Model-View-Controller vs. Presentation-Abstraction-Control

Antonín Daněk
Fakulta elektrotechnická
České vysoké učení technické
Praha, Česká Republika
danekant@fel.cvut.cz

Jakub Stejskal
Fakulta elektrotechnická
České vysoké učení technické
Praha, Česká Republika
stejsj11@fel.cvut.cz

***Abstract*—Článek se zabývá dvěma podobnými architektonickými vzory, jejichž primární úkol je oddělení doménového modelu od uživatelského rozhraní. První část článku se věnuje architektonickému vzoru Model-View-Controller, načež navazuje kapitola o vzoru Presentation-Abstraction-Control. Po uvedení obou vzorů následuje jejich srovnání a článek je zakončen zhodnocením poznatků v závěru.**

***Keywords*—Klíčová slova: MVC, PAC, UI**

I. ÚVOD

Při vývoji interaktivních aplikací s bohatými uživatelskými rozhraními existuje ze strany vývojářů dlouhodobě požadavek na možnost rozdělení vývoje aplikace mezi různě zaměřené odborníky (či celé týmy), stejně jako na možnost tyto dílčí celky samostatně udržovat, modifikovat a testovat.

Kvalitní interaktivní aplikace se skládá ze třech prvků - robustního datového modelu s dobře definovanými pravidly pro doménové objekty, optimalizované aplikační logiky a snadno použitelného uživatelského rozhraní. Protože je nemožné vyžadovat po jednom člověku, aby byl databázovým specialistou, aplikačním programátorem, odborníkem na uživatelská rozhraní, kodérem a grafikem v jedné osobě (ačkoliv se o to mnozí menší živnostníci pokoušejí), je třeba práci rozdělit mezi jednotlivé specialisty.

Především z těchto důvodů, spolu se snahou o zlepšení přehlednosti implementace aplikace a podporou principů OOP[1], bylo navrženo hned několik návrhových vzorů, které tento problém poměrně úspěšně řeší. V článku se zaměřujeme na velmi známý architektonický vzor Model-View-Controller a méně známý Presentation-Abstraction-Control. Podíváme se na princip obou vzorů a pokusíme se je srovnat.

II. MODEL-VIEW-CONTROLLER

Architektonický vzor Model-View-Controller, neboli MVC, dělí aplikaci do nezávislých komponent tří typů: Model, View a Controller. Toto rozdělení má za úkol oddělit od sebe funkční celky z hlediska uživatelského rozhraní: business logiku, výstup aplikace a vstup aplikace s aplikační logikou.

[1]Object-oriented programming - Objektově orientované programování

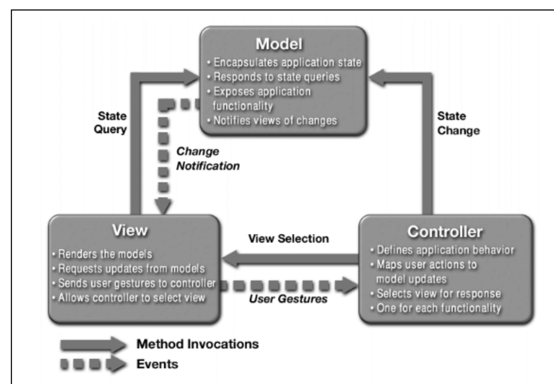

Figure 1. Schéma komponent MVC - zdroj: [1]

A. *Komponenty MVC*

*1) **Model**:* Komponenta Model zapouzdřuje doménovou (business) logiku, stav a datový model aplikace. Je nutné zdůraznit, že kromě doménových objektů a dat tato komponenta obsahuje i vztahy mezi těmito objekty a jejich business pravidla, tedy různé validace (např. povinné položky) či omezení (např. pravomoci uživatele). Komponenta typu Model je zcela nezávislá na ostatních komponentách a bývá v každé aplikaci pouze jedna, ač je vnitřně zpravidla realizována množinou doménových objektů.

*2) **View**:* O získání výstupních dat z Modelu a jejich interpretaci do podoby vhodné k interaktivní prezentaci uživateli se starají komponenty typu View. Kromě schopnosti zobrazování dat může View obsahovat i určitou prezentační logiku, např. filtrování či třídění zobrazovaných dat. Protože Model je na View nezávislý, View samotný musí být schopen získávat data z Modelu a potřebuje tedy do určité míry znát jeho strukturu, což jej činí na Modelu závislým. Komponent typu View bývá v aplikaci často více. Každá z nich pak reprezentuje odlišný pohled na stejná data - např. text, tabulka a graf. Přepínání mezi těmito pohledy patří do kompetence komponenty typu Controller.

*3) **Controller**:* Obsluhu a zpracování vstupů uživatele spravují komponenty typu Controller. Za základě uživatelských událostí Controller mění data v Modelu a přepíná mezi různými View. Každý Controller je tedy závislý

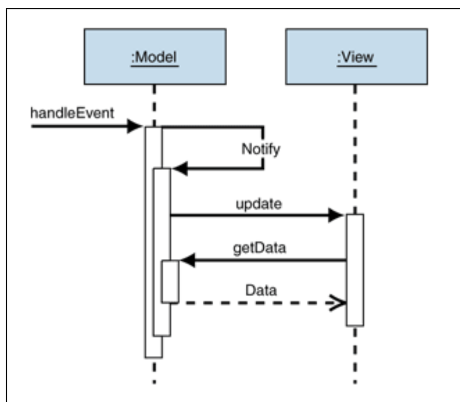

Figure 2. Sekvenční diagram aktivního MVC - zdroj: [2]

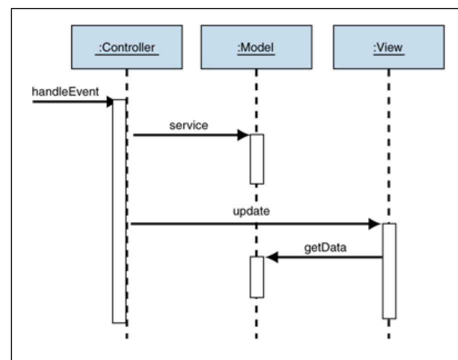

Figure 3. Sekvenční diagram pasivního MVC - zdroj: [2]

jak na Modelu, tak na všech souvisejících View. Na každou funkcionalitu aplikace by měl připadat jeden Controller. Počet a struktura Controller komponent v aplikacích se různí. Často celou aplikaci obsluhuje jeden Controller nebo jsou Controllery hierarchicky uspořádány (např. ve vzoru Front Controller).

B. Příklady použití vzoru MVC

*1) **Příklad - sálové počítače**:* Původ architektury MVC sahá do 70. let, tedy do dob sálových počítačů. Uživatelský vstup, typicky klávesnice, byl značně nezávislý na výstupu, sestávajícího z textového monitoru. Kód ovládající výstup tak byl naprosto přirozeně oddělen od kódu ovládajícího vstup. Návrh MVC tedy vyplynul ze stavby hardwaru.

S příchodem moderních operačních systémů a jejich ovládání pomocí myši se tato přímá spojitost vytratila a vzor MVC musel být často poněkud ohýbán.

*2) **Příklad - Web**:* Jistou renesanci prožil vzor MVC s příchodem webových aplikací. I zde je patrné jasné oddělení vstupu (HTTP požadavek) a výstupu (odpověď na tento požadavek například v podobě HTML stránky). Pokud si za příklad vezmeme J2E aplikaci, můžeme snadno jednotlivé MVC komponenty namapovat na EJB (Model), servlety (Controller) a JSP s JavaScriptem (View).

I zde ovšem při bližším ohledání najdeme odchylky od původní koncepce MVC. Prvním problémem je Client-Server charakter samotného HTTP protokolu - Model (tedy serverová část aplikace) nemá žádné prostředky, jak informovat View (klientův prohlížeč) o své změně. Řešením jsou AJAXové technologie, které pravidelným dotazováním na server simulují obousměrnou komunikaci.

Dalším prohřeškem proti vzoru MVC je způsob, jakým v J2E aplikaci proudí data. Servlety získávají data z doménového modelu a předávají je do JSP stránek. View tedy nedotazuje přímo Model, ale komunikuje skrze Controller. Stejný rys můžeme zaznamenat i u většiny webových frameworků opatřených nálepkou MVC.

C. Varianty MVC a časté odchylky

Architektonický vzor MVC je velice rozšířený a populární napříč technologiemi, platformami a typy aplikací. Jak již bylo demonstrováno na příkladu webových aplikací, každá technologie si pak vzor přizpůsobuje svým potřebám a možnostem, čímž vzniká velká rozmanitost variant a odchylek. Mnoho odborníků i "odborníků" má pak tendenci stavět se do pozice arbitrů určujících, co je a co není možno považovat za MVC architekturu. Pro ilustraci zde uvádíme několik dalších běžných variací vzoru.

*1) **Aktivní vs. Pasivní**:* Rozdíl mezi aktivním a pasivním MVC spočívá v tom, kdo upozorňuje View na změnu Modelu a tedy na nutnost obnovení dat zobrazovaných daným View. V aktivní variantě je to sám Model. Protože však Model nesmí být závislý na View, probíhá tato komunikace nepřímo, což je zpravidla realizováno návrhovým vzorem Observer. V pasivní variantě je View na změnu upozorněn Controllerem, který danou změnu vyvolal.

*2) **Anemický doménový model**:* Některé technologie dávají možnost validovat uživatelské vstupy již ve View nebo Controlleru, což svádí k definici business pravidel mimo Model a redukci Modelu na datové objekty. To může vést k duplikaci kódu, protože tato pravidla typicky sdílí více View zobrazujících stejná data. Tento jev se nazývá anemický doménový model.

*3) **Model-View-Presenter**:* V některých typech aplikacích splývá funkcionalita View a Controlleru. Typicky je tomu tak v desktopém prostředí, kde se UI skládá z widgetů (tlačítek, posuvníků, polí apod.), které mají zabudovanou obsluhu svých událostí přímo v sobě. V takovém případě přechází zodpovědnost za zpracování vstupu na View. Tato architektura se někdy nazývá Model-View-Presenter, kde Presenter je označení pro Controller ochuzený o obsluhu vstupu. [3]

D. Výhody a nevýhody vzoru MVC

*1) **Výhody vzoru MVC**:*

- Rozdělením aplikace do tří typů komponent s jasně danými zodpovědnostmi vzniká snadno pochopitelná a udržovatelná aplikace.
- Protože jednotlivé View sdílí společný Model, lze je jednoduše nahrazovat, upravovat či přidávat a to jak při vývoji, tak i v rámci funkcionality aplikace.
- Díky jednoduché struktuře a přímé vazby View na Model má aplikace nízkou režii, což má kladný vliv na její výkon.

- MVC je velice populární architektura a díky tomu je důkladně popsána a prověřena.

2) Nevýhody vzoru MVC:

- Pro svou jednoduchou strukturu není vzor MVC vhodný pro komplexnější a rozsáhlejší aplikace.
- Popularita tohoto vzoru má i stinné stránky. Z pojmu MVC se stává obchodní značka a kolem vzoru samotného vzniká mnoho zmatení a nedorozumění.
- Častá snaha použít vzor i navzdory použitým technologiím a specifikům aplikace vede ke kontraproduktivitě.

III. Presentation-Abstraction-Control

Architektonický vzor Presentation-Abstraction-Control, jehož akronym je PAC, je tvořen hierarchií spolupracujících agentů (viz. obrázek 5). Každý z těchto agentů je tvořen třemi komponentami: *Presentation*, *Abstraction* a *Control*. Hlavním účelem tohoto vzoru (stejně jako vzoru MVC) je oddělení uživatelského rozhraní od aplikační logiky a datového modelu. PAC tedy definuje strukturu pro interaktivní softwarové systémy. [4]

A. Komponenty PAC agentů

*1) **Presentation**:* Komponenta *Presentation* představuje uživatelské rozhraní navrhovaného systému, přičemž obsluhuje výstupy, ale i vstupy systému. Důležité však je, že tato komponenta kromě zmíněného neobsahuje žádnou další logiku a nedisponuje ani znalostí modelu (Abstraction) aplikace. Presentation komunikuje výhradně a pouze s Control. Pokud chce Abstraction nabídnout Presentation nové informace, musí tak učinit prostřednictvím Control.

Pod touto komponentou si lze představit např. šablonovací systém, který přijme syrová data od Control a pouze je vloží do uživatelského rozhraní, kterým může být pro představu např. XHTML/CSS stránka.

*2) **Abstraction**:* Rozhraní pro datový model reprezentuje *Abstraction*, který tato komponenta spravuje a zároveň do něj přistupuje. Abstraction tedy zajišťuje přístup k datům. Pokud se však nebude jednat o Abstraction komponentu Top-level agenta (více o Top-level agentech v sekci III-B1), pravděpodobně nepůjde o globální data aplikace. Spíše je vhodné si pod Abstraction představit místo pro uchování aktuálního stavu agenta.

Stejně jako Presentation i tato komponenta komunikuje pouze s Control a komunikace s jinými komponentami či agenty pak probíhá opět prostřednictvím Control.

*3) **Control**:* Aplikační logiku v systému navrženém podle vzoru Presentation-Abstraction-Control bychom nalezli v komponentě *Control*.

Jak již bylo nastíněno při popisu Presentation a Abstraction, Control mezi zmíněnými komponentami zprostředkovává komunikaci a zajišťuje tak důsledné oddělení uživatelského rozhraní od datového modelu. Kromě toho navíc Control komunikuje s ostatními agenty, konkrétně opět pouze s jejich Control komponentami.

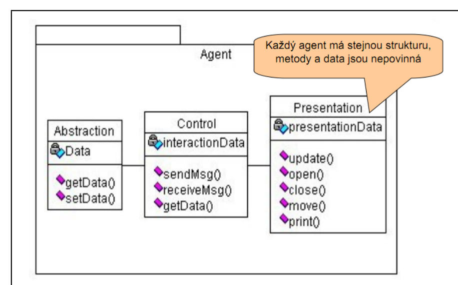

Figure 4. UML reprezentace PAC agenta - zdroj: [5]

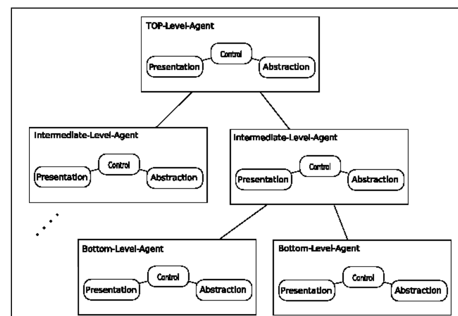

Figure 5. Vizualizace agentů architektonického vzoru PAC - zdroj: [6]

Jak je možné vidět na obrázku 4, komponenty Control často plní roli přeposílatelů zpráv a to především u Intermediate-level agentů (více o Intermediate-level agentech v sekci III-B2).

B. Typy agentů a jejich funkce

Výše zmíněné tři komponenty jsou součástí každého (až na výjimky) PAC agenta, přičemž každý agent řeší specifický aspekt funkcionality aplikace. Právě díky možnosti rozdělit funkce aplikace do jednotlivých agentů a navíc oddělení jednotlivých vrstev do PAC komponent, je možné systém dekomponovat jak horizontálně, tak vertikálně.

Jak je vidět na obrázku 5, architektonický vzor PAC má tři druhy agentů, které dohromady tvoří stromovou strukturu. Z toho vyplývá, že PAC systém by měl obsahovat pouze jednoho Top-level agenta, více Intermediate-level agentů a nejvíce Bottom-level agentů.

Agenti níže (dále od kořene stromu) jsou závislí na agentech výše, až k Top-level agentovi, což z tohoto agenta dělá nejdůležitější součást systému. Je dobré si uvědomit, že PAC agent může být prvek o velikosti pouze jedné třídy, ale i o velikosti celého podsystému.

Pro všechny PAC agenty je kromě komunikace pouze skrze komponenty Control charakteristická také schopnost udržovat si vlastní stav.

*1) **Top-level**:* Příkladem Top-level agenta je *Data repository*. Tento agent často obsahuje jádro systému a poskytuje služby ostatním agentům, kteří se bez něj proto neobejdou.

Vhodnou otázkou potom je, co by mohla u tohoto agenta obsahovat komponenta Presentation. Není vyloučené, že u některých agentů jedna vrstva chybí a typicky právě Top-level agent často neobsahuje Presentation, pokud plní čistě úlohu již

zmíněného jádra systému a rozhraní pro přístup ke globálním datům.

V mnoha případech však i Top-level agent Presentation komponentu má. Ta potom obsahuje ty prvky UI[2], které nejsou specifické pro jednoho agenta. Pokud bychom je nedefinovali v Top-level agentovi, museli bychom je v každém nižším agentovi zavádět znovu, což by nutně do systému zavádělo duplicitní kód. Pro představu se může jednat např. o dialogy nebo hlavní menu.

*2) **Intermediate-level**:* Agenti této úrovně obvykle koordinují komunikaci mezi ostatními agenty, proto je dobrým příkladem Intermediate-level agenta *View coordinator*.

Intermediate-level agenti řídí agenty nižší úrovně a poskytují nad nimi vyšší míru abstrakce. Typyckým případem užití je pak správa více pohledů na ta samá data. Úlohy jednotlivých komponent v takovém případě vypadají takto:

- Presentation komponenta nabízí ovládací prvky pro otevření různých pohledů - např. graf a tabulku.
- Abstraction vrstva udržuje informace o tom, které z těchto pohledů jsou otevřené.
- Control přeposílá zprávy o změnách na datech - pokud jeden pohled změní data, ostatní jsou o tom informování a mají možnost na změnu reagovat (obvykle překreslením). Kromě toho Control také ovládá funkcionalitu zavírání a otevírání pohledů.

```
public class IntermediateLevelAgent {

  private TopLevelAgent topAgent;
  private IntermediateLevelAgent intermediateAgent;
  private BottomLevelAgent bottomAgent;

  public TopLevelAgent getTopAgent() {
    return topAgent;
  }
  public void setTopAgent(TopLevelAgent agent) {
    topAgent = agent;
  }
  public IntermediateLevelAgent getIntermediateAgent(){
    return intermediateAgent;
  }
  public void setIntermediateAgent(
      IntermediateLevelAgent agent ) {
    intermediateAgent = agent;
  }
  public BottomLevelAgent getBottomAgent() {
    return bottomAgent;
  }
  public void setBottomAgent( BottomLevelAgent agent ) {
    bottomAgent = agent;
  }
}
```

Listing 1. Jednoduchá implementace PAC IntermediateLevel agenta - zdroj [5]

*3) **Bottom-level**:* Agenti nejnižší úrovně se starají o konkrétní sémantický koncept aplikace. Jinak řečeno, implementují určitou funkcionalitu. Příkladem Bottom-level agenta může být *Spreadsheet* nebo *PieChart*.

Jeden Bottom-level agent pak může například vykreslovat graf, přičemž se stará i o všechny přidružené operace, jako by mohl být např. zoom či filtrování některých dat.

[2]User Interface nebo-li uživatelské rozhraní

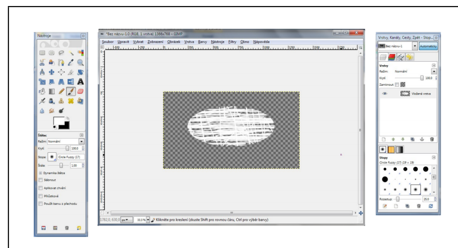

Figure 6. Gimp - software využívající architektonického vzor PAC - zdroj: [5]

C. Příklady použití vzoru PAC

*1) **Praktický příklad - Gimp**:* Jeden z nejpopulárnější příkladů použití PAC je grafický nástroj Gimp, který se snaží nabízet podobné možnosti, jako jeho vyspělejší konkurence Adobe Photoshop. Uživatelské rozhraní programu je vidět na obrázku 6.

Každé okno programu (nástroje, vrstvy, kreslící plátno, ...) plní samostatnou úlohu, ale zároveň se změny v jednom okně okamžitě projevují i ve všech ostatních. [5]

*2) **Teoretický příklad - volby**:* Volby probíhají na úrovni měst. Výsledky jsou však zobrazovány nejen na úrovni měst, ale také na úrovni krajů a celého národa.

V tomto případě Bottom-level agenti obsahují data na úrovni měst, Intermediate-level agenti na úrovni krajů a Top-level agent na úrovni celé země. Každý agent nižší úrovně poskytuje pohled na svá data a zároveň je předává výše pro globálnější pohled.

*3) **Teoretický příklad - řízení letového provozu**:* Řízení letového provozu využívá monitor pro sledování vzdušného provozu a zároveň monitoruje letadla na přistávací dráze nebo např. počasí.

Systém bude mít tedy několik vstupů, přitom každý bude zprostředkovávat jiný PAC agent. Jeden PAC agent přijme informaci z radaru o přilétávajícím letadle a použije svou Presentation komponentu k vykreslení ikony letadla na danou pozici na obrazovce. Druhý PAC agent přijme informaci o jiném, odlétávajícím letadle a také zakreslí tuto informaci na obrazovku. Třetí PAC agent přijme informaci o počasí a vykreslí mraky.

D. Varianty PAC

*1) **Hierarchical MVC**:* Přestože byl tento vzor popsán nezávisle na vzoru PAC, je možné jej chápat jako jeho obměnu. HMVC zachovává hierarchickou strukturu agentů vzoru PAC, ale vnitřní struktura agentů vychází ze vzoru MVC. Každý agent tedy obsahuje komponentu Model, komponentu View (která se přímo dotazuje Modelu na data) a komponentu Controller, která přirozeně přijímá roli prostředníka mezi jednotlivými agenty.

E. Výhody a nevýhody vzoru PAC

*1) **Výhody vzoru PAC**:*

- Odděluje aplikační logiku od uživatelského rozhraní.
- Má díky komunikaci pouze skrze komponentu Control nízký coupling.

- Především z předchozího důvodu je systém dobře upravovatelný. Agenta lze snadno nahradit jiným agentem. Nový agent musí pouze dodržet rozhraní vyžadované v Control.
- Systém využívající PAC je snadno rozšiřitelný. Lze jednoduše přidat další agenty, bez nutnosti provádění změn po celé aplikaci.
- Podpora multitaskingu. Každý agent může běžet ve svém vlákně.
- Podpora distribuovaného výpočtu. Komunikaci mezi agenty můžeme realizovat pomocí RPC[3] a každý agent pak může běžet na jiném fyzickém stroji.
- PAC je vhodný pro tvorbu aplikací se složitými uživatelskými rozhraními. Jednotlivé části UI lze realizovat separátními agenty a systém tak navrhnout bez porušení zapouzdření.

*2) **Nevýhody vzoru PAC:***

- Systém navržený podle vzoru PAC bude složitější a tedy také složitější na pochopení pro nové vývojáře.
- S větší složitostí souvisí i nižší rychlost. Komunikace agentů má určitou režii.
- Riziko rozpadu systému do příliš velkého množství agentů, pro které je pak navíc nutné dopsat další, koordinující agenty. To opět přispívá k větší nepřehlednosti a nižšímu výkonu.

IV. Srovnání vzorů MVC a PAC

V tabulce I je uvedeno základní srovnání obou vzorů. Jak je vidět, MVC komponenta Model přibližně odpovídá PAC komponentě Abstraction. Nelze však říci, že MVC View odpovídá PAC Presentation, či že MVC Controller odpovídá PAC Control. Můžeme pouze prohlásit, že kombinace MVC View s Controller řeší v podstatě to samé, co PAC Presentation s Control.

Hlavní rozdíl mezi PAC a MVC komponentami spočívá v tom, jakým způsobem se získávají data z doménového modelu do uživatelského rozhraní. Zatímco u vzoru MVC komponenta View disponuje určitou logikou a především znalostí modelu, pročež se dotazuje na data přímo, PAC Presentation je zcela bez logiky a lze jej spíše považovat za šablonu, do které jsou pouze vložena příslušná data komponentou Control.

Dalším důležitým rozdílem je pak hierarchická struktura vzoru PAC. Proto zatímco jeden konkrétní pohled do aplikace by v případě MVC měl být realizován pouze jedním View a příslušným Controllerem, v případě PAC jsou obvykle různé části jediného pohledu složeny několika agenty a tudíž je aktivních více komponent najednou.

PAC má oproti MVC o něco menší coupling a to díky komunikaci komponent i jednotlivých agentů pouze skrze komponentu Control. Také je díky možnosti rozdělení jediného pohledu do aplikace mezi více agentů (widgety) vhodnější pro aplikace s velmi složitým uživatelským rozhraním.

Protože má MVC jednodušší strukturu, může nabídnout i vyšší rychlost než PAC, u kterého může navíc snadněji dojít k rozpadu do více komponent, než je ve skutečnosti potřeba.

[3]Remote procedure call - vzdálené volání procedur

MVC	**PAC**
Model	Abstraction
View + Controller	Presentation + Control
View obsluhuje výstup, Controller obsluhuje vstup	Presentation obsluhuje vstup i výstup
View má určitou logiku (znalost modelu)	Presentation je zcela bez logiky (šablonovací systém)
View přijímá data přímo z Modelu	Presenter přijímá data z Abstraction přes Control
	hierarchická architektura

Table I
MVC vs. PAC - srovnání komponent

Mezi společné vlastnosti obou architektonických vzorů patří:

- Oddělení doménového modelu od uživatelského rozhraní.
- Snadná rozšiřitelnost, modifikovatelnost a testovatelnost.
- Přirozené rozdělení aplikace do funkčních jednotek z hlediska UI.

V. Závěr

V této práci jsme se pokusili přiblížit problematiku a úskalí všeobecně známého, avšak nikoli všeobecně pochopeného, architektonického vzoru Model-View-Controller a dále jsme představili méně rozšířený, příbuzný vzor Presentation-Abstract-Control.

Při srovnání obou vzorů a popisu některých jejich variant jsme ukázali, že ani v oboru softwarové architektury neexistuje mnoho absolutních pravd, a že je při návrhu vždy nutné provádět rozhodnutí na základě zkoumání širšího kontextu, jako jsou například použité technologie, jejich vlastnosti a omezení či specifika vyvíjené aplikace a její funkční i nefunkční požadavky.

References

[1] Java SE Application Design With MVC, 16.03.2011, http://blogs.sun.com/JavaFundamentals/entry/java_se_application_design_with
[2] Model-View-Controller, 16.03.2011, http://msdn.microsoft.com/en-us/library/ff649643.aspx
[3] Úvod do architektury MVC, 16.03.2011 http://zdrojak.root.cz/clanky/uvod-do-architektury-mvc/
[4] Presentation-Abstraction-Control (PAC) Pattern, 16.03.2011, http://moosehead.cis.umassd.edu/cis390/slides/08_PAC.pdf
[5] Presentation Abstraction Control, 16.03.2011, http://icpc.felk.cvut.cz/web/slides/y36ass/Prezentace/
[6] Wikipedia - Presentation-abstraction-control, 16.03.2011, http://en.wikipedia.org/wiki/Presentation-abstraction-control
[7] MVC vs. PAC, 16.03.2011, http://www.garfieldtech.com/blog/mvc-vs-pac
[8] Developing Mobile GUIs, 16.03.2011, http://assets.cambridge.org/97805218/17332/sample/9780521817332ws.pdf
[9] Software Architecture, 16.03.2011, http://www.cs.ucy.ac.cy/courses/EPL603/Intro4.pdf
[10] Examples to Accompany: Design Patterns, 16.03.2011, http://kirankawalli.110mb.com/PatternExamples.pdf
[11] Patrick Donohoe, Software architecture: WICSA1 1999, San Antonio, Texas, USA