row['login'];
  $_SESSION['user']=$row['name']." ".$row['surname'];
  $_SESSION['rights']='S';
 }
 else
  $fault=true;

 break;
}
case '2': //lecturer
{
 $result=$db->get_result("*", "lecturers", "login='".$_POST['login']."' AND password='".sha1($_POST['passwd'])."'");
 $row=pg_fetch_assoc($result);

 if(pg_num_rows($result)==1)
 {
  $_SESSION['login']=$row['login'];
  $_SESSION['user']=$row['name']." ".$row['surname'];
  $_SESSION['rights']='L';
 }
 else
  $fault=true;

 break;
}
```

Obrázek 6.5: Zdrojový kód pro přihlašování uživatelů

Použitá literatura v této podkapitole: [23].

7 Závěr

V poslední kapitole bude uvedeno shrnutí zjištěných aspektů, dále pak srovnání rezervačního systému pro výuku s podobnými informačními systémy a konečně možnosti rozšíření tohoto systému. Mezi ně se dají zařadit i některé neimplementované moduly systému z důvodu jejich časové náročnosti, jako například propracovaná kontrola uživatelských práv, vytváření sofistikovaných rozvrhů na základě určitých podmínek nebo automatické přihlašovaní různých skupin studentů na jednotlivé přednášky, popř. zkoušky a jiné události.

7.1 Životní cyklus informačního systému

Ze všeho nejdříve byly probrány vhodné přístupy objektově orientovaného návrhu pro informační systémy, detailně pak návrhové vzory, softwarová architektura, ad. Následně provedená případová studie rezervačního systému pro potřeby výuky a analýza požadavků nastínila, jakým směrem je dobré se ubírat při tvorbě software.

Prostřednictvím modelovacího jazyka UML byl následně proveden detailní objektový návrh tohoto systému s třívrstvou architekturou MVC. Ta byla společně s návrhovými digramy postupně probrána a některé z navržených diagramů vyobrazeny v konkrétních případech a situacích. K jejich tvorbě byl použit program Visual Paradigm ve verzi Community Edition. Zvolen byl na základě konzultace a následného doporučení od vedoucího této práce. Jak je z diagramů vidět, volba to byla jistě správná.

Po důkladné analýze a návrhu přišla na řadu implementace rezervačního systému za použití moderních programovacích technik a nástrojů, které byly postupně do detailu popsány a představeny. Vysvětlen byl důvod zvolení jednotlivých technologií, mimo jiné i poměrně netradiční databázový server PostgreSQL.

V této kapitole jsou na jejím konci probrány možnosti dalšího rozšíření systému, což také neodmyslitelně patří k životnímu cyklu jakéhokoliv software.

7.2 Porovnání s jinými systémy

Pro srovnání se v této části přímo nabízí informační systém fakulty informačních technologií, tedy systém, do kterého se přihlašujeme my, studenti, ale také jednotlivý učitelé, popř. administrátoři. Fakultní systém je poměrně hodně propracovaný, takže je potřeba toto srovnání brát s rezervou.

IS FIT je naprogramovaný v jazyce PHP v kombinaci s databázovým serverem MySQL. To je „zaručená“ kombinace, na které lze je těžko něco zkazit. Prostředí IS FIT je taktéž zkrášleno pomocí kaskádových stylů, avšak místo XHTML standardu je zde použit HTML ve verzi 4.0. Ve

funkčnosti již fakultní systém jasně vede, jeho rozsáhlosti se zde prezentovaný rezervační systém těžko může přiblížit.

7.3 Možnosti dalšího rozšíření

Jako zajímavé rozšíření tohoto systému se jeví přidání registrace studentů přímo na jednotlivé kurzy, nikoliv pouze na pravidelnou či nepravidelnou výuku. Mezi další se řadí např. možnost tvorby vlastního rozvrhu podle zapsaných událostí, diskusní fórum ke každému předmětu, popř. akci, nebo sklad dokumentů k jednotlivým kurzům.

Možnosti postupného rozšiřování jsou prakticky nekonečné a záleží pouze na tom, kolik času vývojář chce nebo může investovat do takového projektu.

Literatura

[1] Gamma, Helm, Johnson, Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[2] Cavaness Chuck. *Programujeme Jakarta Struts*. Grada, 2003.

[3] Šveřepa Jaromír. *MDA, aneb architektura řízená modelem*. Softwarové noviny, 10/2004.

[4] WWW stránky. *Objektová analýza, návrh a programování*. http://www.objekty.vse.cz.

[5] WWW stránky. *Úvod do objektového modelování a jazyka UML*. http://www.komix.cz/Tisk/Clanky/Historie/Uvod_UML.aspx.

[6] Arlow Jim. *UML a unifikovaný proces vývoje aplikací*. Computer Press, 2003.

[7] WWW stránky. *Základy OOP*. http://php.interval.cz/clanky/zaklady-oop/.

[8] Sodomka Petr. *Informační systémy v podnikové praxi*. Computer Press, 2006.

[9] McLaughlin, B. et al. *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*. O'Reilly Media, 2006.

[10] Arlow Jim. *UML a unifikovaný proces vývoje aplikací*. Computer Press, 2003.

[11] WWW stránky. *Základy objektově orientovaného návrhu*. http://is.mendelu.cz/eknihovna/opory/index.pl?cast=5855.

[12] WWW stránky. *Doporučené postupy v programování*. http://dsrg.mff.cuni.cz/teaching/nprg043/lecture07.html.

[13] Zendulka Jaroslav a kol. *Analýza a návrh informačních systémů*. 2006.

[14] Freeman, E. et al. *Head First Design Patterns*. O'Reilly Media, 2004.

[15] Gutmans, Bakken, Rethans. *Mistrovství v PHP 5*. Computer Press, 2007.

[16] WWW stránky. *Úvod do PHP*. http://www.webtvorba.cz/php/uvod-do-php.html.

[17] WWW stránky. *Manuál PHP*. http://cz.php.net/manual/cs/.

[18] WWW stránky. *Seriál PostgreSQL*. http://www.linuxsoft.cz/article.php?id_article=304.

[19] Momjian Bruce. *PostgreSQL*. Computer Press, 2003.

[20] WWW stránky. *Úvod do XHTML*. http://www.webtvorba.cz/xhtml/uvod-do-xhtml.html.

[21] Staníček Petr. *CSS Kaskádové styly*. Computer Press, 2003.

[22] WWW stránky. *Úvod do CSS*. http://www.webtvorba.cz/css/uvod-do-css.html.

[23] WWW stránky. *Pravidla tvorby přístupného webu*. http://www.pravidla-pristupnosti.cz/.

Seznam příloh

Příloha 1. CD se zdrojovými kódy.