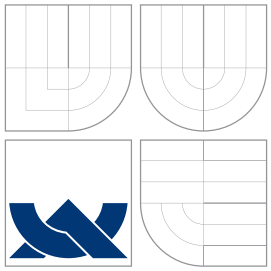
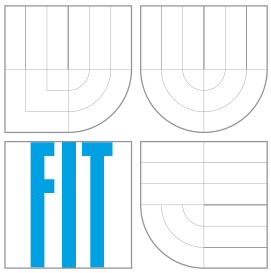


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

BALÍČEK MODULŮ PRO VÝVOJ V JAZYCE PHP

PACKAGE OF MODULES FOR DEVELOPMENT WITH PHP

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — ALEŠ RYBÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA
SUPERVISOR

BRNO 2007

TODO: Zde bude umístěno -------+++ Zadání +++------

TODO: Zde bude umístěno -------+++ Licenční ujednání +++------

Abstrakt

Balíček modulů pro tvorbu webových aplikací pomocí PHP je projekt zaměřený na usnadnění a zvýšení efektivnosti práce programátora při vývoji webových aplikací s využitím skriptovacího jazyka PHP v návaznosti na projekt „Vývopjový systém webových aplikací“. Cílem projektu je úprava stávajícího vývojového systému pro jednoduchou návaznost modulů na něj a následná implementace modulů, které programátorům umožní systém využívat.

Klíčová slova

modul, PHP, objektově orientované programování, framework, uchování stavu, událost, zpětná volání, MVC, Vývojový systém webových aplikací

Abstract

Package of modules for development with PHP is project aimed on simplification and efficiency of programmer's work when developing web based applications in scripting language PHP. This project is continuation of PHP Modular Object framework and wants to extend its connectivity to new modules. Next goal is implementation of modules which allow programmers to use the development system.

Keywords

module, PHP, object oriented programming, framework, state saving, event, callback, MVC, PHP Modular Object framework

Citace

Aleš Rybák: Balíček modulů pro vývoj v jazyce PHP, diplomová práce, Brno, FIT VUT v Brně, 2007

Balíček modulů pro vývoj v jazyce PHP

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Aleše Smrčky a uvedl jsem všechny publikace a literární prameny, z nichž jsem při řešení čerpal.

.....

Aleš Rybák

21. května 2007

Obsah

Kapitola 1

Úvod

Balíček modulů pro tvorbu webových aplikací pomocí PHP je projekt, jehož hlavním cílem je zpříjemnit a zefektivnit práci programátora při vývoji webových aplikací s využitím jazyka PHP v návaznosti na projekt Vývojový systém webových aplikací Miroslava Oprštěného [1]. Jedná se o projekt, jehož cílem je vyvinout jádro systému, jenž přibližuje práci se serverovým software desktopovým aplikacím. Aby tento systém byl použitelný v praxi, je zapotřebí toto jádro rozšířit o prvky, které práci s ním nejen umožní, ale také zjednoduší a zefektivní. Balíček modulů, který bude v rámci této práce vyvíjen bude zahrnovat i některé z těchto prvků.

V ideálním případě by výsledkem projektu měl být produkt poskytující prostředí, v němž bude moci vyvíjet i méně zkušený programátor. Dále by pak mohlo být možné pro tento systém vytvořit i aplikace pro návrh grafického uživatelského rozhraní. Pokud bude systém kompletní a funkční, je možné, že jej programátorská komunita, která je kolem jazyka velmi početná, bude chtít používat a dále rozvíjet.

Celý text je rozdělen do pěti kapitol. Po tomto úvodu následuje nastínění současné situace na poli webových aplikací a jejich vývoje pomocí programovacího jazyka PHP a do jejího kontextu je pak zasazen projekt Vývojový systém webových aplikací, na nějž tato práce navazuje. Třetí kapitola specifikuje požadavky kladené na projekt, čtvrtá se zabývá návrhem všech změn a rozšíření vývojového systému a nakonec také návrhem ukázkové aplikace, na níž budou demonstrovány vybrané funkce systému. Pátá kapitola popisuje implementaci navržených komponent, vzniklé potíže a změny provedené oproti návrhu. V závěrečné kapitole jsou shrnuty dosažené výsledky a diskutováno možné pokračování projektu.

Kapitola 2

Současná situace

V této kapitole bude stručně popsána situace v oblasti webových aplikací a jejich vývoj pomocí jazyka PHP. Do tohoto kontextu bude poté zařazen projekt „Vývojový systém webových aplikací“, na nějž tato práce navazuje.

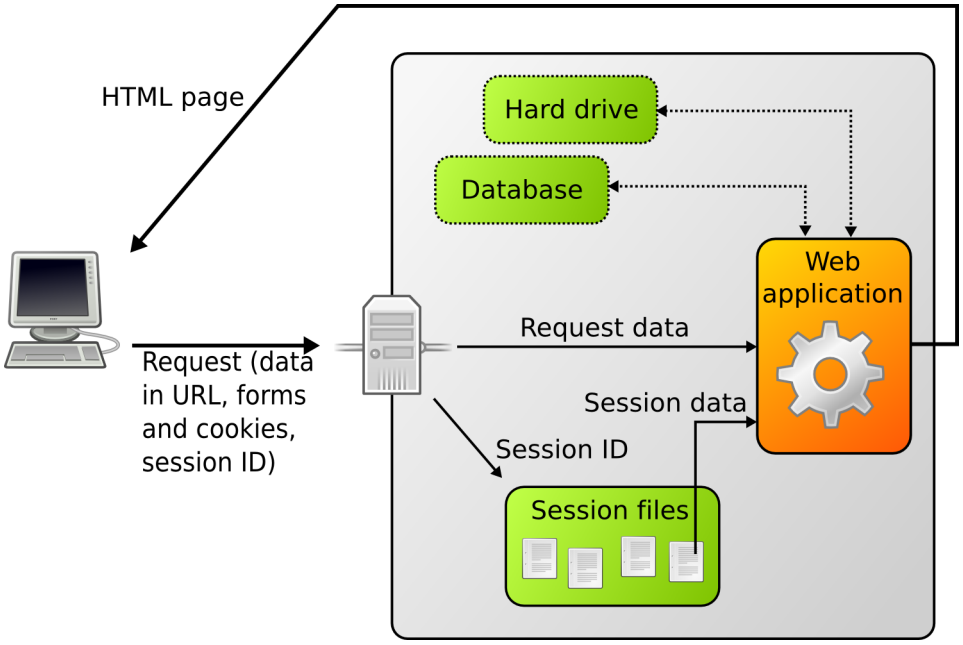
2.1 Vývoj webových aplikací pomocí jazyka PHP

Když byl veřejnosti umožněn přístup k Internetu, byly nejčastějším obsahem statické stránky propojené navzájem pomocí odkazů. Technologický vývoj ale umožnil i vývoj stránek, na nichž se začaly objevovat dynamické prvky (např. počítadla přístupů). Následovaly stránky, které byly celé generovány dynamicky. Takové stránky jsou pro uživatele, který vyžaduje interaktivní, jednoduchý a rychlý přístup k informacím, velkým přínosem. Také pro administrátory mají své přínosy, protože na rozdíl od statických stránek jsou mnohem lépe spravovatelné.

Dalším krokem po dynamicky generovaných stránkách jsou webové aplikace. Jedná se vlastně o složitější dynamické stránky poskytující funkčnost, kterou až do nedávné doby zastávaly pouze desktopové aplikace. V současnosti zažívají tyto aplikace velký rozmach a rychlý rozvoj. Jejich oblíbenost vychází z několika aspektů, které jsou z větší části důsledkem toho, že k jejich používání postačí uživateli internetový prohlížeč. Prvním z nich je dobrá dostupnost, kdy uživatel může k aplikaci přistupovat prakticky z kteréhokoliv počítače, na němž je internetový prohlížeč nainstalován. Dalším je možnost používat aplikaci bez nutnosti její předešlé instalace, což je bezesporu výhodné zejména pro laické uživatele. A třetí je platformní nezávislost, kterou ocení zejména pokročilí uživatelé pracující na více platformách nebo uživatelé mobilních zařízení.

Jazyk PHP vznikl původně jako jednoduchý skriptovací jazyk určený primárně právě pro vkládání jednoduchých dynamických prvků do internetových stránek. S vývojem Internetu a požadavků jeho uživatelů se vyvíjel i tento jazyk a stal se velmi oblíbeným prostředkem k dynamickému generování internetových stránek.

Internetová stránka je generována na základě požadavku od uživatele (viz obrázek 2.1).



Obrázek 2.1: Zpracování požadavku webovou aplikací

V rámci požadavku může klient serveru několika způsoby předat informace — jedná se o informace v URL, data z formulářů či tzv. cookies. Všechny tyto metody předání informací mají své výhody i nevýhody, z nichž vyplývá jejich typické využití. Server požadavek po přijetí předá skriptu jazyka PHP, jež v závislosti na datech v něm generuje internetovou stránku, která je vrácena uživateli.

Jazyk PHP umožňuje vývoj širokého spektra aplikací — počínaje jednoduchým počítadlem přístupů, přes dynamické stránky, podnikové informační systémy až k webovým aplikacím. Možnosti PHP ještě výrazně rozšiřuje množství nejrůznějších knihoven, rozšíření, integrovaných funkcí a v neposlední řadě také možnost spolupráce s mnoha databázovými systémy (např. MySQL, PostgreSQL, ale i dalšími). Oblíbenost jazyka je dále ještě umocněna faktem, že se jedná o open–source projekt s dobrou dokumentací a širokou základnou přispěvatelů. Další výhodou je, že je součástí většiny linuxových distribucí a je tak často k dispozici u poskytovatelů hostingových služeb.

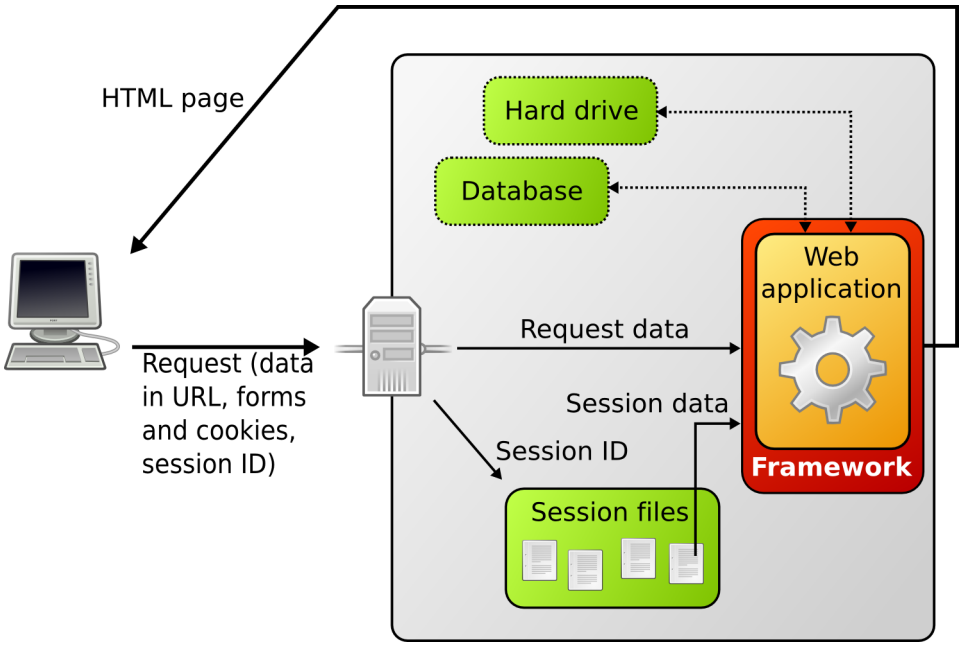
2.2 Projekt Vývojový systém webových aplikací

U webových aplikací lze pozorovat funkcionalitu aplikací desktopových. Desktopové aplikace uchovávají informace o svých proměnných v paměti uživatelova počítače a tyto určují její stav. Na rozdíl od nich ale jsou webové aplikace bezstavové — skript, který generuje stránku sice využívá paměť serveru, do níž ukládá svá data, ale po vygenerování stránky jsou tato data z paměti odstraněna. Jejich stav je tedy zapotřebí uchovávat jiným způsobem. Tímto způsobem je serializace těchto dat a jejich vložení mezi informace, které si vyměňuje klient

se serverem.

„Vývojový systém webových aplikací“ je projekt, jehož snahou je usnadnění tvorby webových aplikací pomocí *automatického uchovávání stavu* aplikací. Projekt nabízí framework, který rozšiřuje standardní funkce PHP o možnosti uchování a předávání stavu proměnných a objektů aplikací, čímž vývoj a práci s aplikacemi v PHP přibližuje k aplikacím desktopovým. Framework vytváří vrstvu nad aplikací, zajišťuje předzpracování požadavků a obnovení stavu aplikace, která tak nepozná, že byl její chod přerušen.

Další vlastností, kterou framework sbližuje webové a desktopové aplikace, je zavedení tzv. modifikačních funkcí. Tyto funkce jsou používány pro změnu stavu aplikace a jsou vyvolávány na základě událostí, které vznikají při interakci uživatele s aplikací.



Obrázek 2.2: Zpracování požadavku webovou aplikací s vývojovým systémem

Pomocí automatického uchování stavu framework výrazně ulehčuje programátorům práci s daty, s nimiž by jinak musel pracně manipulovat a zavedením zpětných volání sbližuje webové a desktopové aplikace. Vývojový systém tak umožňuje programátorům se zkušenostmi s vývojem desktopových aplikací a alespoň základní znalostí jazyka PHP efektivně vyvíjet webové aplikace.

Projekt „Vývojový systém webových aplikací“ poskytuje dobrý teoretický základ, který lze dobře využít a pomocí vhodných úprav posunout celý systém blíže stavu, kdy jej programátoři budou moci začít používat pro vývoj webových aplikací.

Kapitola 3

Specifikace požadavků

V této kapitole budou specifikovány požadavky, které by měl výsledný systém splňovat. Zmíněny budou zejména požadavky na vlastnosti systému, jeho použití a návaznost na další technologie včetně původního Vývojového systému webových aplikací.

Výsledný systém by měl poskytovat programátorovi vývojový nástroj, v němž bude moci vyvíjet webové aplikace způsobem velmi blízkým způsobu vývoje aplikací desktopových. Systém by měl být schopen použít jakýkoli programátor se znalostí PHP, objektového programování a zkušenostmi při vývoji desktopových aplikací. Tento nástroj by tedy měl

- být *čistě objektově orientovaný*,
- zajišťovat *automatické uchování stavu aplikace*,
- poskytovat možnost *změny stavu pomocí modifikačních funkcí*,
- umožňovat *dobrou rozšiřitelnost* a
- podporovat automatické *generování validního HTML kód pomocí objektů*.

Je žádoucí, aby systém byl snadno použitelný, intuitivní a při vývoji efektivní. Proto je zapotřebí, aby

- poskytoval *vhodné implicitní hodnoty*,
- pokrýval alespoň základní a často využívanou funkcionalitu a
- používal vhodně a jednotně pojmenované metody a atributy.

S provozováním webových aplikací vyvíjených pomocí jazyka PHP souvisí i další rozličné technologie a je tedy zapotřebí systém zasadit do jejich kontextu. Na systém nám tak vzniknou následující požadavky.

- Systém by měl využívat v co největším rozsahu využívat poznatky a implementované části původního vývojového systému.

- Aplikace vyvíjené pomocí systému by mělo být možné *použít bez technologie JavaScript*. Zároveň by ale programátor měl mít možnost JavaScript v aplikaci využívat.
- Aplikace musí mít možnost *využívat technologii CSS* (Cascading Style Sheets, česky kaskádové styly) a to jak pomocí externích souborů tak pomocí atributu `style` jednotlivých entit.
- Pro provoz aplikace na serveru by měl postačovat pouze s webový server s podporou skriptování jazykem PHP a potřebnými knihovnami.

Kapitola 4

Návrh

V této kapitole bude nejdříve současný Vývojový systém webových aplikací konfrontován se specifikací požadavků. Následovat bude popis návrhu změn v současném frameworku vývojového systému, návrh provedení zpětných volání a způsobu navázání tříd na framework. Dále budou popsány struktura, chování a použití jednotlivých modulů pro vytváření aplikací pomocí vývojového systému a v poslední části bude navržena jednoduchá aplikace pro demonstraci použití vybraných modulů.

4.1 Nedostatky současného vývojového systému

Současná implementace frameworku vývojového systému má i přes svůj přínos také své nevýhody. Budeme-li jej konfrontovat s požadavky definovanými v kapitole 3, nalezneme některé problémy. V této podkapitole budou tyto problémy rozebrány, popsány jejich dopady a uvedena možná řešení.

Jedním z podstatných problémů je implementace pomocí PHP verze 4, která je v současnosti již jen zřídka používána pro vývoj nových projektů. Je tomu tak zejména díky nedokonalé podpoře dnes stále oblíbeného a častěji používaného objektově orientovaného programování (postrádá např. rozlišení soukromých a veřejných atributů či funkcí). Tento problém výrazně brání rozšíření systému mezi komunitu, protože přidává pracnost při případné integraci do nového systému postaveného na v současnosti nejvíce používaném PHP 5. Bude tedy vhodné systém přenést pod novou verzi jazyka PHP, která má podporu objektového přístupu na mnohem vyšší úrovni.

Mluvíme-li o objektově orientovaném přístupu, stojí také za zmínku fakt, že třídy objektů s nimiž framework pracuje jsou na sobě většinou nezávislé — nevyužívají dostatečně možností, které poskytuje vztah generalizace–specializace. Systém postrádá bod, jehož by se dalo využít k jednotnému napojení dalších rozšíření na framework. Současné objekty navíc postrádají podporu pro jednotný životní cyklus, respektive možnost využít jeho jednotlivé fáze, v důsledku čehož se vyvíjená aplikace může stát nepřehledná a složitější na vývoj a správu. Zde se jako nejvhodnější řešení nabízí navrhnout třídu objektů, která bude imple-

mentovat jednotný životní cyklus a bude děděna všemi třídami pracujícími s frameworkem.

Dalším problémem je, že současný framework používá pro generování cílového HTML textové šablony, do nichž jsou vkládány výstupy jednotlivých objektů. Tímto je ale obcházen čistě objektový návrh, což je v rozporu s požadavky tohoto projektu (dobrá rozšiřitelnost komponent) a navíc je znesnadněn i jeden z cílů původní práce — návrh a sestavování komponent pomocí grafického studia. Řešením tohoto problému je změna struktury komponent pro generování HTML kódu na čistě objektovou.

U zpětných volání, která framework podporuje, lze nalézt další problém. Modifikační funkce musí být programátorem implementována v třídě aplikace a je následně volána přímo frameworkem v závislosti na požadavku od uživatele. Nutnost implementovat všechna zpětná volání na jednom místě v systému na jednu stranu zvyšuje přehlednost, ale je zároveň řešením málo pružným a pro programátora může být omezující. Vhodné by tedy bylo rozšířit zpětná volání tak, aby modifikační funkce mohly být implementovány i do ostatních objektů.

Zmíněné problémy mají negativní vliv zejména na dobrou rozšiřitelnost systému a zastiňují výhody, které nám poskytuje automatické ukládání stavu aplikace. Jejich řešením je přepracované, plně objektově orientované jádro využívající funkcionalitu původního a vylepšující zejména rozšiřitelnost pomocí dědičnosti tříd.

4.2 Přepracování jádra

Vzhledem k faktům zmíněným v předešlé podkapitole je tedy na místě přepracovat jádro vývojového systému tak, abychom se vyhnuli nastíněným problémům. I když toto není přímo předmětem této diplomové práce, velmi úzce to souvisí s druhým bodem zadání, protože bez tohoto kroku nebude možné s frameworkem jednoduše pracovat.

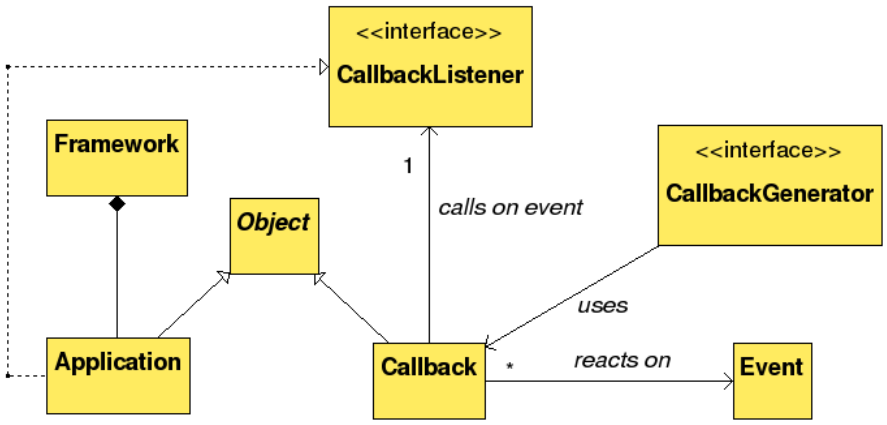
Stávající jádro zahrnuje dvě třídy objektů — `Framework` a `Application`. `Framework` je kontejnerem, v němž se uchovávají aplikace a je jediným vstupním bodem pro příchozí požadavky. Stává se tak další vrstvou nad webovými aplikacemi. Jeho hlavním úkolem je správa aplikací a kontrola požadavků. Zpětná volání jsou reprezentována pomocí řetězce předávaného v požadavku.

Nové jádro (viz obrázek 4.1) rozšiřuje původní o abstraktní třídu `Object` (viz kapitola 4.3), která je bodem pro další rozšiřování frameworku. Dále jsou přepracována zpětná volání, která jsou volána na základě událostí a reprezentována objekty třídy `Callback` a pro práci s nimi jsou navržena nová rozhraní (viz kapitola 4.5).

Framework nadále zůstává vstupním bodem pro veškeré požadavky od uživatele, kontroluje jejich správnost a časovou známku[1]. Navíc se ale také stává jediným výstupním bodem, který zasílá data klientovi. Zajišťuje generování hlaviček stránky a dalších částí internetové stránky. Framework se také stále chová jako kontejner pro aplikace, které jsou navrženy podle architektonického vzoru model–pohled–řadič (viz kapitola 4.4), a obstarává

[1] Kontrola časové známky se provádí proto, aby nedocházelo k opakovanému volání téže funkce vyvolaného např. použitím tlačítek „back“ nebo „refresh“ v prohlížeči.

vytváření jejich instancí a následnou správu.



Obrázek 4.1: Diagram tříd přepracovaného jádra systému

4.3 Třída Object

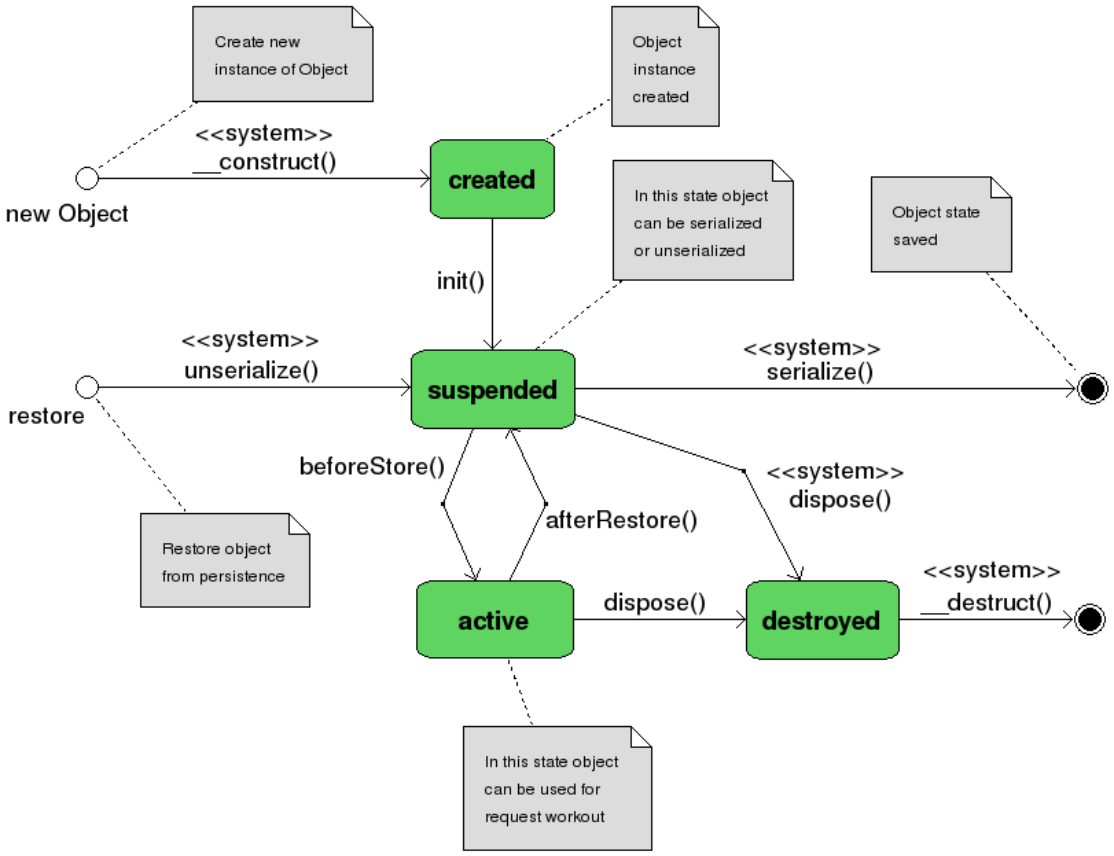
V kapitole 4.1 bylo naznačeno, že současnému vývojovému systému chybí jednotný bod, na který by bylo možné navazovat další rozšíření vývojového systému. Řešením tohoto problému je třída `Object`. Ta představuje základní stavební kámen všech aplikací pracujících nad frameworkem vývojového systému. Všechny nové třídy, u nichž předpokládáme, že jejich instance budou v systému existovat přes několik cyklů aplikace, by měly dědit právě tuto třídu, protože poskytuje základní vlastnosti jako je porovnávací mechanismus, jednotný životní cyklus objektů ve vývojovém systému a také některé další, které objekty v jazyce PHP standardně nepodporují.

Všechny instance třídy `Object` mají v rámci všech aplikací daného uživatele unikátní identifikační číslo, které umožňuje objekty mezi sebou bezpečně odlišit. Právě tohoto čísla využívá k porovnávání objektů metoda `equals()`, kterou třída obsahuje.

Každý objekt třídy `Object` má definován základní životní cyklus (viz. obrázek 4.2). Programátor díky tomu může reagovat na události, které objekt ovlivňují při běhu aplikace. Jedná se o

- vytváření objektu, během něhož je volána funkce `init()`,
- obnovení objektu, po kterém je ihned volána metoda `afterRestore()`, a
- uložení stavu objektu, před nímž je volána funkce `beforeStore()`.

Během provádění metod `afterRestore()` a `beforeStore()` by neměly být volány metody jiných objektů pracujících nad frameworkem. Není totiž určeno, v jakém pořadí se objekty před serializací suspendují, a tak nemáme jistotu, že jsou v aktivním stavu. Ve



Obrázek 4.2: Stavový diagram zachycující životní cyklus objektů

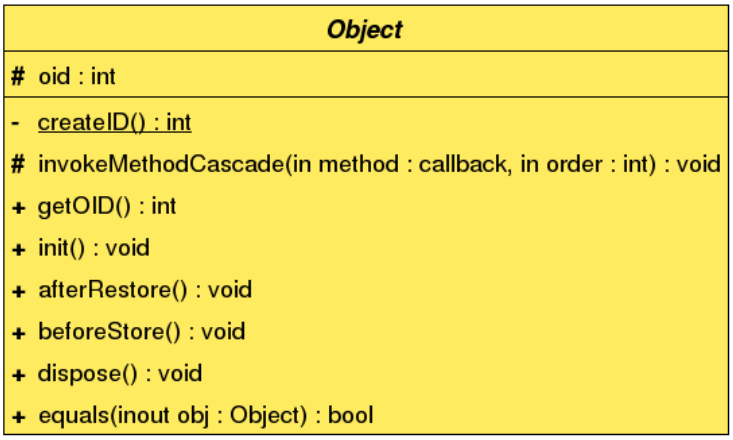
výsledku by pak mohlo docházet nejen k vracení chybných hodnot, ale volaná komponenta by se mohla dostat do neplatného stavu.

Další mechanismus, kterou poskytuje třída `Object` je *kaskádové volání metod*. Při vývoji aplikací pomocí jazyka PHP se můžeme setkat s potřebou implementovat stejně pojmenované metody ve více třídách vzájemně propojených vazbou generalizace–specializace a pak všechny tyto jednoduše volat v určeném pořadí. Třída `Object` tedy poskytuje mechanismus, který umožňuje pomocí *PHP Reflection API*[2] postupně zavolat danou metodu pro každou třídu, kterou daný objekt dědí.

V praxi nám tento mechanismus např. umožňuje provést inicializaci objektu pomocí metod `init()` implementovaných ve více třídách (jež jsou ve vztahu generalizace–specializace) tak, že se postupně volá jedna za druhou v určeném pořadí. Tento mechanismus navíc neumožňuje použít objekt bez volání metod `init()` a omezuje tak nepovolené používání některých tříd. Stejného mechanismu systém využívá i pro volání dalších metod (např. `beforeStore()`, `afterRestore()`) a lze jej užít i u dalších, programátorem definovaných,

[2] API umožňující přímý přístup k objektům a třídám. Díky možnosti vytvářet objekty, volat metody atp. je častou využíváno právě různými frameworky.

metod.



Obrázek 4.3: Třída Object v UML

Díky možnosti jednoznačné identifikace objektů, definovanému životnímu cyklu, který je podporovaný frameworkem, a mechanismu pro kaskádové volání metod se třída Object stává dobrým základem pro další rozšiřování celého systému. Pomocí tohoto mechanismu není složité navázat na framework i třídy, které již existují.

4.4 Model–pohled–řadič

V desktopových aplikacích se často můžeme setkat s oddělením prezentační vrstvy a modelem, jenž zabezpečuje přístup k datům, jejich správu a výpočty nad nimi. Tento přístup výrazně zvyšuje přehlednost aplikace a vymezuje funkcionalitu komponent.

Model–pohled–řadič (anglicky. model–view–controller, zkráceně MVC, viz [4]) je architektonický vzor, v němž jsou od sebe odděleny datový model, pohled a řídící logika. Vzniklé komponenty jsou na sobě nezávislé a modifikace jedné má minimální vliv na ostatní.

Datový model je doménově specifická reprezentace dat, s nimiž aplikace pracuje. Doménová logika dává konkrétní význam holým datům uloženým např. v databázi či XML souboru (model tedy zapouzdřuje přístup k databázím). Pro datový model není prezentační vrstva ani řadič viditelný — tyto komponenty volají model.

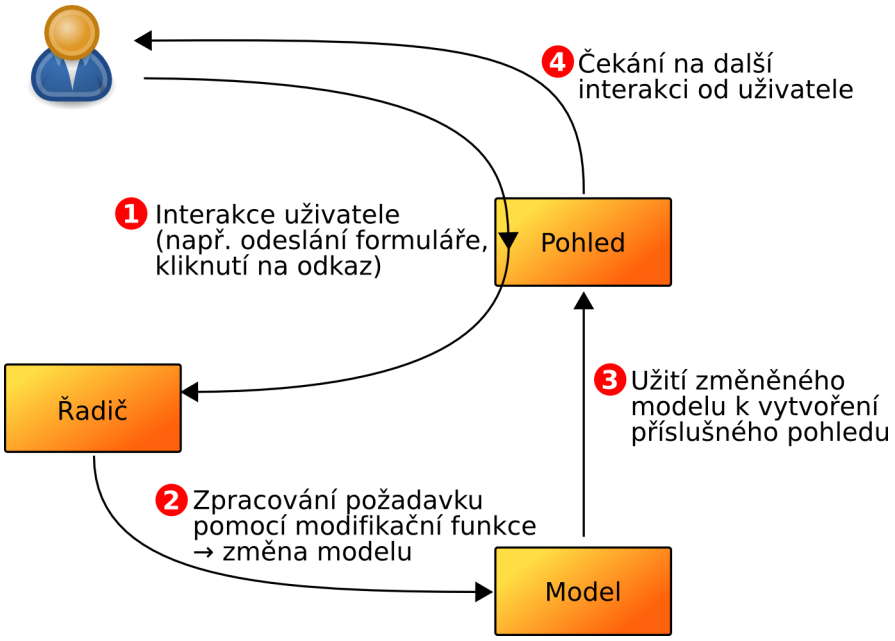
Pohledové komponenty zajišťují transformaci dat z modelu do formy vhodné pro komunikaci s uživatelem. V kontextu tohoto projektu se bude jednat o objekty generující HTML kód právě v závislosti na modelu.

Řadič reaguje a odpovídá na události (v našem případě na požadavky uživatele). V závislosti na požadavku může měnit model a následně vybírá správný pohled (příslušnou stránku, chybovou stránku...).

Model MVC je častější zejména u webových aplikací, u nichž není takové množství událostí jako u aplikací desktopových, ale na druhou stranu jsou tyto události (v podobě

požadavků) často variabilnější. Řadič je pak vhodným místem pro zpracování událostí, na jejichž základě pak rozhoduje o způsobu změny modelu a zobrazení dat uživateli.

Celý koncept pak funguje podle schématu na obrázku 4.4. Uživatel provede v uživatelském rozhraní nějakou akci (např. klikne na odkaz), řadič zpracuje požadavek pomocí zpětného volání, které přistupuje k modelu a mění ho (např. změní obsah uživatelova nákupního košíku). Následně pohled využije model k vygenerování uživatelského rozhraní (např. vytvoří seznam položek v košíku). Uživatelské rozhraní pak čeká na další akci uživatele.



Obrázek 4.4: Princip aplikace využívající vzor MVC

Rozdělení aplikace na několik navzájem nezávislých komponent zvyšuje přehlednost aplikace, usnadňuje návrh a má výhody při změnách v projektu — můžeme např. změnit pohled aniž bychom měnili model nebo naopak můžeme reorganizovat data aniž bychom museli měnit uživatelské rozhraní. Dalšími výhodami menších nezávislých komponent jsou vyšší odolnost systému proti chybám, zmenšení úsilí nutného k jejich nalezení, lepší možnosti testování či snazší rozdělení práce ve vícečlenném týmu.

Jazyk PHP umožňuje programování různými styly a vzhledem k absenci standardizace se zejména u začínajících programátorů stávají výsledné aplikace komplikované a nepřehledné. Architektura MVC nutí takové programátory vytvářet aplikace s využitím praxí ověřených praktik s dobrou spravovatelností a rozšiřitelností, což může být přínosem pro ně samotné i pro komunitu (přehlednější aplikace snadněji pochopí i jiní programátoři).

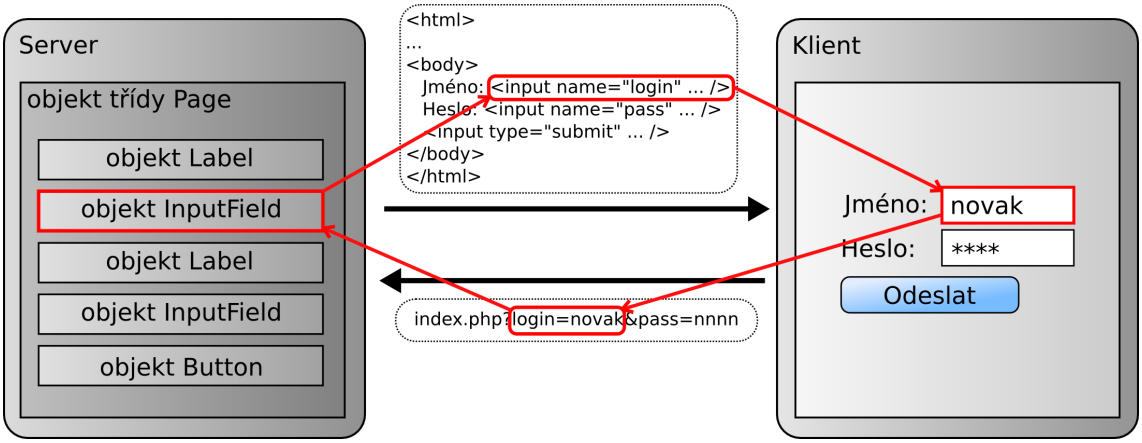
Vzhledem k výhodám, které nám tento model nabízí, je použit jako vzor pro aplikace pracující s vývojovým systémem. Dále budou popisovány jednotlivá rozšíření v jednotlivých částech tohoto konceptu.

4.5 Události a zpětná volání

Z předchozích kapitol jsme se dozvěděli, že chceme usnadnit programátorovi práci tím, že sblížíme webové a desktopové aplikace. Dříve nastíněný rozdíl mezi těmito druhy aplikací týkající se uchování stavu a možnosti aplikace přístupu k nim se projevuje i na prezentační vrstvě. Rozdíly, které tak vznikají, je zapotřebí nějakým způsobem překlenout.

Grafické uživatelské rozhraní desktopových aplikací je v moderních systémech vytvářeno pomocí objektů, které mají své specifické vlastnosti (velikost, barvy, text apod.) a metodu, která umožňuje vytvořit grafickou reprezentaci každého takového objektu. Tato metoda je využívána aplikací pro vykreslování objektů na obrazovku. Stav těchto objektů je uložen v paměti uživatelova počítače a měněn pomocí modifikačních funkcí, které jsou volány v závislosti na událostech nad objektem zaznamenaných (např. pohyb myši, kliknutí nebo stisk klávesy). Probíhá tedy častá komunikace mezi objektem a aplikací, která objektu zasílá události vznikající během interakce s uživatelem či jiným programem.

Přiblížit se tomuto modelu na úrovni webových aplikací nese jistá úskalí, protože webové aplikace jsou oproti desktopovým mnohem méně pružné. Komunikace uživatele s webovou aplikací běžící na serveru probíhá na základě dynamicky generovaných internetových stránek zobrazovaných prostřednictvím internetového prohlížeče. Internetový prohlížeč všechny uživatelovy akce zachycuje, mění podle nich uživatelské rozhraní a se serverem komunikuje jen v případě nutnosti. Chová se tak jako zásobník událostí a pokud zaznamená událost vyžadující komunikaci se serverem (odeslání formuláře, kliknutí na odkaz apod.), jsou všechny doposud nashromážděné události transformovány v požadavek, který je zaslán serveru — výsledkem všech těchto událostí jsou v podstatě nové hodnoty ve vstupních polích a informace o tom, kterým tlačítkem byl formulář odeslán.

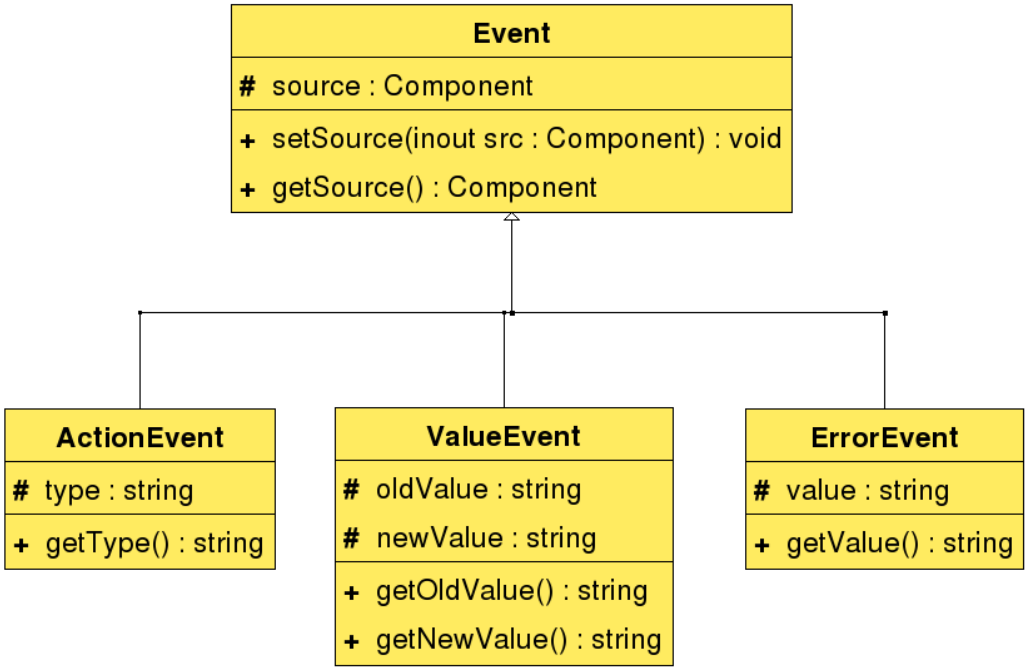


Obrázek 4.5: Komunikace prezentační vrstvy na úrovni serveru a klienta

Požadavek od uživatele obsahuje pouze data, která jsou výsledkem jeho akcí v internetovém prohlížeči. Z tohoto balíku dat lze ovšem určit dva druhy událostí, které jsou pro aplikaci na serveru velmi důležité.

- Změnu hodnoty lze pro komponenty, jejichž hodnotu může uživatel měnit (tj. textová pole, výběry, zaškrtávací pole apod.), určit v závislosti na vztahu staré a nové hodnoty pole.
- Použití daného tlačítka či odkazu můžeme získat pomocí vhodného kódu HTML.

Všechny tyto události se mohou samozřejmě vztahovat pouze ke komponentám, které byly zobrazeny na stránce. Změn hodnoty může být v jednom požadavku i více (např. uživatel vyplnil více polí), ale použití tlačítka či odkazu jen jedno, protože se jedná o událost vyžadující komunikaci se serverem.



Obrázek 4.6: Diagram tříd událostí v UML

Protože stránky a jejich komponenty generujeme pomocí objektů, jejichž stav uchováváme, vzniká jistá redundance dat. Celá prezentační vrstva je uchovávána na serveru, navíc je ale její část předávána prohlížeči (viz obrázek 4.5). Tyto dvě redundantní části si předávají navzájem řízení (server posílá prohlížeči HTML stránky, klient serveru zase požadavky), a tak je v jednu chvíli aktivní pouze jedna z nich, zatímco druhá čeká. Během předávání řízení je zapotřebí tyto části synchronizovat. Data v prohlížeči se chovají jako mezipaměť, která je příchozí stránkou vždy celá znovu naplněna — není tedy potřeba data dále zpracovávat a prohlížeč je může rovnou převést do grafické reprezentace uživatelského rozhraní. Naopak data příchozí z prohlížeče do aplikace jsou jen částí většího celku a je zapotřebí je do tohoto celku nějakým způsobem zařadit.

Synchronizace dat z prohlížeče probíhá voláním metody, kterou musí všechny objekty na

prezentační vrstvě implementovat. V ní dochází ke zpracování dat z požadavku, při kterém mohou být identifikovány výše zmíněné události nad jednotlivými entitami. Pokud tedy došlo ke změně hodnoty nebo k akci nad tlačítkem nebo odkazem je generována událost (objekt třídy `Event`, viz obrázek 4.6), která je postoupena řadiči.

Během synchronizace může dojít ke stavu, kdy uživatel v prohlížeči zadal do některého z objektů hodnotu, která pro daný objekt není přípustná. Bez použití technologie *JavaScriptu* tuto skutečnost nelze ošetřit dříve než během provádění synchronizace. Pro tento případ je navržena událost `ErrorEvent`, kterou komponenta během synchronizace vygeneruje.

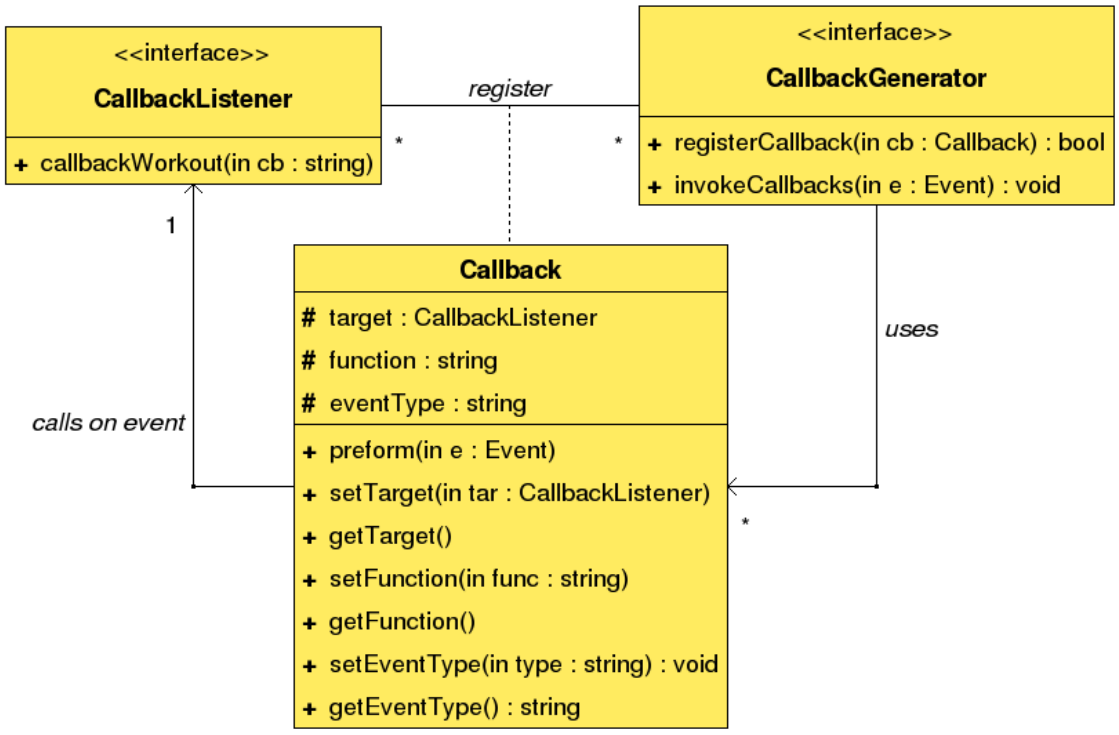
Všechny události, které byly komponentami identifikovány, shromažďuje řadič a následně je vhodně poskládá. Aplikace na serveru neví, v jakém pořadí uživatel akce prováděl (zná pouze jejich výsledek) a tak musí události poskládat logicky za sebe. Prioritní jsou chybové události ErrorEvent. Po nich jsou zpracovány události týkající se změn hodnot a nakonec akce provedené nad tlačítkem. V případě, že uživatel použil odkaz, bude identifikována pouze tato událost, protože zároveň s odkazem nejsou odesílána žádná formulářová data.

Následuje zpracování událostí. Prioritně se zpracovávají chybové události — na ně reaguje přímo řadič. Pokud se ve frontě událostí objeví jedna či více chybových událostí, přejde řadič do chybového stavu. V tomto stavu vygeneruje řadič stránku s chybovými hláškami, na které chybné hodnoty nahradí hodnotami původními nebo implicitními. V chybovém stavu zakáže řadič provedení zpětných volání, a tak nemůže dojít ke změně stavu aplikace s nepřípustnými hodnotami. Ve frontě ale zůstávají události změn hodnot, aby mohla být provedena zpětná volání až budou všechny hodnoty platné. Naopak události akcí postrádají význam a jsou tedy zahazovány[3].

Nezaznamená-li řadič žádné chybové události, začne se prováděním zpětných volání. Zpětná volání si registrují objekty, které implementují rozhraní `CallbackListener` u generátorů zpětných volání, tj. objektů implementujících rozhraní `CallbackGenerator` (viz inicializační část na obrázku 4.8). Zpětné volání se vždy registruje pro nějaký typ události. Pokud tato událost nastane, je postoupena zpětnému volání, které volá předem danou metodu „posluchače“.

Oproti desktopovým aplikacím dochází k výrazné redukci událostí, na něž může aplikace reagovat. Na druhou stranu se ale jedná o ty nejdůležitější a také nejpoužívanější události z hlediska funkcionality a komunikace prezentační vrstvy s ostatními vrstvami aplikace. Ostatní události, které běžně poskytují desktopové aplikace, se většinou používají pouze v prezentační vrstvě a lze je případně zpracovávat přímo v prohlížeči pomocí jazyka JavaScript.

[3] Analogií v desktopových aplikacích by mohl být případ, kdy tlačítko pro odeslání hodnot by v případě, že některé pole obsahuje špatná data, nebylo aktivní. Pak by veškeré akce na tomto tlačítku byly také ignorovány až do chvíle, kdy jsou hodnoty v polích validní.

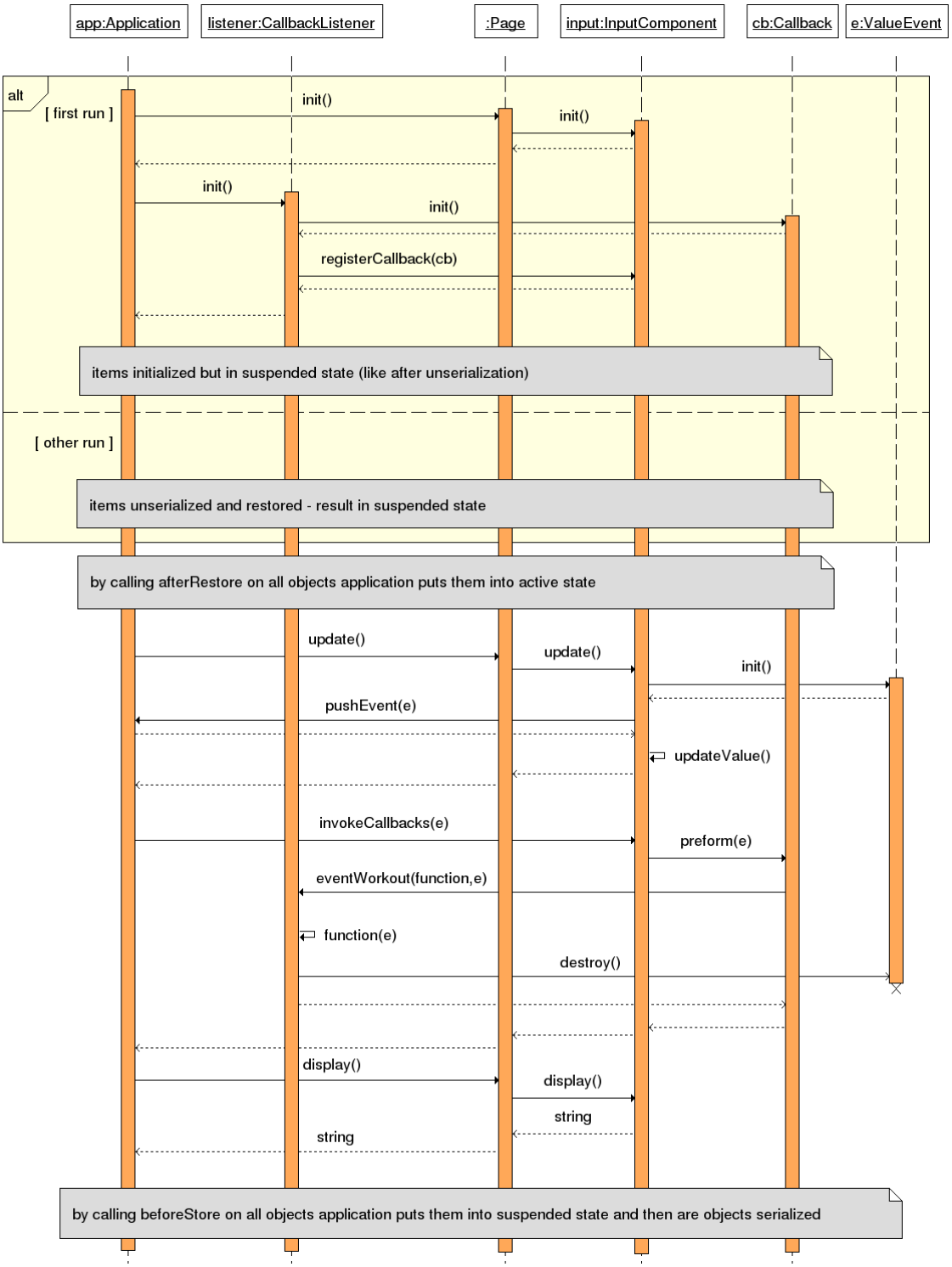


Obrázek 4.7: Diagram tříd pro práci se zpětnými voláními

4.6 Aplikace

Aplikace může programátor ve vývojovém systému vytvářet děděním třídy `Application`. V architektuře MVC představuje aplikace řadič, který přijímá frameworkem předzpracované požadavky od uživatele a řídí jejich zpracování.

Přijetím požadavku je spuštěn životní cyklus aplikace. Pokud se jedná o první spuštění aplikace, je tato inicializována včetně všech jejích komponent. Během inicializace jsou objektům nastaveny výchozí hodnoty a nachází se tak v stejném stavu jaký mají objekty po obnovení. Pokud aplikace již byla inicializována dříve, je obnoven stav dříve vytvořených objektů. Pak jsou všechny objekty aplikace uvedeny do aktivního stavu voláním jejich metody `afterRestore()` pomocí kaskádového volání (viz kapitola 4.3). Následuje volání metody `update()` aktuální stránky. Ta je implementována jako kontejner, který volá metodu `update()` všech svých položek. V metodě update jsou kontrolovány uživatelem provedené změny hodnot a akce a následně generovány případné události, které jsou zařazeny do fronty událostí aplikace. Je-li to zapotřebí a událost není chybová, provede se změna stavu dané komponenty na straně serveru (např. změna hodnoty) a řízení je vráceno zpět aplikaci. Aplikace začne se zpracováním událostí — volá odpovídající komponenty, které v závislosti na události využijí zpětné volání k vyvolání modifikační funkce. Poté je vytvořena odpověď pro framework, pro níž si aplikace od stránky vyžádá název a vygenerování obsahu. Nako-



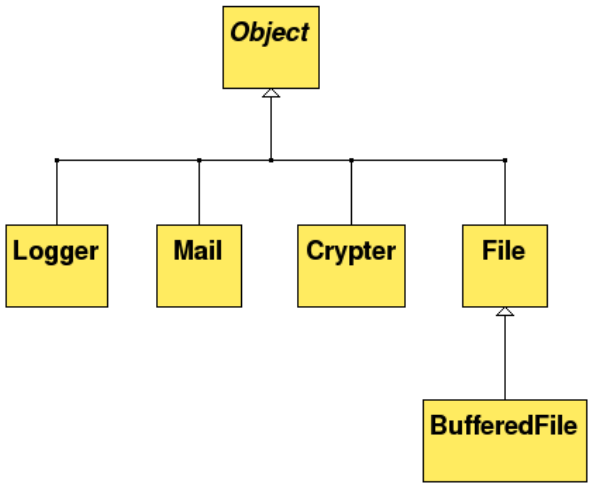
Obrázek 4.8: Diagram sekvence pro zpracování požadavku

nec jsou objekty kaskádovým voláním metody `beforeStore()` připraveny pro serializaci a uložení stavu.

Vzhledem k faktu, že jazyk PHP nevznikl jako čistě objektově orientovaný a také jím není, narážíme na problém v podobě globálních proměnných, které jsou jedním z aspektů určujících stav aplikace a zároveň často používanou praktikou. Uchování stavu se provádí pomocí serializace objektu aplikace a proto si aplikace udržuje přehled o globálních proměnných a využívá svých metod `beforeStore()` resp. `afterRestore()` k uložení resp. obnovení globálních proměnných, které jí náleží.

4.7 Model

V aplikaci model představuje doménově specifickou reprezentaci informací s nimiž aplikace pracuje. Zahrnuje aplikační vrstvu zajišťující přístup k datům (v souborech, databázi atp.) a dává jim konkrétní význam. Model a jeho funkcionalita se aplikaci od aplikace může diametrálně lišit a využívá širokého spektra funkcí. Chceme-li programátorovi umožnit efektivnější a jednodušší programování v této oblasti, je zapotřebí vystihnout často používané a zároveň nesnadno implementovatelné funkce jazyka PHP a vnesením jisté míry abstrakce a vhodných implicitních hodnot zmenšit objem kódu a znalostí nutných pro jejich použití. Tímto přístupem umožníme začínajícím programátorům použití těchto funkcí a zároveň neomezujeme programátory profesionály, kteří — pokud jim funkcionalita v modulu nestačí — mohou využít funkcí původních.



Obrázek 4.9: Třídy pro vývoj modelu

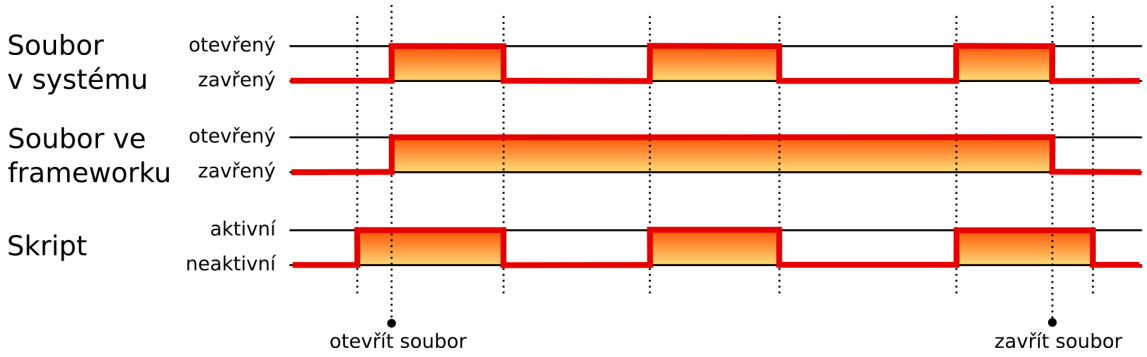
V následujících podkapitolách budou podrobně popsány návrhy rozšíření, která mají ulehčit práci programátora na doménové funkcionalitě. Jejich přehled a vztahy jsou vidět na obrázku 4.9.

4.7.1 Práce se soubory

Soubor je jedním z prostředků umožňující dlouhodobé uchování dat. Použití souboru ve vývojovém systému standardním způsobem se poněkud liší od desktopových aplikací, jimž se chceme přiblížit. V desktopové aplikaci můžeme mezi otevřením a zavřením souboru zpracovat více požadavků, v nichž může a nemusí být soubor použit jak pro čtení tak pro zápis. Ve webových aplikacích můžeme soubor otevřít až během zpracování požadavku, tudíž mezi otevřením a zavřením souboru může proběhnout maximálně jedno zpracování požadavku.

Abychom mohli simulovat chování souborů tak, jak je tomu u desktopových aplikací, je zapotřebí, aby byl soubor otevřen na začátku zpracování každého požadavku a na konci byl opět uzavřen zároveň s uložením informací o stavu souboru. Stav souboru je dán třemi aspekty. Jedná se o obsah souboru, způsob otevření (zavřený, otevřený pro čtení, otevřený pro zápis atd.) a v případě, že je soubor otevřen, také pozice kurzoru v něm.

Ukládání stavu obsahu souboru ovšem skrývá jistá úskalí. První problém spočívá v zachování integrity dat v souboru. Soubor je totiž zdroj, k němuž může přistupovat více skriptů zpracovávající požadavky zároveň. Navíc je nutné rozlišovat otevření souboru na úrovni souborového systému a na úrovni frameworku (tento vztah je zachycen na obrázku 4.10). Zatímco je soubor otevřen na úrovni frameworku, na úrovni souborového systému je opakovaně otevírán a uzavírán. Je tedy nutné zavést mechanizmus, který zajistí, aby nedocházelo ke kolizím během přístupu k souborům. Tímto mechanizmem je systém zámkových souborů. Pro každý otevřený soubor je vytvořen další soubor s metadaty zahrnujícími informace o způsobu otevření konkrétními objekty pracujícími nad frameworkem.



Obrázek 4.10: Změny stavu souboru v souborovém systému a frameworku v průběhu času

Pro správnou funkčnost zámkových souborů je zapotřebí, aby skripty měly možnost zápisu do předem určeného adresáře. Do něj se budou ukládat zámkové soubory, které budou pojmenované pomocí funkce `sha1()`, jíž bude jako argument předáno celé jméno souboru i s cestou.

Podíváme-li se, jak se soubory pracují desktopové aplikace, k jejichž chování se chceme přiblížit, najdeme další problém. Ty totiž často nezapisují změny do souboru až do chvíle,

Aktuální mód	Použití mezipaměti	Označení	Další povolené módy					
			R	Rb	W	Wb	A	Ab
Čtení	ne	R	X	X		X		X
	ano	Rb	X	X	X	X	X	X
Zápis	ne	W						
	ano	Wb	X	X				
Připisování	ne	A					X	X
	ano	Ab	X	X			X	X

Tabulka 4.1: Módy otevření souboru v závislosti na současném stavu

než je soubor uzavřen. Simulovat toto chování znamená ukládat lokální kopii souboru a po uzavření souboru přepsat původní soubor změněnou kopií. Tento přístup ovšem vyžaduje operace, které u velkých souborů mohou výrazně zatěžovat disk a to zejména při častém otevírání a zavírání souborů na úrovni frameworku. Na druhou stranu poskytuje lepší možnost práce více objektů s jedním souborem, protože soubory jsou ukládány do mezipaměti, k níž přistupují pouze vlastnické objekty, a tak navzájem nekolidují. Možnosti přístupu, pokud je soubor již otevřen jiným procesem, shrnuje tabulka 4.1.

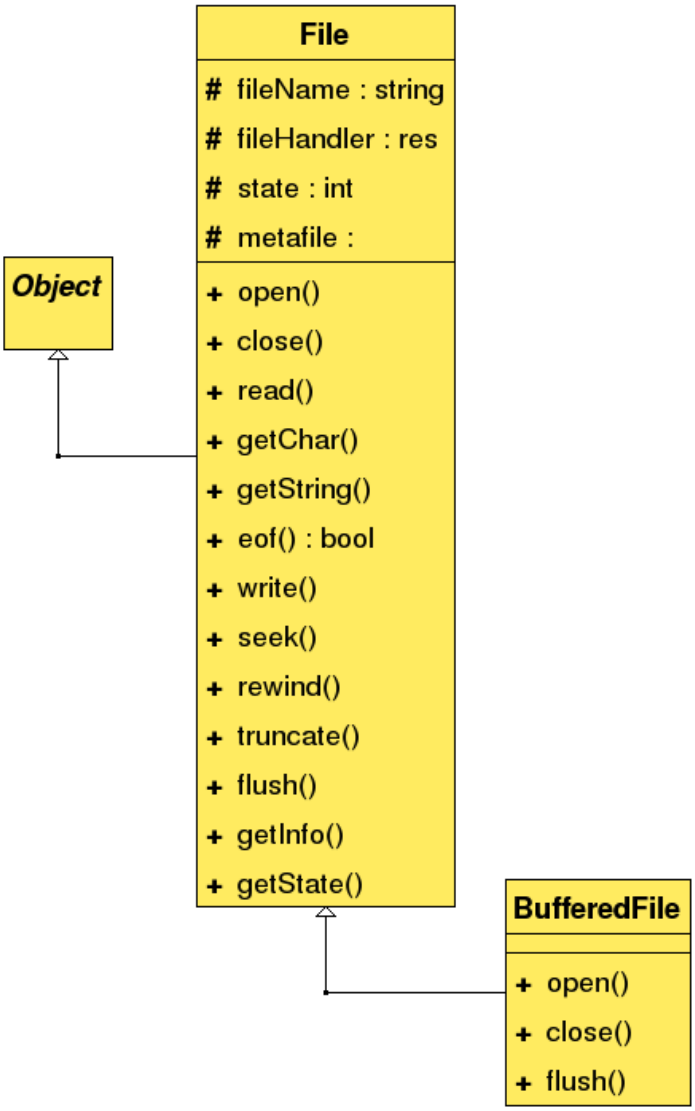
Pro práci se soubory má programátor možnost použít dvě třídy objektů (viz obrázek 4.11) lišící se načítáním souboru do mezipaměti:

- `File` je třída objektů určených pro rychlou práci se soubory, časté otevírání a zavírání (typicky pro soubory otevřené a zavřené během několika málo požadavků) a pro velké soubory.

- `BufferedFile` jsou objekty naopak určené pro soubory otevřené po delší dobu nebo tam kde chceme, aby obsah soubor byl načten do mezipaměti a nedocházelo tak ke kolizím s ostatními procesy.

Při použití objektů `BufferedFile` existuje riziko, že uživatel přestane užívat aplikaci dříve, než je soubor na úrovni frameworku uzavřen. V systému by tak mohlo dojít k postupnému hromadění souborů. Je proto nutné (nejlépe ve frameworku) implementovat mechanismus starající se o odstranění starých, nepotřebných souborů představujících již nepoužívanou mezipaměť.

Funkcionalitu, kterou nabízí tento modul, je vhodné využívat zejména ve chvíli, kdy programátor ví, že soubor bude otevřen po dobu delší, než je jedno zpracování požadavku. Další vhodné použití se nabízí po rozšíření některé z těchto tříd tak, že třída nová bude zpracovávat konkrétní typ souboru (např. soubor s hodnotami oddělenými středníkem apod.).



Obrázek 4.11: Třídy objektů pro použití souboru

4.7.2 Modul pro logování

Tento modul v podobě třídy `Logger` (viz obrázek 4.12) je určen pro zápis rozličných událostí, které nastanou při běhu aplikace, do souboru. Každý záznam je označen časovým razítkem a typem zprávy, které jsou rozlišovány podle jejich závažnosti (viz tabulka 4.2).

Úroveň	**Chyba**	**Popis**
1	Fatální chyba	Systém nemůže pracovat dál.
1	Kritická chyba	Je vyžadován co nejrychlejší zásah administrátora.
1	Chyba	Některá z funkcí nemusí být dostupná.
2	Upozornění	Chyba, která nemusí v určitých situacích systém ovlivnit.
3	Poznámka	Hlášení o méně významných chybách.
3	Informace	Sdělování výsledků důležitých operací.
4	Ladící hlášení	Hlášení užívaná pro ladění aplikace.

Tabulka 4.2: Typy hlášení pro záznam

Programátor by má možnost si vybrat, jaký typ zpráv se bude do souboru zapisovat. Pro méně zkušené programátory je možné rozlišovat zprávy do několika úrovní a zvolit příslušnou úroveň detailnosti zápisu. Tyto úrovně jsou naznačeny v prvním sloupci tabulky 4.2.

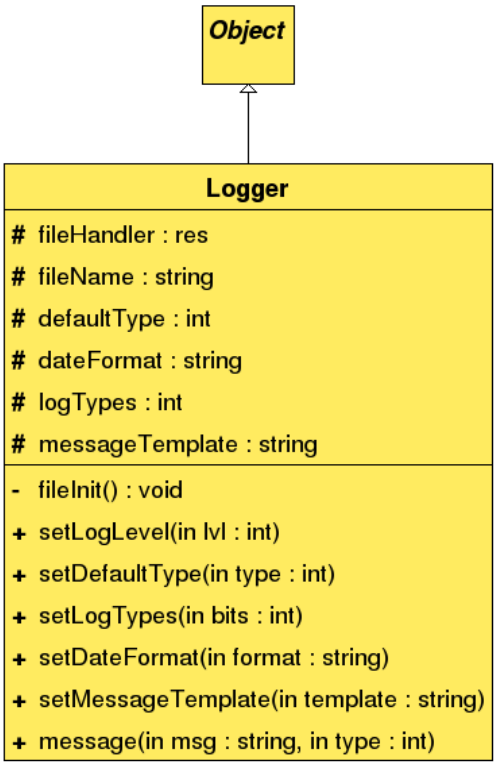
Logování probíhá voláním jedné metody, jejímiž argumenty jsou text zprávy a nepovinně typ zprávy. Pokud programátor typ nezadá, použije se buď typ implicitní nebo takový, jaký si uživatel přednastavil. Při zavolání metody se otevře příslušný soubor. Pokud neexistuje, tak se automaticky vytvoří i s příslušnými adresáři a následně se do něj připíše zpráva a časovou známkou a typem zprávy.

Pro tento modul jsou zvažována další rozšíření, která ale nebudou v rámci toho projektu implementována. Jedná se o možnost nastavení a odesílání zpráv do systémového logu (na Unixových systémech), možnost generování samostatných souborů pro jednotlivé úrovně či typy zpráv a nakonec možnost přidávání tzv. „observerů“ (pozorovatelů), tj. objektů, které budou volány, když do logu bude poslána zpráva konkrétního typu. Toto by mohlo mít využití např. pro odesílání e-mailových zpráv administrátorovi, ve chvíli, kdy je zapsána kritická či fatální chyba.

4.7.3 Modul pro odesílání elektronické pošty

Odesílání e-mailů z prostředí PHP je administrátory často užívaný prostředek pro upozorňování na nejrůznější události. Jeho využití je ale samozřejmě mnohem širší. Tento modul se zaměřuje zejména na ulehčení práce při odesílání elektronické pošty více uživatelům a nastavením nejvíce užívaných vlastností zpráv.

V jazyce PHP je implementována funkce `mail()`, která zajišťuje rychlé odesílání e-mailových zpráv na základní úrovni. Mnohým programátorům tato funkce stačí, ale její



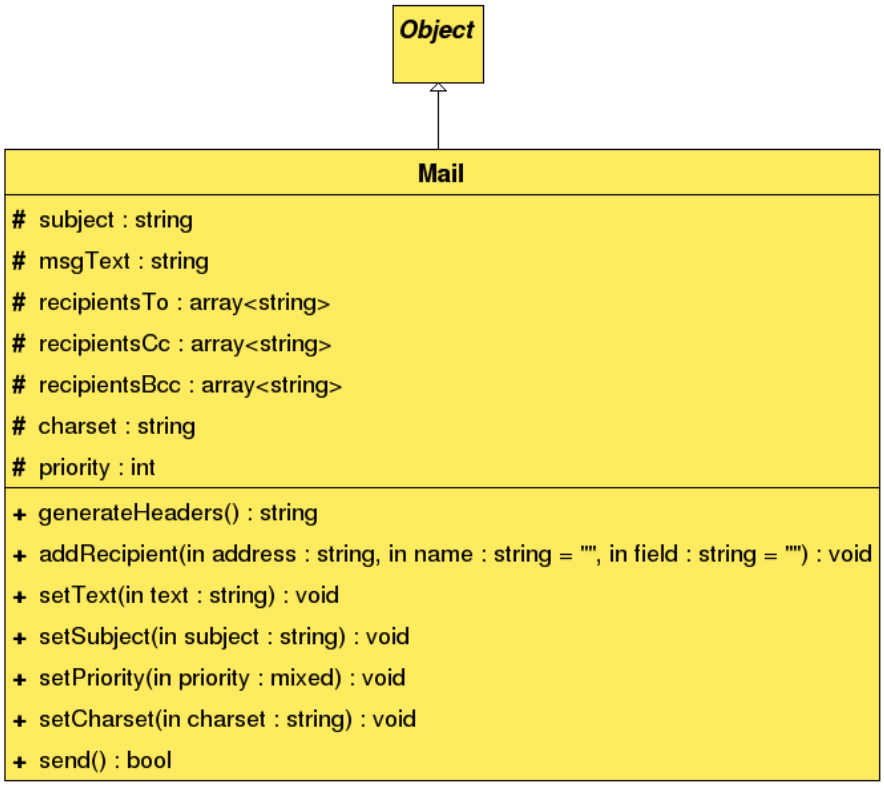
Obrázek 4.12: Třída pro logování popsaná pomocí UML

nevýhoda tkví v tom, že pokud chce programátor udělat něco málo víc než jen odeslat zprávu s nějakým předmětem na jednu či více e-mailových adres, je nucen použít parametr `header`, kam musí zapsat příslušné hlavičky zprávy (kódování, priorita, kopie, skrytá kopie). To je věc, která pro každého méně zkušeného programátora znamená nepříjemnost v podobě několika hodin strávených na Internetu vyhledáváním příslušných formátů. Zejména v neanglicky mluvících zemích pak roste potřeba těchto funkcí, protože e-mailové zprávy bez hlavičky mají implicitně nastavené kódování na ISO-8859-1, v němž nejsou některé znaky zobrazitelné (např. znaky z české abecedy).

Navrhované rozšíření je implementováno třídou `Email` (viz obrázek 4.13) a její instance budou reprezentovat konkrétní e-mailové zprávy k odeslání. Konstruktorem vytvoříme prázdnou zprávu a poté pomocí příslušných metod nastavíme její předmět, tělo, její příjemce a další příznaky a nakonec zprávu odešleme.

Objekty třídy `Email` mají metodu pro přidání příjemce `addRecipient()`. Tato metoda má tři parametry:

- adresa příjemce (povinný parametr),
- jméno příjemce (nepovinný parametr) a
- pole, do něhož se příjemce přidá, tj. běžný příjemce, kopie, nebo slepá kopie. Tento pa-



Obrázek 4.13: Třída pro odesílání e-mailů popsaná pomocí UML

rametr je nepovinný. Pokud programátor poslední parametr nepoužije, bude adresát implicitně přidán do pole běžných příjemců.

Metody `setSubject()` a `setText()` slouží k nastavení předmětu zprávy, resp. jejího obsahu. Další metody, `setFrom()` a `setReplyTo()`, umožňují nastavit adresy pro odpovědi. Obě jako parametry přijímají adresu a jméno.

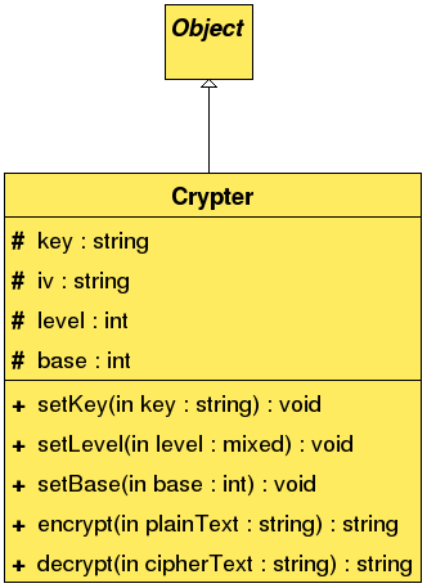
Pomocí metody `setEncoding()` může programátor nastavit kódování zprávy. Funkce má pouze jeden parametr a to právě název cílového kódování. Kódování je součástí hlavičky popisující typ zprávy a součástí této hlavičky je také typ obsahu, který je tudíž také potřeba nastavit. Mezi dva nejběžnější typy obsahu patří prostý text a text v HTML, které lze od sebe snadno automaticky rozeznat využitím regulárního výrazu. Pokud tělo neobsahuje HTML, je považováno za prostý text.

Dále třída obsahuje metodu pro nastavení priority zprávy. Prioritu je možné zadávat jako číslo nebo lépe jako řetězec (`high, normal, low`).

Tento modul napomáhá programátorovi při odesílání elektronických zpráv s nastavením některých standardních polí, která by jinak musel nastavovat pomocí hlaviček. Jedná se ale stále o zprávy obsahující pouze text, modul zatím neumí odesílat zprávy obsahující binární soubory či jiné přílohy.

4.7.4 Modul pro šifrování

V dnešní době se o bezpečnosti a kryptografii mluví stále více. Jsou vyvíjeny nové šifrovací algoritmy a postupy. V PHP je šifrování dostupné přes knihovnu resp. modul *MCrypt*. Pro jakékoliv zašifrování řetězce pomocí této knihovny je ovšem zapotřebí se alespoň na základní úrovni orientovat v šifrování, abychom věděli, jaký algoritmus jak použít, a dále je potřeba se zorientovat v příkazech zmíněného modulu. Toto by mělo být zpřístupněno vnesením určité míry abstrakce i běžnému uživateli. Navrhovaná třída `Crypter` (viz obrázek 4.14) tedy poskytuje programátorovi možnost jednoduchým způsobem šifrovat a dešifrovat řetězce.



Obrázek 4.14: Třída pro šifrování popsaná pomocí UML

Abstrakci zavedeme do té míry, že po programátorovi žádáme jen klíč, podle nějž se bude šifrovat, a úroveň zabezpečení, kterou požaduje (viz. tabulka 4.3). Pak už může do vytvořeného objektu pouze posílat řetězce k zašifrování či rozšifrování.

Dále má programátor možnost nastavit převádění šifrovaných řetězců na řetězce se znaky se základem 16 nebo 64. Řetězce získané ze šifrování totiž často obsahují netisknutelné znaky a programátor by mohl mít problémy při práci s nimi (např. při výpisu na stránku a opětovné poslání přes formulář by nemuselo proběhnout korektně).

Pro jednotlivé úrovně zabezpečení byly vybrány algoritmy podle jejich bezpečnosti, velikosti bloku a rychlosti šifrování. Pro nízkou úroveň je to metoda DES, střední a silné zabezpečení je prováděno metodou Rijndael pracující s 128 a 256-bitovými klíči, což je metoda, která je současným standardem pro šifrování.

Vzhledem k faktu, že málokterý programátor (zejména začínající) bude zapisovat klíč v 256-bitovém kódu, je nutné, aby vložený řetězec byl transformován právě do 256-bitového kódu. Toho dosáhneme pomocí HASH funkce `sha1()`. Ta sice generuje pouze 20-bajtový

Úroveň zabezpečení	Algoritmus	Velikost klíče
Nízká	DES	56 bitů
Střední	Rijndael	128 bitů
Vysoká	Rijndael	256 bitů

Tabulka 4.3: Algoritmy a velikosti klíče pro jednotlivé úrovně zabezpečení

kód, ale pokud ji použijeme několikrát na různé části řetězce, dostaneme potřebnou délku klíče.

Při kódování v jiném módu než ECB (z anglicky. Electronic code book) je zapotřebí tzv. inicializační vektor. Podle něj se inicializují hodnoty použité při šifrování a stejný vektor je zapotřebí užít i pro dešifrování. Vzhledem k faktu, že nechceme, aby uživatel musel zadávat kromě šifrovacího klíče další „klíč“ pro inicializaci, generuje se inicializační vektor automaticky. Nabízejí se dvě možnosti:

- Náhodné generování – vektor je generován náhodně a připojen k výsledku šifrování. Nevýhodami jsou delší výsledné řetězce, nižší bezpečnost a problém s tím, že šifrování stejného řetězce dává různé výsledky (což může být někdy užitečné).
- Generování z šifrovacího klíče – když uživatel zadá klíč, je z něj vygenerován inicializační vektor.

V modulu je využita druhá z možností, tj. je generován inicializační vektor zároveň s klíčem ze zadaného hesla opět kombinováním výsledků HASH funkce `sha1()`.

Modul se zdá být velmi dobře použitelný zejména pro výrazné zefektivnění práce — implementace pomocí tohoto modulu může výrazně snížit velikost kódu.

4.8 Prezentační vrstva

Prezentační vrstva je rozhraní, přes které komunikuje uživatel s aplikací. Toto rozhraní bývá u webových aplikací nejčastěji grafické, založené na jazyku HTML a internetovém prohlížeči, který je schopen data v tomto jazyce graficky reprezentovat uživateli. V prezentační vrstvě se velmi často objevují velmi podobné či stejné komponenty. Jsou tak přímo předurčeny k využití znovupoužitelnosti, kterou nám poskytuje objektově orientované programování.

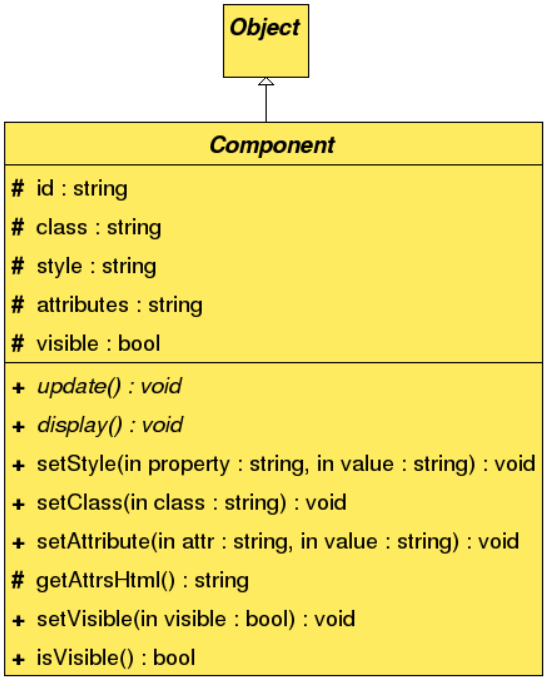
Návrh a implementace prezentační vrstvy v původním vývojovém systému ovšem nejsou zcela vhodné, protože objektového programování nevyužívají plně. Použití šablon, do nichž jsou vkládány objekty, nutí jednak programátora k znalosti jazyka HTML, ale také je obtížně rozšiřitelné a hlavně ztěžuje jednoduše vytvářet dynamické stránky obsahující proměnný počet komponent. Existující komponenty tak, jak jsou navrženy, navíc nevyužívají některých znaků dědičnosti komponent a neposkytují možnost kontroly vstupních dat jinak než jako součást modifikační funkce. To je sice přístup velmi jednoduchý a

přímočarý, ale opět těžko rozšiřitelný. Můžeme tedy tvrdit, že původní implementace neodpovídá současným požadavkům projektu, tedy jednoduché použitelnosti a dobré rozšiřitelnosti.

V podkapitolách toho oddílu budou nejdříve popsány rozdíly mezi prezentačními vrstvami desktopových a webových aplikací a důsledky z těchto rozdílů plynoucí, dále pak komponenty, pomocí nichž budeme vytvářet uživatelské rozhraní, jejich vlastnosti, rozdělení do skupin a konkrétní třídy reprezentující části grafického uživatelského rozhraní.

4.8.1 Struktura stránky a její komponenty

Komunikace mezi aplikací a uživatelem probíhá pomocí dynamicky generovaných internetových stránek, které jsou uživateli zobrazovány prostřednictvím internetového prohlížeče. Každá z těchto stránek je postavena na jazyce HTML popisujícím entity, které se na stránce objevují. Tyto entity mohou být do sebe předem daným způsobem vnořené a na stránce tak jistým způsobem rozmístěné.



Obrázek 4.15: Třída Component v UML

Pro generování HTML kódu těchto entit budeme používat potomky třídy objektů `Component` (viz obrázek 4.15). Tato abstraktní třída umožňuje nastavení základních atributů, jako jsou třída či vlastnosti kaskádových stylů, jež jsou všem entitám společné. Dále nutí programátora implementovat funkce `update()` a `display()`. Funkce `update()` je volána během zpracování požadavku a v ní má reagovat na změnu hodnoty v prohlížeči — využívá se tedy zejména u formulářových položek. Funkce `display()` vrací fragment

HTML kódy reprezentující entitu.

Entity, z nichž se skládá internetová stránka, lze rozdělit do tří skupin, jež mají společné vlastnosti:

- kontejnery, které umožňují zanoření entit,
- vizuální komponenty jako jsou obrázky či text a
- vstupní komponenty, jež se vyskytují zpravidla ve formulářích.

Tyto tři druhy entit a jejich specifické vlastnosti rozebereme v následujících podkapitolách.

4.8.2 Kontejnery

Na internetové stránce (a zejména z jejího HTML kódu) je často patrné, že entity jsou seskupovány dohromady a navzájem zanořeny. Tohoto lze dosáhnout pomocí entit s funkcí kontejneru jako jsou sekce, odstavce apod. Abychom mohli generovat takové fragmenty HTML kódy, potřebujeme objekty s podobnou funkcionalitou, do nichž bude možné vkládat entity a další kontejnery. Tyto požadavky splňuje třída `Container`, která umožňuje kromě operace vkládání položek také jejich odstranění, změnu jejich pořadí a přístup k nim.

Máme tedy třídu zajišťující správu položek. Tu je zapotřebí rozšířit o funkce umožňující generování HTML kódu. K tomu nám slouží třída `Panel`. Ta navíc využívá pro generování fragmentu HTML kódu objekt třídy `Layout`, který může mezi fragmenty kódu jednotlivých entit v kontejneru vkládat dodatečné značky a dosáhnout tak specifického rozložení entit v kontejneru. Typickým příkladem takového rozložení jsou tabulky.

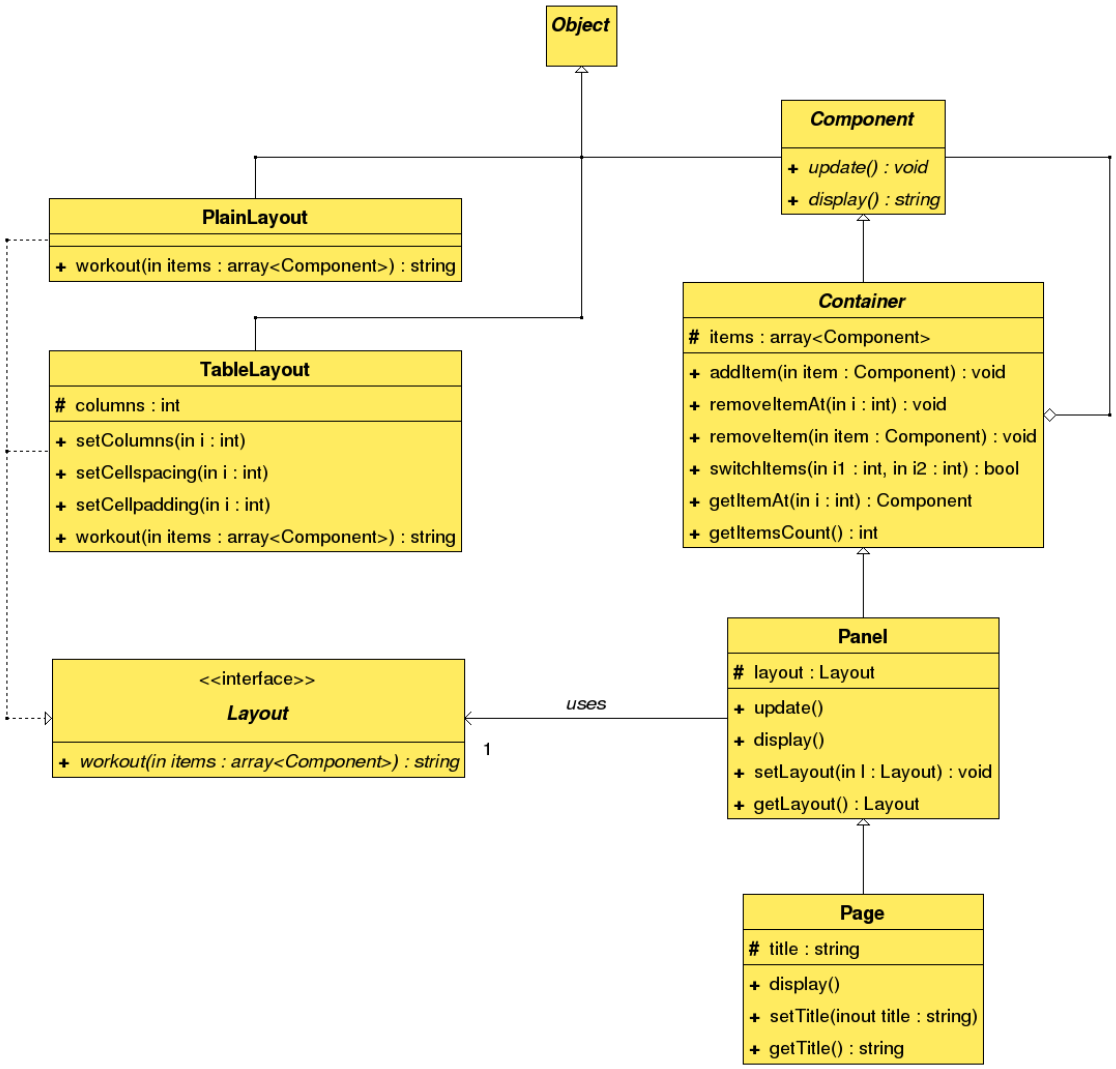
Internetová stránka samotná je také kontejner a je generována pomocí objektu třídy `Page`. Objekty této třídy na rozdíl od třídy `Panel` ještě obsahují název dané stránky, který si od nich během generování výstupu od nich vyžádá aplikace. Aplikace pak obsahuje jednu nebo více takovýchto stránek (v závislosti na složitosti a struktuře aplikace).

4.8.3 Vizuální komponenty

Vizuálními komponentami rozumíme entity, které mají přímo grafický výstup. Jedná se tedy o obrázky a textové popisky.

Textové popisky reprezentované třídou `Label` generují jednoduchý text obalený párovými značkami dle výběru programátora. Vybrat si může z následujících značek:

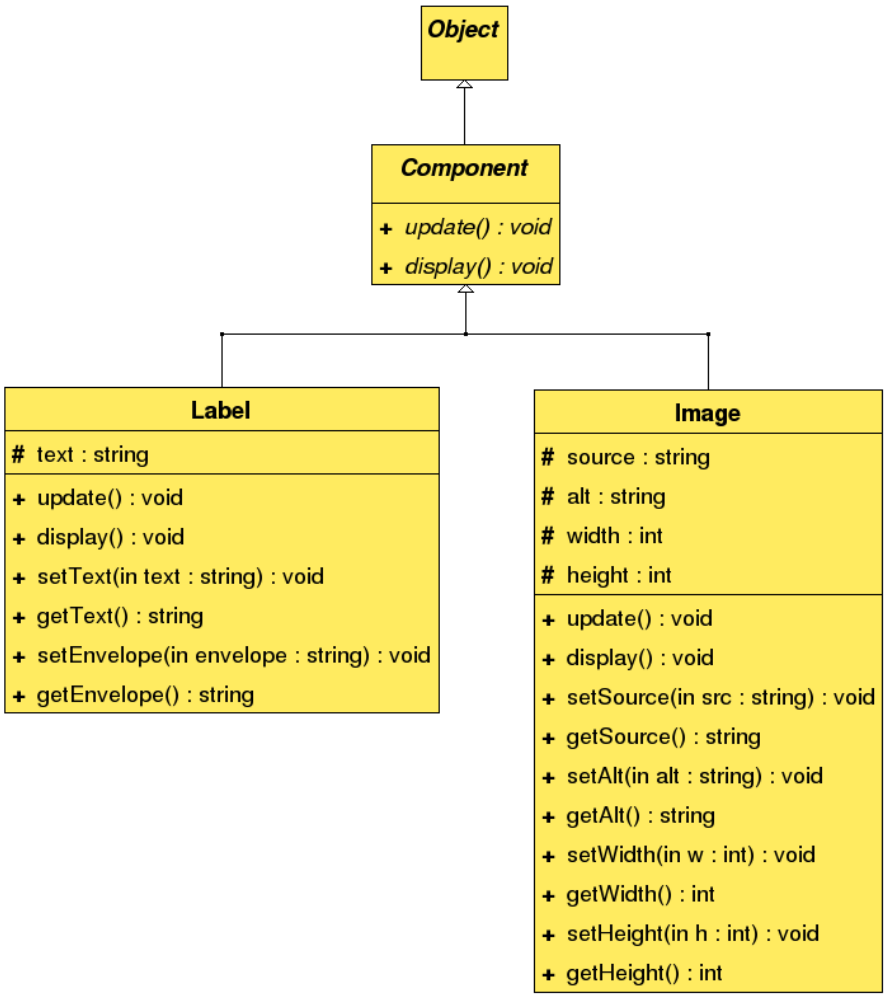
- seskupení <SPAN> ,
- oddíl <DIV>,
- odstavec <P> nebo
- značky pro nadpis <H1> až <H6>.



Obrázek 4.16: Diagram tříd zobrazující vazby a funkcionalitu kontejnerů

Jednoduchým textem myslíme text neobsahující značky jazyka HTML. Tímto omezením je možné jednoduše zabránit vkládání nevalidního kódu a tak i nevalidnosti výsledného dokumentu. Tento způsob je dosti omezující a tak je jedním z cílů při pokračování projektu vytvořit komponentu, zvládající validaci a zobrazení textu se značkami HTML.

Třída `Picture` generující obrázky je navržena tak, aby pokrývala základní atributy HTML značky <IMG>, a tudíž umožňuje nastavení zdroje obrázku, alternativního textu a jeho proporcí. Chce-li programátor obrázek více přizpůsobit, může použít kaskádových stylů k nastavení dalších atributů.



Obrázek 4.17: Diagram tříd vizuálních komponent

4.8.4 Formuláře a vstupní komponenty

Formuláře a jejich komponenty umožňují uživateli internetových stránek zadávat údaje, které jsou v vzápětí posílány serveru. Je zapotřebí upozornit na to, že v desktopových aplikacích běžně neexistují formuláře v takové podobě, jak jsou známé z webových aplikací — tedy formuláře vytvářené pomocí značek `<FORM></FORM>`. Proto ani pomocí vývojového systému nelze explicitně takovéto formuláře vytvářet. Veškeré vstupy jsou řešeny tak, že každá stránka obsahuje vždy jeden velký formulář, který obsahuje všechny komponenty. Vytvářet větší množství formulářů by bylo jen zbytečnou komplikací, protože systém má pouze jeden vstupní bod pro zpracování požadavků (tj. dat z formuláře).

Všechny objekty, které se ve formulářích objevují, dědí třídu `NamedComponent`. Tato abstraktní třída zahrnuje všechny objekty, které mají atribut *name*. Tento atribut se vyskytuje u položek formulářů, protože podle jeho hodnoty se určuje proměnná, která se použije při odesílání jeho hodnoty na server. Tato třída zároveň implementuje rozhraní

`CallbackGenerator` a jeho funkce pro práci se zpětnými voláními. Tím je zajištěno, že každá ze vstupních komponent bude schopna generovat události, resp. provádět zpětná volání na nich závislá.

Třídu `NamedComponent` přímo dědí dvě třídy komponent s konkrétní vizuální podobou. Jedná se o

- *tlačítka* reprezentovaná třídou objektů `Button` a
- *hypertextové odkazy* definované třídou objektů `Anchor`.

Žádná z těchto dvou komponent nepracuje přímo s nějakou hodnotou. Tlačítka a odkazy reagují pouze na akci s nimi provedenou a jsou tak nejčastěji původci událostí typu `ActionEvent`.

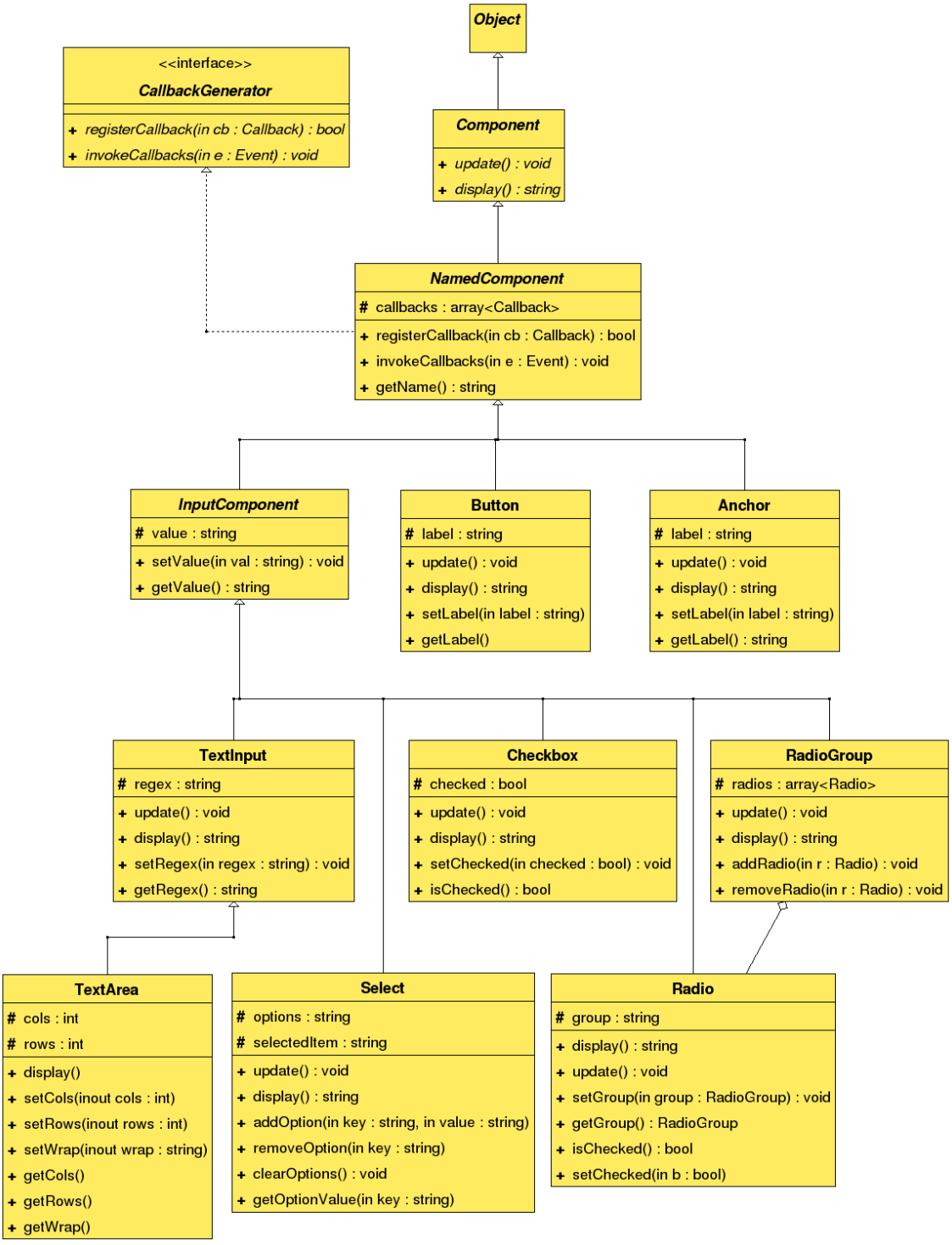
K práci s hodnotou jsou určeni potomci abstraktní třídy `InputComponent`. Tato třída definuje metody pro nastavení a získání hodnoty z daného pole `setValue()` a `getValue()`. Třídy odvozené od této jsou generátory událostí typu `ValueEvent` a také `ErrorEvent` v případě uživatelem zadaných neplatných hodnot. Programátor může při vývoji tedy využít tyto komponenty:

- *textové vstupy* generované pomocí objektů třídy `TextInput`,
- *víceřádková textová pole* vytvářené třídou objektů `TextArea` (ta je specializací třídy `TextInput` k níž přidává atributy určující velikosti vstupního pole),
- *výběry* reprezentované třídou `Select`,
- *zaškrtávací políčka* vytvářená třídou `Checkbox` a
- *výběrová políčka* generovaná třídou `Radio`.

Textové vstupy umožňují kontrolovat řetězec pomocí regulárních výrazů. Slouží k tomu metoda `setRegex()`, která programátorovi umožní nastavit regulární výraz pro dané pole. Pokud poté uživatelem zadaná hodnota regulárnímu výrazu neodpovídá vzniká událost `ErrorEvent`.

Poslední ze jmenovaných — výběrová políčka — jsou seskupeny pomocí objektu třídy `RadioGroup`. Seskupování políček se provádí voláním funkce `setGroup()`, která přijímá jako argument objekt třídy `RadioGroup`. Metoda `setValue()` je pak u políček určena k nastavení hodnoty dané varianty, kdežto u skupiny slouží k výběru správného políčka, které do ní patří. K zjištění, která varianta je vybrána, může programátor použít buď metodu `getValue()` skupiny a nebo testovat, zda je příslušná varianta označena pomocí funkce `isChecked()`.

Objekty třídy `RadioGroup` nemají svůj přímý protějšek v jazyce HTML (políčka se v něm seskupují pomocí atributu `name`), a tak při volání funkce `display()` vrací prázdný řetězec. Význam třídy `RadioGroup` spočívá v synchronizaci políček — v jednu chvíli může být vybráno pouze jedno ze skupiny.



Obrázek 4.18: Diagram tříd vstupních komponent

Všechny uvedené třídy umožňují programátorovi pohodlný vývoj webových aplikací chováním podobných aplikacím desktopovým. Svým rozsahem pokrývají komponenty pouze základní funkčnost a pro plné produkční nasazení je zapotřebí některé z nich ještě rozšířit. Ukázka jejich použití je předmětem další části této práce.

4.9 Ukázková aplikace

V této podkapitole popsán návrh ukázkové aplikace, na níž bude prezentováno, jak lze pomocí vývojového systému vytvářet aplikace. Aplikací je jednoduchý program pro hromadné odesílání elektronické pošty.

Data aplikace budou ukládány v šifrovaných souborech. V souboru `users.conf` budou uloženy informace o uživatelích (tj. přihlašovací jména a hesla), a v souborech pojmenovaných podle vzoru `<login>.dat`[4] se budou v šifrované podobě ukládat adresy, na něž může uživatel odesílat zprávy.

Uživatel s aplikací pracuje podle následujících bodů:

1. Uživatel se přihlásí a zadané informace jsou ověřeny oproti souboru `users.conf`.
2. Uživateli se zobrazí seznam adres, do něhož může přidávat další adresy.
3. V seznamu vybere uživatel adresy, na něž chce zprávu odeslat.
4. Uživatel vyplní předmět, vybere prioritu, napíše zprávu a odešle ji.

Pro práci s aplikací je zapotřebí některých komponent, které je nutné vytvořit děděním. Jedná se zejména o vstupní pole pro zadávání přihlašovacího jména, hesla a e-mailových adres. Všechny jsou odvozeny od třídy `TextInput` pomocí přednastaveného regulárního výrazu.

Na aplikaci je demonstrováno použití komponent pro vytváření prezentační vrstvy a modulů pro práci se soubory, šifrováním a pro odesílání e-mailů. Dále je také prezentována dobrá rozšiřitelnost komponent.

[4] V názvu je řetězec `<login>` nahrazen uživatelovým přihlašovacím jménem

Kapitola 5

Implementace

V této kapitole budou popsány postupy použité během implementace navrženého systému, postřehy, problémy, které se během ní objevily, a popis jejich řešení. Dále budou následovat informace o implementaci jednotlivých částí a popsány změny v modulech, které byly implementovány během semestrálního projektu.

5.1 Použité prostředky

Pro vývoj celého systému byl podle návrhu použit programovací jazyk PHP verze 5. Všechny doposud existující části systému byly přepsány tak, aby mohly plně využívat výhod vycházejících z této verze PHP.

Zdrojový kód byl během vývoje opatřen komentáři ve formátu phpdoc, z nichž lze pomocí nástroje *phpDocumentor* vygenerovat programovou dokumentaci v různých formátech.

Komponenty systému generují stránky obsahující validní XHTML dokumenty. Systém umožňuje využití kaskádových stylů aplikovaných jak přímo do kódu HTML, tak pomocí externích souborů. Aplikace jsou funkční bez použití technologie JavaScript, ale systém umožňuje vkládat externí soubory s JavaScripty do hlaviček stránky a javascriptové atributy k entitám.

5.2 Komponenty

Komponenty byly implementovány podle návrhu — jako jednoduché objekty. Implementované třídy tedy mají konstruktor bez parametrů, privátní atributy s přednastavenými vhodnými implicitními hodnotami a přístupovými funkcemi pro nastavení a získání hodnoty. Pokud je atributem pole, umožňují přístupové funkce tam, kde je to vhodné, přidávání, odebírání a získávání položek.

5.3 Problémy

5.3.1 Externí CSS a ID entit

Během implementace se objevil problém související s použitím kaskádových stylů v externích souborech. Z nich se lze běžně na entity v HTML kódu odkazovat třemi způsoby:

- pomocí typu entity (tedy názvem značky),
- pomocí tříd entit, definovaných pomocí atributu `class` v entitě a
- pomocí ID entity, definovaného pomocí atributu `id` v entitě.

Poslední způsob však není možné u vývojového systému využít, protože identifikátory v souboru jsou statické, kdežto identifikátory entit v HTML kódu jsou generovány dynamicky. Pokud by chtěl programátor vložit hodnotu identifikátoru ručně, mohlo by dojít při vložení dvou více objektů takto pozměněné třídy k vytvoření nevalidní stránky.

Řešením tohoto problému je několik. Prvním z nich je použití kaskádových stylů přímo v atributu `style` v kódu HTML. Tato možnost je přímo podporována komponentami vývojového systému. Další možností je využívat zbylé dva způsoby odkazování na entity (tedy pomocí třídy nebo typu entity). Poslední možností by mohlo být dynamické generování identifikátorů do souboru s kaskádovým stylem. Pro tento způsob by bylo nutné výrazně rozšířit funkcionalitu systému, což není nezbytně nutné.

5.3.2 Odesílání hodnoty zaškrtávacích políček a odkazů

Další problém se objevil během implementace zaškrtávacího políčka. Běžně komponenta se vstupní hodnotou kontroluje v příchozím požadavku

1. zda-li jí byla poslána nějaká hodnota a
2. je-li zaslaná hodnota nová.

Pokud nepřichází komponentou předem určený parametr, je to považováno za stav, kdy komponenta není na aktuální stránce. Zaškrtávací políčko ovšem pracuje tak, že pokud je zaškrtnuto, posílá nějakou hodnotu, a pokud zaškrtnuto není, parametr neodešle. Vzniká tak efekt, že pokud uživatel políčko označí, vše funguje správně, pokud jej ale odznačí, nic se nestane.

Tento problém byl vyřešen přidáním skrytého pole ke každému zaškrtávacímu políčku, které indikuje, zda-li je políčko na dané stránce, a přítomnost či nepřítomnost hodnoty políčka pak indikuje jeho stav.

Podobný problém vyvstal i u odkazů, které samy o sobě žádnou hodnotu neposílají. Proto je hodnota odesílána na server pomocí parametru umístěného v adrese požadavku (tedy metodou GET).

5.3.3 Zámkové soubory

Během práce se soubory pomocí třídy `File` vytváří systém tzv. zámkové soubory, do nichž ukládá metadata o souborech otevřených na úrovni frameworku. Problém vzniká ve chvíli, kdy uživatel v aplikaci soubor otevře a následně přestane s aplikací pracovat. Systém nemá jak poznat, že uživatel ukončil svou práci, pokud o tom uživatel nezašle explicitní informaci. Takto může docházet jednak k hromadění souborů na disku serveru jednak k jevu, kdy budou soubory, s nimiž se nepracuje, uzamčeny.

Problém je řešitelný dvěma způsoby, z nichž oba mají své výhody i nevýhody.

1. Neplatné zámky můžeme odstranit pravidelným spouštěním skriptu pomocí plánovače úloh (cron atp.). Tento způsob zajistí i odstranění zámků souborů, které již neexistují, a udržuje zámky obecně v lepším stavu. Nevýhodou této metody je závislost na plánovači úloh a nutnost tento plánovač nastavit.

2. Platnost zámků můžeme také kontrolovat vždy při, když přistupujeme k souboru pomocí frameworku (nebo některou jeho součástí). Nevýhodou tohoto systému je, že pokud je soubor, k němuž zámek patří, odstraněn, nedojde k odstranění tohoto zámku. Výhodou je, že programátor nemusí systém nijak nastavovat.

Vzhledem k specifikovaným požadavkům, kdy chceme, aby k provozu vývojového systému stačil pouze webový server s možností skriptování pomocí jazyka PHP, byla implementována metoda druhá.

5.4 Struktura souborů

Během implementace systému bylo vytvořeno několik desítek tříd. Kvůli přehlednosti byly zavedeny následující konvence:

1. Implementace každé třídy je umístěna do samostatného souboru.

2. V kořenovém adresáři je umístěn pouze jediný soubor — `index.php` — který zajišťuje spuštění frameworku.

3. Všechny soubory vývojového systému jsou umístěny v adresáři `include/clode/`, v němž je vytvořena následující struktura:

   - soubor `clode.php` je skript, s jehož pomocí lze připojit vývojový systém do kteréhokoliv dalšího skriptu,
   - soubor `defines.php` obsahuje definice některých globálních konstant a nastavení,
   - soubor `includes.php` obsahuje skript pro připojení všech důležitých souborů (jádro, události výjimky…),

- adresář `core` obsahuje třídy jádra vývojového systému,
- adresář `events` obsahuje třídy událostí,
- adresář `exceptions` obsahuje třídy pro výjimky,
- adresář `intercom` obsahuje třídy objektů, které se užívají při komunikaci mezi frameworkem a aplikací,
- adresář `view` obsahuje třídy pro tvorbu prezentační vrstvy,
- adresář `wrappers` obsahuje třídy pro snadnější vývoj modelu aplikace.

4. V domovském adresáři je umístěn adresář `config`, v němž je skript `clode_init.php`, který je zpracován při prvním spuštění frameworku.

5.5 Ukázková aplikace

Ukázková aplikace je implementována podle návrhu a je možné na ní demonstrovat použití vybraných komponent pro vytváření prezentační vrstvy a modulů pro práci se soubory, šifrováním a pro odesílání e-mailů.

Dále je také demonstrováno použití architektury MVC, kdy aplikace zpracovává zpětná volání a mění při tom model a pohled. Prezentační vrstva je rozdělena do dvou stránek — stránku pro přihlášení a stránku pro správu příjemců a odeslání e-mailu. Model obsahuje soubory pro autentizaci uživatele a jeho seznam adresátů, objekt šifrování a dešifrování a objekt elektronické zprávy, která se bude odesílat.

Kapitola 6

Závěr

Nejvýznamnějším přínosem projektu je automatické ukládání stavu bez zásahu programátora, které zmenšuje propast mezi desktopovými a webovými aplikacemi. Objektový přístup a změna stavu aplikace pomocí callbackových funkcí navíc dovoluje implementovat aplikaci podle návrhového vzoru MVC a oddělit tak prezentační a datovou vrstvu. Implementace datové vrstvy je podporována automatickým ukládáním stavu a objekty ulehčující použití některých funkcí, prezentační vrstva pak množstvím objektů, které generují validní HTML kód. Vývojový systém tedy umožňuje programátorovi efektivnější přístup k vývoji webových aplikací, zjednodušuje jejich správu či případnou pozdější modifikaci a je koncipován tak, aby mohl být dále rozvíjen a rozšiřován.

Během semestrální práce byly vytvořeny moduly pro logování, šifrování a odesílání elektronické pošty. V průběhu diplomové práce bylo pozměněno jádro systému, celý systém byl převeden pod PHP 5, byla přepracována prezentační vrstva tak, aby komunikovala s aplikací pomocí událostí, a přidán modul pro práci se soubory.

6.1 Pokračování projektu

Vývojový systém v této podobě je sice použitelný, ale obsahuje pouze základní komponenty. Během pokračování projektu by tedy mělo dojít k rozšíření funkčnosti jednotlivých komponent.

Systém nyní využívá k uchovávání stavu tzv. sessions, což se zdá být nejlepší volbou v nedistribuovaném prostředí. Zvýšení výkonnosti by mohlo být provedeno pomocí rozložení zátěže, které by muselo být podpořeno napojením na databázi. V tomto případě by bylo zapotřebí, aby se o serializaci staraly objekty samy. Další zvyšování výkonu by bylo možné dosáhnou cachováním některých komponent na prezentační vrstvě.

Vývojový systém nyní neumožňuje lokalizaci, což by pro nasazení do většího produkčního prostředí mohlo být překážkou. Bylo by tedy vhodné navrhnout a na systém napojit řešení, které by umožňovalo zobrazení obsahu ve více jazycích.

Na prezentační vrstvě chybí komponenta, která by byla schopna vytvářet na výstup

formátovaný HTML text. Tento problém by mohla řešit komponenta rozšiřující třídu `Label` tak, že by buď prováděla validaci HTML v ní, nebo by užívala alternativní formátování (např. wiki formátování apod.).

Pro usnadnění práce programátora, by měla existovat další komponenty, např pro práci s databázemi, zpracování XML apod. Modul pro logování by mohl být napojen na systémový log (na unixových serverech).

Na implementaci ukázkové aplikace je patrné, že programátor při vývoji musí udělat relativně hodně kroků, které by bylo možné automatizovat pomocí grafického návrhového studia, které by umožňovalo vytvořit prezentační vrstvu.

Literatura

[1] Oprštěný, M.: PHP Modular Object Framework. Diplomová práce, Brno, Vysoké učení technické v Brně, Fakulta informačních technologií, 2006.

[2] Rybák, A.: Balíček modulů pro tvorbu webových aplikací pomocí PHP. Semestrální práce, Brno, Vysoké učení technické v Brně, Fakulta informačních technologií, 2006.

[3] Darie, C., Brinzarea, B., Chereches-Tosa, F., Bucica, M.: AJAX a PHP - tvoříme interaktivní webové aplikace profesionálně. Zoner Press 2006.

[4] Baray, C.: The Model–View–Controller Design Pattern. Indiana University 1999. URL: http://www.cs.indiana.edu/~cbaray/projects/mvc.html