VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

POROVNÁNÍ SOFTWAROVÝCH ARCHITEKTUR

SOFTWARE ARCHITECTURE COMPARISON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MICHAL MOTYČKA**
AUTHOR

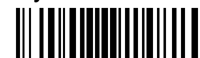
VEDOUCÍ PRÁCE **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce



Student: **Motyčka Michal**
Program: Informační technologie
Název: **Porovnání softwarových architektur**
Software Architecture Comparison
Kategorie: Softwarové inženýrství

Zadání:

1. Nastudujte běžně používané typy softwarových architektur, např. Transaction script, Onion, Hexagon architektury.
2. Stanovte požadavky na aplikace, na kterých demonstrujete vybrané architektury. Po konzultaci s vedoucím, proveďte návrh demonstrační aplikace ve zvolených architekturách.
3. Navržené aplikace implementujte ve všech zvolených architekturách.
4. Dle pokynů vedoucího stanovte objektivní metriky a proveďte porovnání architektur na základě zkušeností získaných z bodu 2 a 3.

Literatura:

1. Fowler, M. (2002). *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc..
2. Martin, R. C. (2017). *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press.
3. Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
4. Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Pluskal Jan, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 30. října 2018

Abstrakt

Cílem této práce je porovnat softwarové architektury tak, aby čtenář byl schopný rozhodnout, která architektura je vhodná právě pro jeho projekt. K dosažení tohoto cíle práce nabízí porovnání a ukázkovou implementaci Cibulové architektury, Architektury anemického doménového modelu a Architektury aktivních záznamů. Architektury jsou porovnány na základě čitelnosti, rozšiřitelnosti, testovatelnosti a potřebných znalostí pro vývoj. Práce také nabízí popis dalších obecně známých softwarových architektur a konceptů používaných v těchto architekturách.

Abstract

The goal of this thesis is to compare software architectures to help the reader decide which architecture is the most suitable for their project. The thesis compares the Onion architecture, Anemic domain model architecture and Active record architecture. The architectures are compared based on their readability, extensibility, testability and the amount of knowledge neccesary to implement each architecture. It also offers the description of other well-known software architectures and the concepts which they use.

Klíčová slova

Softwarová architektura, Hexagonální architektura, Cibulová architektura, Transakční skript, Doménou řízený vývoj, Aktivní záznam, Repozitář, Sdílený slovník, Vrstvená architektura, Doménový model, Byznys logika

Keywords

Software architecture, Hexagonal architecture, Onion architecture, Transaction script, Domain Driven Design, Transaction script, Active record, Repository, Service, Dependency inversion, Ubiquous language, Layered architecture

Citace

MOTYČKA, Michal. *Porovnání softwarových architektur*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Porovnání softwarových architektur

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Pluskala. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Michal Motyčka
16. května 2019

Poděkování

Rád bych poděkoval Ing. Janu Pluskalovi za vedení při vypracování bakalářské práce.

Obsah

1 Úvod 3

2 Moderní softwarová architektura 4

2.1 Byznys logika . . . 4

2.2 Vrstvená architektura . . . 6

2.2.1 Otevřené a uzavřené vrstvy . . . 7

2.2.2 Otevřené vrstvy . . . 7

2.2.3 Výhody a nevýhody uzavřených a otevřených vrstev . . . 8

2.3 Chytré UI . . . 9

2.4 Transakční skript . . . 9

2.4.1 Organizace transakcí . . . 9

2.4.2 Architektura transakčních skriptů . . . 10

2.5 Aktivní záznam . . . 10

2.5.1 Cizí klíče v Aktivním záznamu . . . 11

2.5.2 Architektura založená na Aktivním záznamu . . . 11

2.6 Hexagonální architektura . . . 12

2.6.1 Výhody hexagonální architektury . . . 13

2.6.2 Nevýhody Hexagonální architektury . . . 14

2.7 Doménový model . . . 14

2.7.1 Vývoj řízený doménou . . . 15

2.7.2 Sdílený slovník . . . 16

2.7.3 Entity . . . 16

2.7.4 Hodnotové objekty . . . 17

2.7.5 Služby . . . 17

2.8 Agregáty . . . 19

2.9 Cibulová architektura . . . 19

2.10 Anemický doménový model . . . 22

3 Návrh demonstrační aplikace 24

3.1 Školní systém . . . 24

3.2 Návrh aplikace v Architektuře aktivního záznamu . . . 26

3.3 Návrh aplikace v Cibulové architektuře . . . 26

3.4 Návrh aplikace v Architektuře anemického modelu . . . 27

4 Implementace vybraných architektur 29

4.1 Uplatňování byznys pravidel a validací . . . 29

4.2 Mapování doménového modelu na databázový pomocí Entity frameworku . 31

4.2.1 Mapování privátních atributů . . . 31

4.2.2 Mapování M:N vztahu 32
4.3 Volání metod ze stejné vrstvy 33

5 Porovnání architektur **35**
5.1 Potřebné znalosti týmu vývojářů 35
5.2 Čitelnost architektur 36
5.3 Rozšiřitelnost architektury 37
5.3.1 Výsledky testování 39
5.3.2 Zhodnocení 40
5.4 Testovatelnost Architektury aktivního záznamu 41
5.5 Celkové zhodnocení testovaní architektur 41
5.6 Limitace porovnání architektur 42

6 Závěr **43**

Literatura **45**

Přílohy **48**

A Obsah přiloženého paměťového média **49**

B Test znalostí programátorů **50**
B.1 Cibulová architektura 50
B.2 Anemic domain model 52
B.3 Active record 53
B.4 Bonusová otázka: 55
B.5 Výsledky testu 55

C Výsledky měření čitelnosti architektur **57**
C.1 Architektura anemického doménového modelu 57
C.2 Cibulová architektura 57

D Výsledky měření rozšiřitelnosti architektur **58**
D.1 Architektura anemického doménového modelu 58
D.2 Cibulová architektura 58
D.3 Architektura aktivních záznamů 58

Kapitola 1

Úvod

Softwarová architektura je pojem s mnoha různými významy, avšak pro tuto práci je nejvhodnější následující definice — softwarová architektura jsou rozhodnutí, která by si programátoři přáli udělat správně na začátku vývoje aplikace, jelikož jsou vnímány jako obtížné na změnu později ve vývoji [10]. V této definici je důležité, že softwarová architektura je subjektivní. Pokud považujeme něco za obtížné změnit, ale později se ukáže, že změna není tak obtížná, tak už to není architektonické rozhodnutí. Z této definice vyplývá, že výběr architektury je jedním z nejdůležitějších rozhodnutí, které programátoři musí na začátku vývoje udělat, jelikož pozdější změna je obtížná. Zvolená architektura často provází programátory po celou dobu života aplikace a špatná volba může vést k prodražení celého vývoje.

Výběr architektury je obtížný, jelikož neexistuje jedna architektura, která by byla vhodná pro všechny druhy aplikací. Architektury mají obvykle silné a slabé stránky a pro každý typ aplikace jsou vhodné jiné vlastnosti. Otázkou tedy je, jak zvolit správnou architekturu aplikace. Kvůli komplexitě softwarových aplikací není jednoduché na tuto otázku odpovědět přesně. Můžeme ale porovnat jednotlivé architektury mezi sebou a na základě těchto výsledků rozhodnout, která architektura je nejvhodnější pro určitý projekt.

Tato práce slouží jako výukový materiál pro programátory, kteří vyvíjejí novou aplikaci a snaží se rozhodnout, kterou architekturu zvolit. Práce neposkytuje přesnou odpověď na to, jakou architekturu je nejlepší zvolit. Poskytuje pouze obecné porovnání některých vlastností architektur, na základě kterého se pak čtenář musí rozhodnout, která architektura je nejvhodnější pro jeho projekt. Práce se také zabývá pouze architekturami monolitických aplikací a převážně byznys aplikacemi.

Ještě před porovnáním architektur práce obsahuje popis často používaných architektur, návrh demonstrační aplikace a popis zajímavých částí implementace této aplikace. Architektury vybrané pro popis byly zvoleny na základě známých knih o softwarové architektuře a také podle použití architektur v programech s otevřeným zdrojovým kódem. Z popsaných architektur jsou pak vybrány tři, pomocí kterých je navržena a implementována demonstrační aplikace. Na závěr práce porovnává architektury pomocí experimentů na ukázkových aplikacích.

Kapitola 2

Moderní softwarová architektura

Architektury aplikací se často liší v mnoha aspektech, mnohdy je ale možné identifikovat několik vlastností, které poměrně specificky definují celou architekturu aplikace. V některých případech aplikace může používat i kombinaci několika architektur, avšak u monolitických aplikací není tento způsob často používán.

Byznys logika aplikace obvykle obsahuje jeden ze čtyř návrhových vzorů — doménový model, transakční skript, tabulkový modul nebo Aktivní záznam [31]. Tato kapitola popisuje všechny z těchto návrhových vzorů s výjimkou tabulkového modulu, jelikož tabulkový modul je pouze variací Aktivního záznamu [31] a Aktivní záznam je častěji používán [28, 16, 5]. Každý z těchto návrhových vzorů mluví pouze o byznys logice aplikace a architektura vzniká až kombinací s vrstvenou nebo hexagonální architekturou. Tato kapitola vždy popisuje nejdříve návrhový vzor a až poté architekturu celé aplikace. Diagram 2.1 ukazuje jednotlivé architektury a koncepty popsané v této kapitole.

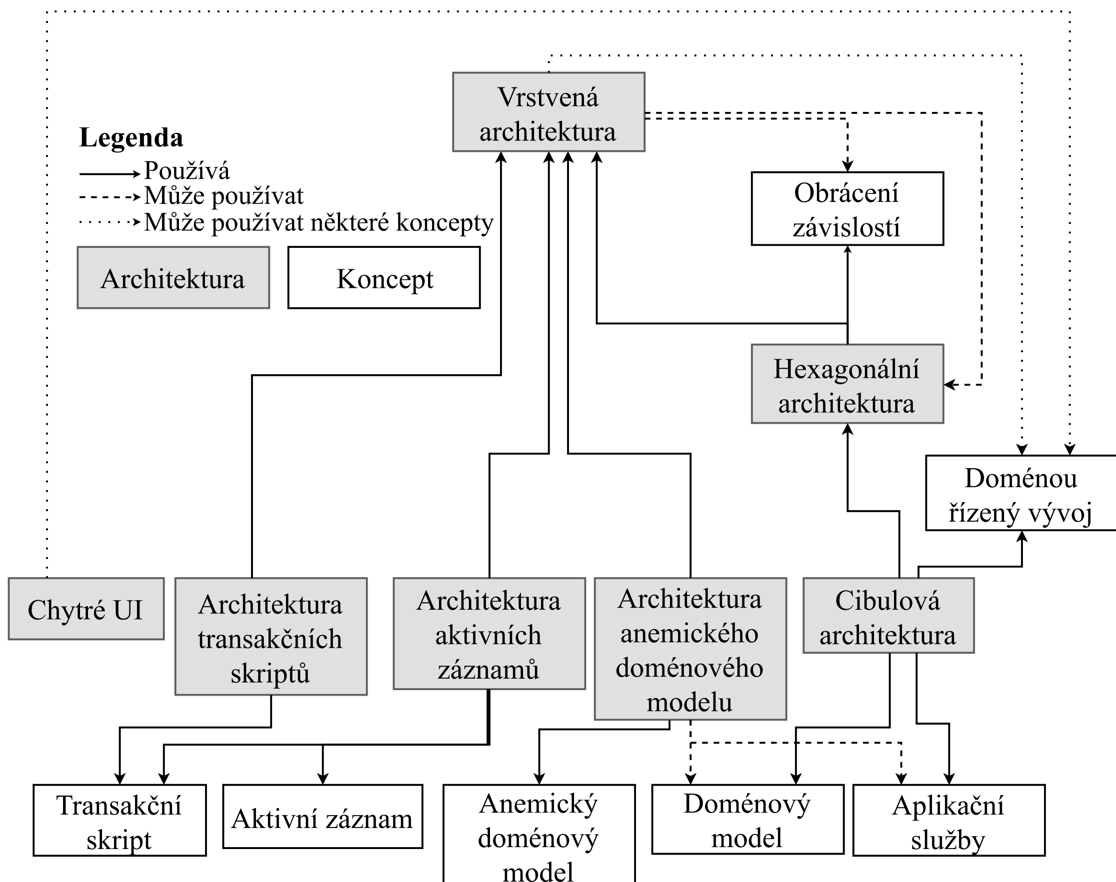
Několik následujících kapitol popisuje základní informace potřebné pro porozumění architektur. Poté následuje popis architektur samotných. Architektury jsou logicky seřazeny podle toho, jakým způsobem na sebe navazují. Pokud na sebe nenavazují, jsou seřazeny podle složitosti na implementaci.

2.1 Byznys logika

Architektury a příklady v této práci se často soustředí na byznys aplikace. Byznys aplikace je každý program, který je používaný lidmi z byznysu pro provedení různých byznys funkcí [38]. Každá byznys aplikace obsahuje byznys logiku, která popisuje byznys pravidla z reálného světa, která rozhodují, jestli mohou data být vytvořena, uložena nebo změněna [37]. Byznys pravidla jsou pak aspekty nebo omezení byznysu, která jsou vždy pravdivá nebo nepravdivá. Například v aplikaci, která modeluje byznys účetnictví, by byznys logika byla ta část aplikace, která obsahuje jaké náležitosti by měla obsahovat faktura. Naopak součástí byznys logiky nebude ta část aplikace, která se stará o ukládání dat do databáze.

Model, Pohled a Řadič

Návrhový vzor Model, Pohled, Řadič, [2] dále jen MVC, rozděluje aplikaci na tři části, podle kterých získal název: model, pohled a řadič. Pohledy obsahují kód, který se stará o vykreslení uživatelského rozhraní. Řadič je třída, která přijímá požadavky od klienta a koordinuje operace, které jsou potřeba pro jejich provedení. Model obsahuje přístup k databázi a také byznys logiku aplikace. Běžný požadavek od uživatele je zpracován tímto způsobem: Řadič



Obrázek 2.1: Diagram architektur a konceptů popsaných v této kapitole. Všechny vztahy jsou tranzitivní. Například platí, že Cibulová architektura používá obrácení závislostí díky tomu, že má vztah s Hexagonální architekturou.

přijme požadavek od klienta, poté vezme data z modelu a předá je pohledu. Nakonec řadič vrátí naformátovaná data z pohledu zpět uživateli.

MVC je pro tuto práci důležité, jelikož jsou architektury často demonstrovány pomocí tohoto návrhového vzoru. Pro demonstraci architektury typicky stačí Model a Řadič, kde řadič přijímá požadavky od klienta a poté vrací data. Příkladem takové aplikace bez Pohledu může být webové API.

Objektově relační mapování

Objektově relační mapování, dále jen ORM, je programovací technika, která zajišťuje konverzi mezi relačními databází a objektově orientovaným jazykem [39]. V mnoha aplikacích existují objekty, které přesně odpovídají tabulkám v databázi a jejich atributy přesně odpovídají sloupcům v databázi. Poté je použit nástroj, který se stará o ORM mapování mezi těmito tabulkami a objekty. Ve složitějších aplikacích nemusí objekty přesně odpovídat databázovým tabulkám a je potřeba komplexnější nástroj pro ORM.

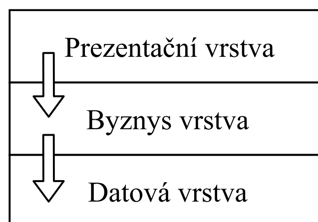
2.2 Vrstvená architektura

První z vybraných architektur je vrstvená architektura. Rozdělení do vrstev je jedna z nejběžnějších technik používaných pro modelování systémů [30]. Příkladem vrstvení může být protokol TCP/IP, který používá čtyři vrstvy [15, p. 16]. Byznys aplikace v minulosti používaly jednu vrstvu, avšak s příchodem objektového programování a klient server aplikací se začaly dělit pomocí tří a více vrstev [10]. Každá z těchto vrstev má nějakou zodpovědnost, o kterou se stará, například datová vrstva se stará o ukládání dat do databáze. *Patterns of enterprise architecture* [10] zmiňuje tři hlavní vrstvy a jejich zodpovědnosti:

1. Prezentační — stará se o zobrazení dat uživateli a zpracování dat od uživatele.
2. Doménová (byznys) — provádí byznys logiku.
3. Datová — stará se o ukládání a načítání dat z databáze.

Na obrázku 2.2 můžeme vidět průchod požadavku od uživatele vrstvami. Uživatel obvykle komunikuje s prezenční vrstvou, která komunikuje s vrstvami pod ní. Tyto vrstvy pak komunikují s dalšími nižšími vrstvami, dokud nějaká z vrstev nerozhodne, že je požadavek zpracován. V každém kroku se s požadavek provede nějaká operace. Touto operací může být například transformace předávaných dat do jiného formátu nebo třeba uložení dat do databáze. Požadavek se nemusí vždy dostat až k nejnižší vrstvě. Některá z vrstev se může rozhodnout, že data nejsou validní nebo že dokáže celý požadavek obsloužit bez pomoci nižších vrstev. V případě, že vrstva zpracuje požadavek, může vrátit data, které poté projdou přes vrstvy zpět ke klientovi.

Je důležité zmínit, že žádná vrstva nesmí komunikovat s vrstvou nad ní, ale pouze s nižšími vrstvami [6]. Toto rozdělení aplikace má výhodu, že některé změny v aplikaci vedou pouze ke změně jedné vrstvy a není nutné měnit i jiné části aplikace. Například pokud se rozhodneme změnit UI z konzolové aplikace na webovou stránku, stačí změnit pouze prezenční vrstvu a ostatní vrstvy mohou zůstat stejné.



Obrázek 2.2: Průchod uživatelského požadavku vrstvami. Šipky naznačují průchod přes jednotlivé vrstvy.

2.2.1 Otevřené a uzavřené vrstvy

V některých situacích se může stát, že vrstva, přes kterou požadavek prochází, nemá žádnou operaci, kterou nad požadavkem potřebuje provést. Pokud například aplikace používá prezenční, doménovou a datovou vrstvu a chceme uživateli zobrazit všechny uživatele z databáze, musíme projít přes všechny vrstvy. Požadavek vznikne na prezenční vrstvě, ta předá informace doménové vrstvě a ta zažádá datovou vrstvu o data z databáze. Data se poté vrátí zpět k uživateli přes všechny tři vrstvy. Kód k tomuto příkladu můžeme vidět v ukázce 2.1. V tomto příkladu můžeme vidět, že doménová vrstva neprovedla žádnou operaci, pouze dostala informace z prezenční vrstvy a poté zavolala datovou vrstvu. Toto zbytečné procházení vrstev může vést k několika nevýhodám, a proto je vhodné v některých implementacích vrstvené architektury použít otevřené vrstvy [30, 27].

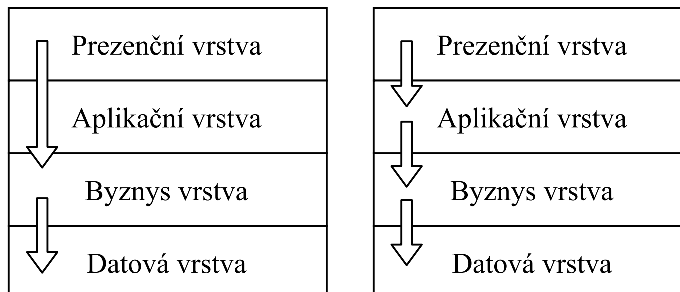
```
1  public class UserBusinessLayer {
2      private readonly UserDataLayer userDataLayer;
3
4      public List<User> GetAllUsers() {
5          return userDataLayer.GetAllUsers();
6      }
7  }
8
9  public class UserDataLayer {
10     public List<User> GetAllUsers() {
11         //Access db and get all users
12     }
13 }
```

Výpis 2.1: Ukázka architektury používající uzavřené vrstvy. `UserBusinessLayer` reprezentuje byznys vrstvu architektury, která umožňuje získat všechny uživatele z databáze. Pro získání všech uživatelů ale není potřeba žádná byznys logika, proto byznys vrstva pouze zavolá datovou vrstvu a vrátí výsledek operace.

2.2.2 Otevřené vrstvy

Otevřená vrstva je taková vrstva, kterou můžeme obejít a přejít přímo k vrstvě pod ní. Na obrázku 2.3 můžeme vidět, jak systémem prochází požadavek, pokud je byznys vrstva otevřená. Uzavřené vrstvy jsou pak ty, které nelze obejít. Pokud architektura obsahuje pouze uzavřené vrstvy, můžeme takovou architekturu nazvat uzavřená architektura, v opačném

případě otevřená architektura [27, 30]. Použití uzavřených a otevřených vrstev má své výhody a nevýhody.



Obrázek 2.3: Ukázka průchodu uživatelského požadavku aplikací s otevřenou vrstvou. Aplikační vrstva je otevřená vrstva a všechny ostatní jsou uzavřené. V první ukázce vlevo požadavek přeskočí aplikační vrstvu a pokračuje přímo k byznys vrstvě. V druhé ukázce vpravo jiný požadavek prochází i přes otevřenou vrstvu.

2.2.3 Výhody a nevýhody uzavřených a otevřených vrstev

Výhodou uzavřené architektury je minimalizace závislostí mezi jednotlivými vrstvami. Každá vrstva má závislost pouze na vrstvě přímo pod ní. Pokud se tedy změní veřejné rozhraní[1] uzavřené vrstvy, změní se většinou pouze vrstva přímo nad ní [27]. Úprava veřejného rozhraní vrstvy může vést i ke změně veřejného rozhraní další vrstvy, což způsobí kaskádu změn. Uzavřená aplikace tedy nezajišťuje, že se změna projeví pouze na jedné vrstvě.

Nevýhodou uzavřené architektury je, že některé změny v systému mohou vést ke změně ve všech vrstvách, příkladem takové změny může být rozšíření UI o nové pole a přidání tohoto pole do databáze [10]. Nová data v databázi musejí být předána přes všechny vrstvy až k té nejvyšší, což nutně vede k úpravě objektů používaných v jednotlivých vrstvách. Uzavřená architektura tedy nezaručuje úplné zapouzdření. Další nevýhody jsou uvedeny v následujícím seznamu:

- vrstvy bez funkcionality mohou vést ke zpomalení celé aplikace,
- programátor musí psát kód navíc, který nic nedělá,
- provolávání může dělat kód méně přehledný.

Otevřené vrstvy řeší všechny nevýhody uzavřených vrstev, ale za to zvyšují počet závislostí v aplikaci. V určitém smyslu porušují zapouzdření aplikace. V příkladu 2.1 je ukázáno získávání všech uživatelů z databáze pomocí uzavřené vrstvy. Pokud by bylo potřeba přidat byznys logiku ke získání uživatelů, změnila by se pouze metoda `GetAllUsers` ve třídě `UserBusinessLayer`. Pokud bychom použili architekturu, ve které by byla byznys vrstva otevřená, mohli bychom z UI volat přímo `UserDataLayer.GetAllUsers()`. Pokud by bylo potřeba přidat byznys logiku, do tohoto požadavku bylo by nutné vytvořit metodu `UserBusinessLayer.GetAllUsers()` a tu volat z UI namísto `UserDataLayer.GetAllUsers()`. Změnila by se tedy jak UI vrstva, tak byznys vrstva. Otevřené vrstvy by také mohly svádět programátora k vložení byznys logiky do UI vrstvy namísto vytváření

[1] Rozhraní, které může být voláno jinými vrstvami.

metody v byznys vrstvě. Vložení byznys logiky do UI by pak porušilo rozdělení zodpovědností jednotlivých vrstev. Oba přístupy mají tedy výhody i nevýhody a rozhodnutí, který způsob vrstvení použít, závisí na tom, jakou aplikaci píšeme.

2.3 Chytré UI

Chytré UI jednoduše rozděluje aplikaci do uživatelských rozhraní, kde každé toto rozhraní obsahuje všechnu logiku, kterou potřebuje pro své fungování a vykreslování [6]. Chytré UI je jediný zástupce z vybraných architektur, který nepoužívá vrstvenou architekturu. Tuto architekturu je vhodné použít na malých projektech, které nemají mnoho byznys logiky [6]. U větších aplikací začne být kód nepřehledný a těžko rozšiřitelný [6]. Pokud se rozhodneme použít tuto architekturu, je vhodné použít generátory UI [6]. Použitím těchto nástrojů dosáhneme rychlých výsledků a vyhneme se psaní nepotřebného kódu.

Pokud je šance, že se projekt, na kterém pracujeme, bude dále rozšiřovat, je dobré použít jinou architekturu. Popřípadě je možné co nejrychleji vytvořit MVP[2] pomocí Chytrého UI a poté aplikaci přepsat od začátku použitím jiné architektury.

2.4 Transakční skript

O většině byznys aplikací můžeme přemýšlet jako o sérii transakcí [10]. Transakční skript [21] rozděluje aplikaci do procedur, kde každá odpovídá jedné byznys transakci. Jako jednoduchý příklad transakcí můžeme uvést nákup zboží v E-shopu:

- přidání zboží k zboží k objednávce
- vložení údajů o dodání zboží,
- potvrzení nákupu,
- zaplacení.

Každé takové transakci pak v aplikaci bude odpovídat jedna metoda, která provede všechny potřebné byznys operace. Na obrázku 2.4 jsou ukázány jednotlivé metody přiřazené do tříd.

Transakční skript není objektově orientovaný způsob programování [31]. Ačkoliv jednotlivé Transakční skripty dělíme do tříd, k modelování byznys logiky nemůžeme použít objektový přístup. Jednotlivé Transakční skripty pracují pouze s daty, které jim jsou předány jako parametr. Aby Transakční skript měl přístup k předaným datům, musí tato data být veřejná. Předaná data také nesmí obsahovat žádné metody, které provádí byznys logiku, jelikož logika je obsažena v Transakčních skriptech. Transakční skript tedy porušuje základní principy objektového programování jako je spojování dat a operací [8].

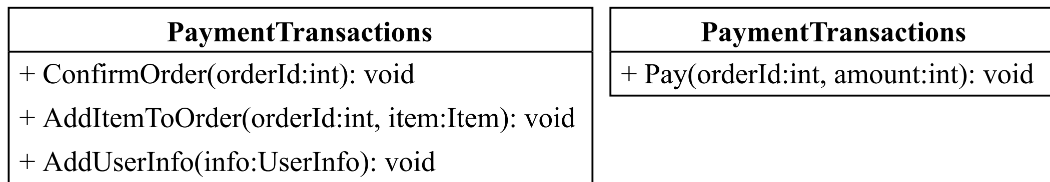
2.4.1 Organizace transakcí

Transakční skripty můžeme organizovat do metod, ale také do objektů. V případě organizace do objektů můžeme využít návrhový vzor Příkaz [11] a přiřadit každé transakci jednu třídu. Tento příklad je ukázán na obrázku 2.5.

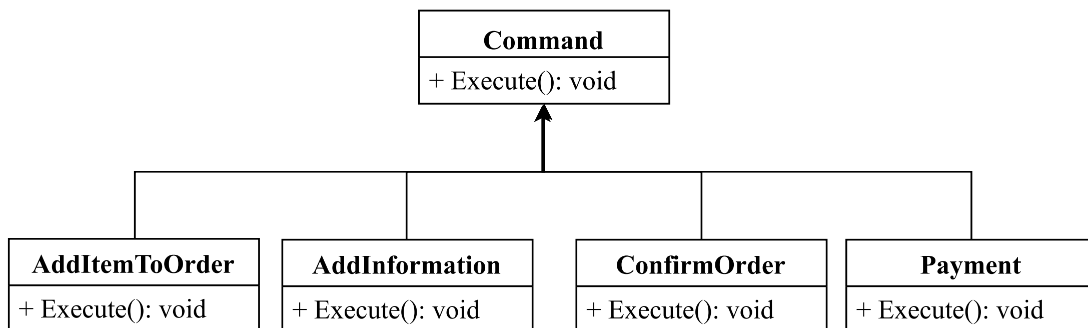
Použití návrhového vzoru Příkaz má tu výhodu, že můžeme s jednotlivými transakcemi pracovat jako s obecným příkazem. Někdo nám tedy může dodat několik příkazů a my

[2]Produkt s nejmenší možnou funkcionalitou, ale takovou, aby ho bylo možné používat.

je můžeme spustit bez znalosti vnitřní implementace. Martin Fowler ale zmiňuje, že on sám viděl pouze pár aplikací, které by tuto možnost využily [10].



Obrázek 2.4: Ukázka rozdělení transakcí do metod a tříd.



Obrázek 2.5: Ukázka použití návrhového vzoru Příkaz v Transakčním skriptu.

2.4.2 Architektura transakčních skriptů

Při implementaci Transakčního skriptu je vhodné oddělit prezenční a datovou vrstvu od transakcí. Transakční skript je tedy vhodné použít v kombinaci s tří vrstvou architekturu. V případě, že se rozhodneme nepoužít datovou vrstvu, může dojít k častému duplikování SQL dotazů v transakcích. Ve většině případů je tedy vhodné použít i datovou vrstvu. Pro Transakční skript neexistuje žádné obecné pravidlo, které by říkalo, které vrstvy by měli být otevřené a které uzavřené. Záleží tedy na programátorovi, jaký druh implementace zvolí.

2.5 Aktivní záznam

Aktivní záznam [18] je třída, jejíž atributy odpovídají sloupcům v tabulce. Pro každou tabulku v databázi aplikace pak existuje jeden Aktivní záznam. Aktivní záznam se stará o ukládání, vkládání, aktualizování a mazání záznamů z databáze a může obsahovat i byznys logiku [18]. Příklad Aktivního záznamu můžeme vidět na obrázku 2.6. Aktivní záznam typicky obsahuje tyto věci [10]:

- Statickou metodu pro vytvoření instance Aktivního záznamu z vybraného řádku v SQL tabulce popřípadě z SQL pohledu.
- Statickou metodu, popřípadě konstruktor pro vytvoření nové instance, která bude později uložena do databáze.
- Statické metody pro často používané dotazy na databázi.
- Metody pro aktualizaci dat v databázi.

- Metodu pro vložení dat do databáze.
- Metody pro získání a nastavení atributů.
- Malé množství byznys logiky.

Course
- courseId:int - name:string - teacherId:int
+ Update(): void + Insert(): void + Delete(int id): void + GetById(int id): Course

Obrázek 2.6: Ukázka Aktivního záznamu. Z ukázky jsou pro zkrácení vynechaný metody pro získání a změnu atributů. Tento Aktivní záznam odpovídá tabulce v databázi, která obsahuje sloupce `courseId`, `name` a `teacherId`. `Update` metoda uloží aktuální stav objektu do databáze. Metoda `Insert` vytvoří nový záznam v databázi s aktuálním stavem. Statická metoda `Delete` smaže záznam s předaným identifikátorem. Statická metoda `GetById` vytvoří nový objekt `Course` a nastaví aktuální stav atributů podle hodnot z databáze.

2.5.1 Cizí klíče v Aktivním záznamu

Důležitou otázkou při implementaci Aktivního záznamu je, jak se vypořádat s cizími klíči. Nejjednodušším způsobem je v Aktivních záznamech vytvořit atributy, které odpovídají cizím klíčům v databázi. V některých případech však může být vhodné použít Mapování cizích klíčů. Mapování cizích klíčů je návrhový vzor, který nahrazuje identifikátory z databáze za reference na objekt/objekty [10]. Příklad obou přístupů můžeme vidět na obrázku 2.7.

2.5.2 Architektura založená na Aktivním záznamu

Použití Aktivního záznamu můžeme vidět v Ruby on Rails [29] komunitě [28]. Ruby on Rails používá návrhový vzor MVC popsaný v kapitole 2.1, kde model ve většině případů obsahuje

Course
- courseId:int - name:string - teacherId:int

Course
- courseId:int - name:string - teacher: Teacher

Obrázek 2.7: Ukázka mapování cizích klíčů. Vlevo je ukázána třída bez použití mapování cizích klíčů a vpravo je ukázána třída s použitím.

Prezenční	Řadič
Byznys	Transakční skripty Aktivní záznamy
Datová	SQL

Obrázek 2.8: Ukázka vrstev v Architektuře založené na Aktivních záznamech. U každé vrstvy je uveden její název a věc, kterou obsahuje. Prezenční vrstva v této ukázce obsahuje řadič, ale mohla by obsahovat jinou službu pro komunikaci s uživatelem.

pouze Aktivní záznamy [28]. Stejná architektura se obvykle používá i v PHP frameworku Laravel [16] a Python frameworku Django [5].

Architektura, ve které model obsahuje pouze Aktivní záznamy, vede k vkládání byznys logiky do Aktivních záznamů a popřípadě také do řadičů. Aktivní záznamy by ale měli obsahovat pouze malé množství byznys logiky [10]. Jako správné řešení se tedy může zdát vložení byznys logiky do řadičů, avšak toto řešení vede k několika problémům:

- Při změně prezentační vrstvy je potřeba měnit byznys logiku aplikace.
- Podle principu jedné zodpovědnosti [18] by každý objekt nebo modul měl provádět pouze jednu věc. Řadič se stará o přijímaní požadavků od klienta, a proto přidání byznys logiky do řadiče vede k porušení principu jedné zodpovědnosti.

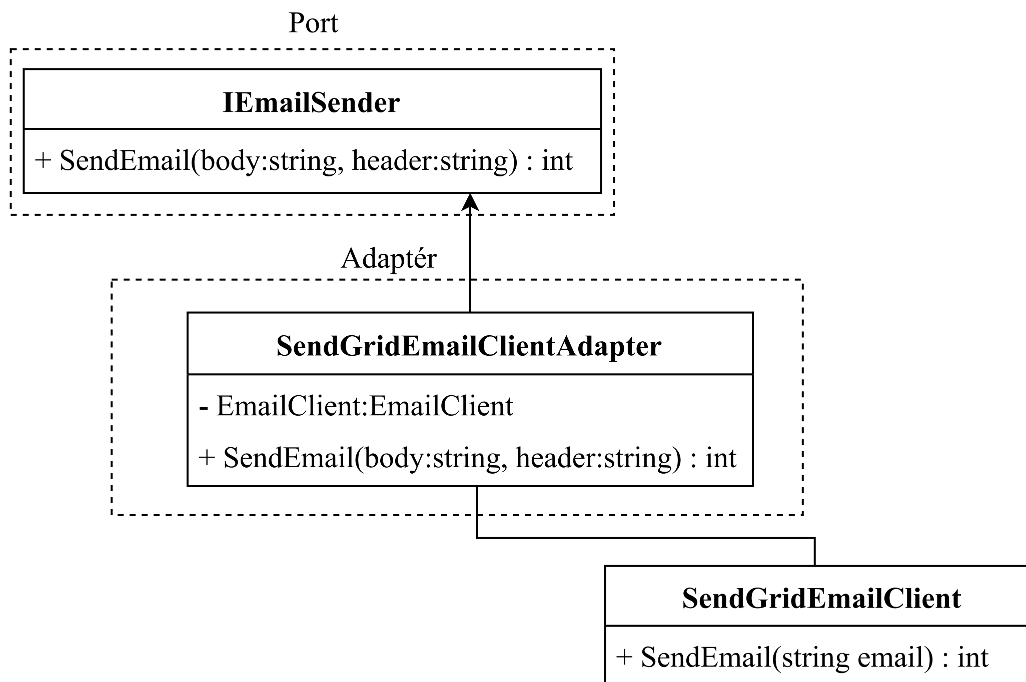
Problém s byznys logikou v řadičích můžeme vyřešit kombinací Aktivních záznamů s Transakčními skripty. Namísto vkládání byznys logiky do řadičů je možné logiku vložit do transakčních skriptů. Tímto způsoben není porušen princip jedné zodpovědnosti a není ani potřeba vkládat velké množství logiky do Aktivních záznamů. Transakční skripty poté používají ve většině případů Aktivní záznamy pro přístup do databáze. V některých případech je ale vhodné přistupovat k databázi přímo z transakčních skriptů. Obrázek 2.8 ukazuje vrstvy architektury popsané v tomto odstavci.

2.6 Hexagonální architektura

Hexagonální architektura [23] rozděluje aplikaci na vnitřní a vnější část. Vnitřní část obvykle obsahuje pouze byznys logiku aplikace a vnější část obsahuje vše, co není součástí byznys logiky. Datová a prezenční vrstva, které byly popsány v podkapitole 2.2, tedy patří do vnější části aplikace. Vnější a vnitřní část spolu komunikují pomocí návrhového vzoru adaptér, tento vzor je ukázán na obrázku 2.9. Adaptér v kontextu hexagonální architektury se na rozdíl od návrhového vzoru Adaptér skládá z portu a adaptéru.

Hexagonální architektura dále dělí adaptéry na primární a sekundární. Primární adaptéry jsou ty, které zahajují komunikaci s byznys vrstvou a sekundární jsou ty, se kterými zahajuje komunikaci byznys vrstva. Příkladem primárního adaptéru může být již zmíněná prezentační vrstva. Diagram 2.10 ukazuje primární adaptér.

Sekundární adaptéry obvykle oddělují byznys logiku aplikace od externích služeb využívaných aplikací, jako je například databáze nebo e-mailový klient. Ukázka sekundárního adaptéru je na obrázku 2.9.



Obrázek 2.9: Ukázka návrhového vzoru adaptér a jeho konkrétního využití v Hexagonální architektuře. Byznys logika aplikace komunikuje pouze s portem a nezná jeho přesnou implementaci. Adaptér zajišťuje, že se byznys logika nemusí přizpůsobit e-mailovému klientovi.

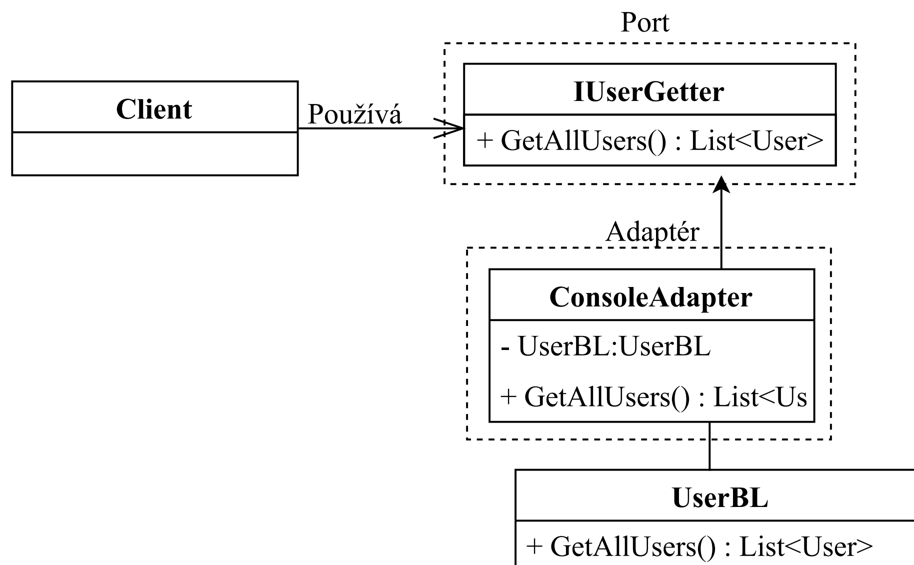
Obrázek 2.11 ukazuje celou hexagonální architekturu. Mezi každou službou na obrázku 2.11 leží port a adaptér. Služby na pravé straně jsou ty, se kterými byznys logika komunikuje pomocí sekundárních adaptérů a na levé straně jsou ukázány služby, které využívají primární adaptéry.

2.6.1 Výhody hexagonální architektury

Jednou z hlavních výhod hexagonální architektury je testovatelnost aplikace pomocí jednotkových testů. Primární porty přesně definují veřejné rozhraní byznys logiky, které je potřeba testovat, a sekundární porty umožňují nahradit všechny závislosti byznys logiky za testovací. Testy v této architektuře mají také výhodu, že dokáží odhalit externí služby, do kterých se omylem dostala byznys logika aplikace. Například pokud programátor vloží byznys logiku aplikace do prezenční vrstvy, bude tato logika chybět v jádru aplikace a tester nebude mít možnost jí otestovat. Tímto způsobem dojde ke zjištění, že se byznys logika dostala na špatné místo a je potřeba ji přesunout.

Použití hexagonální architektury umožňuje nejdříve implementovat byznys logiku a v případě potřeby vytvořit pouze port bez implementace. Tento přístup zaručí, že se programátor může soustředit na psaní byznys logiky aplikace a nemusí se zabývat infrastrukturou.

Další výhodou je, že byznys logika není ohýbána tak, aby vyhovovala externím službám, ale naopak externí služby jsou pomocí adaptérů přizpůsobeny potřebám byznys logiky. Diagram 2.9 ukazuje externí službu, která je pomocí adaptéru upravena tak, aby vyhovovala byznys logice aplikace. V tomto diagramu třída `SendGridEmailClient` vyžaduje, aby e-mail byl předán jako jeden řetězec, avšak aplikace má k dispozici pouze `header` a `body`. Bylo



Obrázek 2.10: Ukázka primárního adaptéru a portu. Na levé straně je klient, který využívá byznys logiku, kterou zakrývá port a adaptér. Použití portu a adaptéru zajišťuje, že změna byznys logiky nezpůsobí změnu klienta. Na rozdíl od sekundárního adaptéru je primární adaptér a jeho port součástí byznys logiky aplikace.

by chybou, kdyby se byznys logika přizpůsobila službě. Služba se pomocí adaptéru musí přizpůsobit byznys logice tak, jak to ukazuje diagram.

V hexagonální architektuře je i databáze externí službou. Byznys logika tedy nezná konkrétní implementaci databáze a používá pouze databázoví port. Tento přistup vede k oddělení byznys logiky od databáze. Je tedy možné vytvořit aplikaci bez nutnosti návrhu databáze. Tento přístup vede k architekturám, které nepoužívají databázi jako hlavní část.

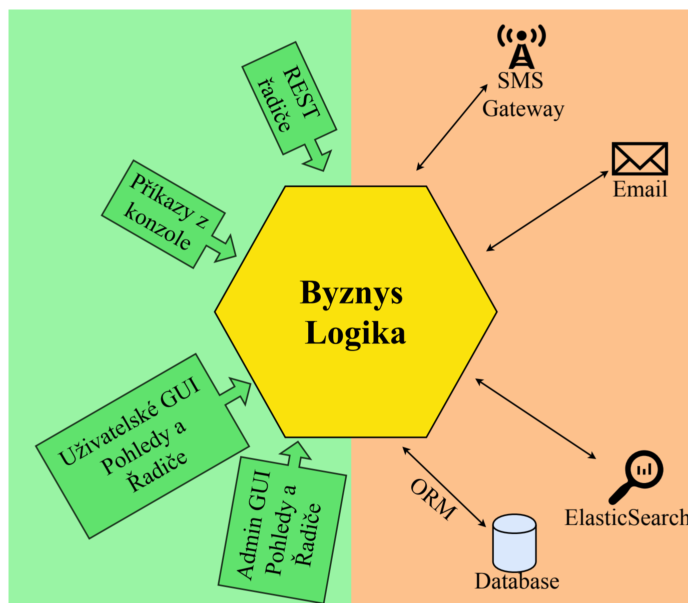
2.6.2 Nevýhody Hexagonální architektury

Jedinou nevýhodou hexagonální architektury je delší implementace. Jelikož všechna komunikace s byznys logikou musí probíhat přes porty a adaptéry, je potřeba větší množství kódu. Stejně tak volání externích služeb musí probíhat přes porty a adaptéry, což vede k většímu množství kódu, který u jednodušších aplikací může být zbytečný.

2.7 Doménový model

> The key to controlling complexity is a good domain model, a model that goes beyond a surface vision of a domain by introducing an underlying structure, which gives the software developers the leverage they need. A good domain model can be incredibly valuable, but it's not something that's easy to make. Few people can do it well, and it's very hard to teach. [6]

Doménový model [21] je objektový přístup k modelování byznys logiky aplikace. Všechny předchozí architektury, s výjimkou Hexagonální, jsou obvykle modelovány na základě schéma databáze. Klasický přístup při návrhu architektury je identifikace podstatných jmen a sloves, podle kterých se následně vytvoří model databáze. Na základě tohoto modelu je poté



Obrázek 2.11: Ukázka hexagonální architektury na levé straně jsou služby, které používají byznys logiku, a na pravé jsou ty, které jsou používány byznys logikou. Obrázek byl převzat z webové stránky [12].

modelována i byznys logika aplikace. Tento přístup k návrhu byznys logiky je zavádějící, jelikož databázový model ve většině případů nedokáže vyjádřit složitou byznys logiku.

Doménový model řeší tento problém použitím objektově orientovaného programování pro modelování byznys logiky aplikace. OOP na rozdíl od databázového modelu umožňuje použití mnoha prostředků pro vyjádření byznys logiky, jako je dědičnost a zapouzdření. Naopak databázový model obsahuje pouze tabulky a relace, které nedokáží dobře vyjádřit problémy doménové logiky. Například pokud aplikace potřebuje modelovat různé chování pro různé skupiny uživatelů, je jednoduché a intuitivní použít polymorfizmus v OOP, avšak databázový model nemá žádné řešení, které jednoznačně vyjadřuje tuto byznys logiku.

Nevýhodou doménového modelu je, že aplikace musí mapovat doménový model do databázového, aby bylo možné ukládání dat do databáze. V některých případech se doménový model aplikace velice liší od databázového modelu a je nutné vytvořit dva modely. Mezi těmito modely je poté potřeba vytvořit mapovací vrstvu, která převádí doménový model na databázový a databázový na doménový. Ve většině případů je ale dostačující vytvořit jeden model a nastavit ORM tak, aby správně mapovalo atributy na tabulky. ORM často umožňuje specifikovat, které atributy není potřeba ukládat do databáze a které se mají nějakým způsobem mapovat na jiný typ nebo tabulku. Dalším řešením problému mapování je použití NoSQL databáze, avšak takové řešení je za hranicemi rozsahu této práce. Další informace o použití NoSQL jako úložiště doménového modelu můžeme najít zde [7].

2.7.1 Vývoj řízený doménou

Vývoj řízený doménou, dále jen DDD, je technika, která mimo jiné popisuje tvorbu doménového modelu popsaného v kapitole 2.7. Tato práce nepopisuje všechny koncepty DDD, ale soustředí se pouze na ty, které jsou podstatné z pohledu porovnávání architektur. Další informace o DDD je možné nalézt v knihách [6] a [31]. DDD obsahuje dva druhy návr-

hových vzorů, strategické a taktické. Strategické vzory tvarují řešení problému a taktické se používají k modelování domény [31]. Strategické vzory mají výhodu, že je můžeme použít na všech projektech, bez ohledu na architekturu nebo programovací jazyk [31]. I přes tuto výhodu se tato práce dále zabývá převážně Taktickými vzory, jelikož ty jsou potřebné pro modelování doménového modelu.

DDD dává důraz na srozumitelný kód, který dokáží pochopit i lidé, kteří neumí programovat. Toho se snaží docílit pomocí striktního oddělení byznys logiky a zbytku aplikace, podobně jako hexagonální architektura popsaná v kapitole 2.6. DDD se také snaží popsat kód tak, aby obsahoval pojmy používané v běžné řeči mezi programátory a lidmi, kteří se starají o byznys stránku aplikace. Díky tomu je kód čitelnější a jednodušší na pochopení.

Entity, hodnotové objekty a služby jsou koncepty, popsané v DDD. Tyto koncepty popisují, jak by měli vypadat jednotlivé části doménového modelu. Další částí DDD jsou agregáty, které popisují modelování skupin entit a hodnotových objektů. Poslední částí DDD popsané v této práci jsou Repozitáře, které slouží k přístupu do databáze.

2.7.2 Sdílený slovník

Na vývoji aplikace se obvykle podílí více skupin lidí. Programátoři, doménoví experti[3], manažeři a další. Každá z těchto skupin mluví trochu jiným jazykem a kód obsahuje terminologii používanou pouze programátory. Programátoři pak musí jazyk zachycený v aplikaci překládat pro doménové experty a další skupiny lidí. Zároveň je potřeba překládat řeč doménových expertů zpět do jazyka programátorů. Při překladu se ale může stát, že se ztratí myšlenka, kterou doménový expert vyjádřil a poté kód neodpovídá reálnému světu.

Řešením tohoto problému je Sdílený slovník [22]. Sdílený slovník obsahuje terminologii dohodnutou mezi všemi skupinami, které se podílejí na vývoji aplikace [6]. Byznys logika aplikace by měla obsahovat pouze slova obsažená ve sdíleném slovníku a žádné odborné pojmy. Pokud doménový expert nebo vývojáři zjistí, že sdílený slovník je nevyhovující nebo nedostatečně popisuje problém, který se snaží aplikace modelovat, je potřeba změnit tento slovník [6]. Změna slovníku by pak měla vést véstovídající změně v kódu aplikace [6].

2.7.3 Entity

Při řešení doménových problémů často narazíme na koncepty, o kterých doménový expert bude mluvit jako o věcech s identitou [31]. Tento koncept se v DDD nazývá entita. Příklad uvedený v knize *Patterns, principles, and practices of domain driven design* [31] skvěle demonstruje, jak může entita vypadat. Aplikace na správu hotelů modeluje byznys požadavek: „Jeden uživatel může být ubytován pouze v jednom hotelu v jeden čas. Hotely můžou mít stejné jméno a všechny ostatní vlastnosti.“ Tento požadavek nepřímo říká, že hotel musí mít identitu a bude v aplikaci odpovídat entitě. V objektově orientovaných jazycích jsou entity obvykle modelovány pomocí objektu s atributem Id. Příklad entity hotelu je na ukázce 2.12. Kromě identity můžeme entity rozpoznat tak, že mají pokračující životní cyklus [6]. Pokud v systému existuje objednávka, se kterou bude uživatel opakovaně pracovat, je pravděpodobné, že objednávka by měla být entitou.

Entity mohou obsahovat metody a atributy, které odpovídají vlastnostem objektu, který se entita snaží modelovat. Avšak je důležité zmínit, že entita ani DDD obecně neslouží k přesnému zachycení reálného světa, proto entity nemusí obsahovat všechny vlast-

[3] Doménový expert je osoba, která má speciální znalost nebo schopnost v konkrétní oblasti snažení [20]. Například účetní je odborník v oblastí účetnictví.

Hotel

- id:int
- name:string
- teacherId:int

+ Hotel(id:int, name:string)
+ Book(from:DateTime, to:DateTime, numberOfPeople:int): bool

Obrázek 2.12: Ukázka zjednodušené entity `Hotel`.

nosti, které má reálný objekt [31]. Je postačující, aby entita obsahovala pouze vlastnosti, které jsou důležité pro řešení daného problému [31].

2.7.4 Hodnotové objekty

Mnoho objektů, které se v doméně vyskytují, nemají identitu, ale jsou identifikovány pomocí hodnot. Tyto objekty se nazývají hodnotové objekty [41]. Příklad takových objektů můžeme vidět v ukázce 2.2. Hodnotové objekty jsou obvykle neměnné, jelikož jsou identifikovány pomocí hodnot a změna těchto hodnot by vedla ke změně identity celého objektu. Proto jsou hodnotové objekty implementovány jako neměnné a v případě potřeby změnit objekt se vytvoří nový se změněnou hodnotou.

Hodnotové objekty se obvykle používají na místo primitivních typů. Například místo předávání jména uživatele jako řetězec je možné vytvořit hodnotový objekt, který zapouzdřuje daný řetězec. Použitím hodnotového objektu reprezentující jméno uživatele můžeme explicitně vyjádřit, co řetězec reprezentuje. Ukázku explicitního a implicitního vyjádření konceptu můžeme vidět na ukázce 2.3.

2.7.5 Služby

Některá funkcionalita v aplikaci nemusí patřit do entit nebo hodnotových objektů, pro takové případy se v DDD používají bezstavové třídy s názvem Služby [6]. Služby obsahují pouze metody. Metody odpovídají nějaké operaci, kterou aplikace potřebuje provést, například odeslání emailu. Služby se dělí na tři druhy.

- Doménové Služby (Domain Services) — jsou součástí domény aplikace a odpovídají konceptům popsaným ve Sdíleném slovníku.
- Aplikační Služby (Application Services) — přijímají požadavky od klienta a koordinují doménovou logiku tak, aby provedla požadavek klienta. Aplikační Služby můžeme chápat jako adaptér mezi klientem a doménou aplikace. Adaptéry byly popsány v kapitole 2.6. Mezi běžné zodpovědnosti patří Autentizace, Autorizace, navazování spojení s databází a ukončování spojení s databází [31].
- Služby Infrastruktury (Infrastructure Services) — starají se o procesy, které nejsou součástí domény aplikace. Příkladem může být odesílání emailu. Tyto služby jsou implementovány stejně jako sekundární adaptéry popsané v kapitole 2.6.

```
public class EmailAddress {
    public string Email { get; private set; }

    public EmailAddress(string email) {
        if(!ValidateEmail(email)) {
            throw new ArgumentException("Invalid email");
        }
        this.Email = email;
    }

    private bool ValidateEmail(string email) {
        // Overeni zda je email validni
    }

    public string GetEmailDomain() {
        // Vrati cast emailu ktera je za zavinacem
    }
}
```

Výpis 2.2: Hodnotový objekt reprezentující e-mailovou adresu. Objekt při vytváření příjme řetězec, který reprezentuje emailovou adresu. Poté zkontroluje, zda je adresa platná a na základě toho buď vyhodí výjimku, nebo uloží e-mail. Hodnotový objekt také obsahuje metodu `GetEmailDomain`, která vrací část e-mailu za zavináčem.

```
bool IsUserOldEnough(Customer customer) {
    var time = DateTime.UtcNow.AddYears(-18);
    var birthYear = customer.dayofbirth.Year;
    return (time - birthYear) >= 15;
}
//explicit
bool IsUserOldEnough(Customer customer) {
    return customer.Age.AgeAt(DateTime.Now) >= 15;
}
```

Výpis 2.3: Ukázka implicitního a explicitního vyjádření konceptu. V první verzi metody `IsUserOldEnough` se nikde přímo nevyskytuje věk uživatele, ale tento koncept v doméně aplikace existuje. Takové vyjádření se nazývá implicitní. Explicitní vyjádření konceptu je ukázáno ve druhé metodě na řádku č. 7. Věk uživatele je vyjádřený pomocí hodnotového objektu `Age`, který je obsažen v entitě `Customer`.

2.8 Agregáty

Entity i hodnotové objekty by měli zajišťovat své invarianty. To znamená, že každá entita a hodnotový objekt by měly být vždy ve validním stavu z pohledu byznys pravidel. Tato byznys pravidla jsou obvykle kontrolována při vytváření a editaci objektů. Příklad kontroly invariant ukazuje obrázek 2.2.

Problém nastane pokud je potřeba zajistit invarianty mezi více entitami. Tento problém demonstruje ukázka 2.4. Pro řešení tohoto problému DDD zavádí koncept agregátu [41]. Agregát je skupina objektů, jejichž invarianty jsou zajištěny díky tomu, že všechny operace s entitami v agregátu procházejí přes kořen agregátu. Kořen agregátu je vždy entita. V předchozím příkladu by tedy entita `Order` byla kořenem agregátu a ten by obsahoval metodu, která by měnila množství některého z Item. Příklad použití agregátu pro zamezení překročení ceny je ukázán na ukázce 2.5.

V některých případech je nutné, aby kořen agregátu obsahoval odkaz na jiný kořen agregátu, v takovém případě je potřeba odkazovat na agregát pouze pomocí globálního identifikátoru namísto reference. Entity a hodnotové objekty uvnitř agregátu také mohou odkazovat na jiné kořeny agregátů pomocí identifikátoru.

Modelování agregátů je velice obtížné a programátoři mnohdy dělají chybu, ve které vytvoří jeden velký agregát obsahující všechny entity a hodnotové objekty. Je důležité si uvědomit, že agregáty slouží pouze pro zajištění invariant v systému a jen málo kdy se stává, že agregáty obsahují více než tři entity [35]. Většina agregátů obvykle obsahuje pouze kořen agregátu a několik hodnotových objektů [35]. Více informací o modelování agregátu je možné najít zde [34].

Repozitáře

Repozitáře jsou objekty, které se starají o ukládání a načítání agregátů z databáze. Repozitáře by měly umožňovat získání pouze kořene agregátu, [6, Repositories] jelikož přístup k jednotlivým entitám a hodnotovým objektům by mohl vést k porušení invariant agregátu. Repozitář tedy běžně obsahuje metody pro získání, smazání, aktualizaci a vytvoření agregátu. V některých případech může repozitář obsahovat také metody sloužící pro filtrování podle daných podmínek.

2.9 Cibulová architektura

Cibulová architektura rozšiřuje Hexagonální architekturu o koncepty z DDD. Jak ukazuje obrázek 2.13, byznys logika není závislá na žádné části aplikace a všechny závislosti směřují ke středu. Toto obrácení závislostí je stejné jako v Hexagonální architektuře. Cibulová architektura dále popisuje čtyři vrstvy — služby infrastruktury, aplikační služby, doménové služby. Tyto služby jsou stejné jako ty popsané v předchozí podkapitole o DDD 2.7.1. Poslední částí Cibulové architektury je Doménový model [25]. Doménový model je část aplikace, která obsahuje entity, hodnotové objekty.

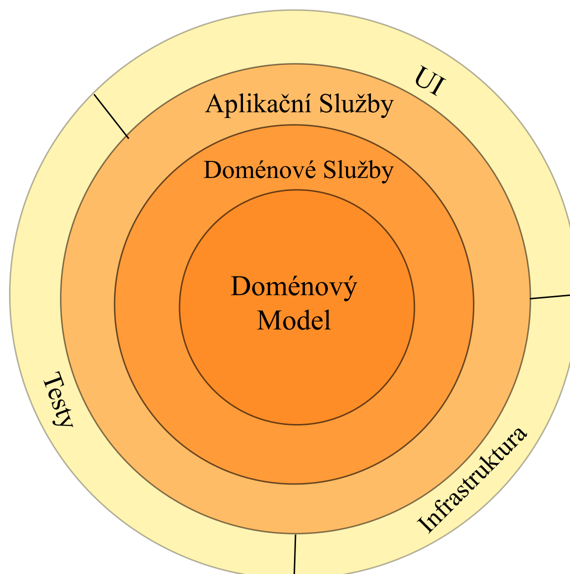
Obrázek 2.13 ukazuje jak vypadá pohled na celou architekturu. Jednotlivé části této architektury byly popsány v podkapitolách 2.7 a 2.7.1. Martin Fowler [10], Eric Evans [6] i Jeffrey Palermo [26] se shodují že všechny vrstvy v této architektuře jsou otevřené. Otevřené vrstvy popisuje podkapitola 2.2.2. Při zpracování požadavku se tedy může stát, že některé vrstvy budou přeskočeny.

```
public class Order {
    public int Id { get; private set; }
    public IReadOnlyList<OrderLineItem> Items
                                    { get => items.AsReadOnly(); }
    private OrderLineItem> items;

    public void AddItem(OrderLineItem orderLineItem, int amount) {
        ValidateTotalPrice(orderLineItem);
        items.Add(orderLineItem);
    }
    public void ValidateTotalPrice(OrderLineItem newItem) {
        int totalPrice = items.Sum(x => x.TotalPrice);
        if (totalPrice + newItem.TotalPrice > 1000) {
            throw new InvalidOperationException(
                "Cena objednavky nesmi presahnout 1000 Kc");
        }
    }
}

public class OrderLineItem {
    public int Price { get; private set; }
    public int Amount { get; private set; }
    public int TotalPrice { get; private set; }

    public void ChangeAmount(int newAmount) {
        Amount = newAmount;
        TotalPrice = newAmount * Price;
    }
}
```

Výpis 2.4: Ukázka porušení invariant. V této aplikaci nesmí cena objednávky nikdy přesáhnout 1000 Kč. Při přidání nového zboží do objednávky metodou `AddItem` může objednávka zkontrolovat, zda celková cena nepřekročila 1000Kč. Avšak u metody `ChangeAmout` nemá `Order` ani `Item` možnost kontroly maximální ceny celé objednávky. Proto může v některých případech dojít k porušení invariant. Z ukázky byly pro jednoduchost odstraněny validace a konstruktor pro objekt `Order`.

```
public class Order {
    public int Id { get; private set; }
    public IReadOnlyList<IReadOnlyOrderLineItem>
                    Items { get => items.AsReadOnly(); }
    public void AddItem(Item item, int amount) {
        OrderLineItem orderLineItem = item.CreateOrderLineItem(amount);
        ValidateTotalPrice(orderLineItem);
        items.Add(orderLineItem);
    }
    public void ChangeAmount(Item item, int amount) {
        var orderLineItem = items.FirstOrDefault(x => x.Id == item.Id);
        var originalAmount = orderLineItem.Amount;
        orderLineItem.ChangeAmount(amount);
        try {
            //null object pattern
            var itemWithPriceZero = OrderLineItem.GetEmptyItem();
            ValidateTotalPrice(itemWithPriceZero);
        }
        catch (InvalidOperationException) {
            orderLineItem.ChangeAmount(originalAmount);
            throw;
        }
    }
    public void ValidateTotalPrice(OrderLineItem newItem) {
        int totalPrice = items.Sum(x => x.TotalPrice);
        if (totalPrice + newItem.TotalPrice > 1000) {
            throw new InvalidOperationException(
                "Cena objednavky nesmi presahnout 1000 Kc");
        }
    }
}
```

Výpis 2.5: Ukázka řešení předchozího problému pomocí agregátu. Položka `OrderLineItem` byla rozdělena na dva objekty. `Item` reprezentující předmět v obchodě a `OrderLineItem` reprezentující předmět již vložený do košíku. Tyto entity se liší například tím, že `Item` neobsahuje počet kusů. Dalším rozdílem proti ukázce 2.4 je, že `Order` obsluhuje vytváření entity `OrderLineItem`, díky tomu nikdo mimo agregát nemůže přistupovat k entitám `OrderLineItem` po vložení do košíku. Další změnou je kolekce, přes kterou lze přistoupit k jednotlivým položkám. Tato kolekce nyní obsahuje pouze rozhraní, které zveřejňuje pouze metody pro získání hodnot z `OrderLineItem`. Z ukázky byly pro jednoduchost odstraněny validace a konstruktor pro objekt `Order`.



Obrázek 2.13: Ukázka Cibulové architektury. Všechny závislosti v této architektuře směřují do středu.

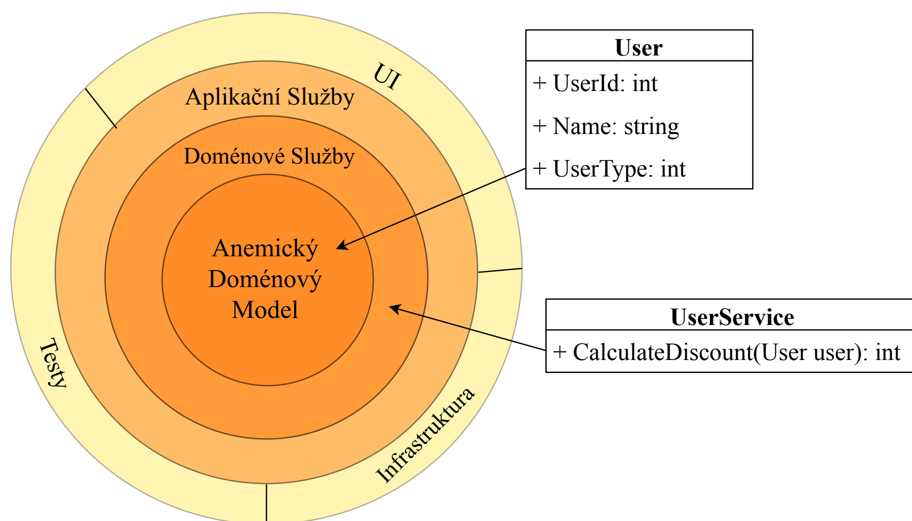
2.10 Anemický doménový model

Při implementaci architektury používající doménový model se může stát, že programátor přesune všechnu byznys logiku do doménových služeb a díky tomu se z entit stanou pouhé balíčky dat bez funkcionality [8]. Takový doménový model se nazývá anemický. Aplikace používající Anemický doménový model se pouze tváří objektově, ale ve skutečnosti není [8]. Příklad Anemického doménového modelu je ukázán na obrázku 2.14.

Anemický doménový model může v některých případech používat entity, hodnotové objekty a agregáty. Jelikož ale entity a hodnotové objekty Anemického doménového modelu nemají metody, nemá tento způsob modelování byznys logiky velký přínos [8]. Z tohoto důvodu mnoho implementací Anemického doménového modelu používá datové objekty, které přesně odpovídají databázovým tabulkám.

Repozitáře v DDD načítají celé agregáty, avšak pokud Anemický doménový model nepoužívá koncept agregátů, je potřeba změnit i implementaci repozitářů. V implementaci bez agregátů repozitáře odpovídají datovým objektům. Každý datový objekt má přiřazený jeden repozitář, který se stará o jeho načítání, ukládání, aktualizaci a mazání. Repozitáře také obvykle obsahují další operace s databází.

Při implementaci architektury Anemického doménového modelu je také možné vynechat aplikační vrstvu a použít pouze doménové servisy, které budou volány přímo z prezentační vrstvy. Použití aplikační vrstvy je ale odlišné od použití v Cibulové architektuře. V Cibulové architektuře obvykle aplikační vrstva načte agregát z repozitáře a poté na něm provede potřebné operace. Bez existence agregátů v Anemické modelu je ale potřeba k provedení operace načíst několik datových objektů, které jsou pak předány doménové službě, která provede potřebné operace. Použití aplikační vrstvy ale není příliš časté například žádný z oblíbených frameworků pro e-shop v jazyce C# nepoužívá aplikační vrstvu [24, 32, 13]. Architektura Anemického doménového modelu má tedy mnoho podob a nejpoužívanější variantou je Anemický doménový model s datovými třídami a bez aplikační vrstvy.



Obrázek 2.14: Ukázka Architektury používající Anemický doménový model. Architektura na obrázku je podobná Cibulové architektuře, ale mohla by se i více podobat architektuře používající Transakční skript. Transakční skript byl popsán v kapitole 2.4.2. Hlavním rozdílem oproti Cibulové architektuře je, že Doménový model obsahuje pouze data a všechna byznys logika pracující s daty je obsažena v doménových službách. Aplikační služby se v této ukázce starají o věci, které nejsou pro byznys logiku důležité, stejně jako v DDD. Aplikační služby byly popsány v kapitole 2.7.5. Na obrázku je příklad služby, který počítá slevu pro daného uživatele. Šipky naznačují, do kterých vrstev třídy patří.

Kapitola 3

Návrh demonstrační aplikace

Tato kapitola navrhuje demonstrační aplikaci tak, aby na ni bylo možné provést porovnání dále vybraných architektur. Navržená aplikace se snaží co nejlépe demonstrovat výhody a nevýhody architektur. K tomu potřebuje být aplikace dostatečně složitá. Zároveň ale musí být dostatečně jednoduchá, aby ji bylo možné implementovat jednou pro každou z dále vybraných architektur.

Z pohledu softwarových architektur nemá význam, aby aplikace obsahovala grafické rozhraní, proto byla aplikace implementována pouze jako webové API. Pro ověření správnosti implementace byly vytvořeny integrační testy a pokud to architektura umožňovala byly implementovány i jednotkové testy. Tato kapitola dále popisuje aplikaci a její byznys pravidla. V následující části jsou pak vybrány architektury, na kterých bude demonstrována aplikace.

3.1 Školní systém

Pro demonstraci architektur byl vybrán informační systém pro vysokou školu. Tento systém byl vybrán, jelikož může obsahovat velké množství komplikovaných byznys pravidel, která dávají smysl v reálném systému. Následující seznam popisuje byznys pravidla tak, jak by o nich mohl mluvit zákazník, který potřebuje vytvořit aplikaci školního systému.

- Studenti si mohou zaregistrovat předměty. Předměty mohou být registrovány pouze v předem určených časech. Tyto časy jsou určeny školou. Výraz „registrace otevřena“ se často používá pro označení času, kdy si studenti mohou registrovat předměty.
- Čas, kdy si student může zaregistrovat předmět, se mění podle studentova průměru. Typickým příkladem je, že studentovy s dobrým průměrem se otevře registrace dříve. Přesné časy jsou aktuálně neznámé, až bude systém spuštěn, bude se testovat, jaký průměr je vhodný pro jaký čas.
- Každý předmět má přiřazený počet kreditů. Počet kreditů za předmět určuje škola.
- Rok se pro jednoduchost nedělí na dva semestry.
- Student si může zaregistrovat maximálně 60 kreditů na rok
- Škola může udělit titul studentovi, který má více než 180 kreditů.
- Škola může ukončit studentovo studium, pokud mu v minulosti nebyl udělen titul.

- Škola může upravovat informace o studentech. O změnách studentů, kteří byly v minulosti promováni nebo vyloučeni, musí být informován děkan e-mailem. Stačí odeslat pouze informaci, který student byl upraven.
- Škola může přijímat nové studenty.
- Škola může vytvářet, editovat, mazat a upravovat předměty.
- Každý předmět musí mít přiřazeného alespoň jednoho učitele.
- Při změně kreditů u předmětu se nesmí změnit počet kreditů pro studenty, kteří tento předmět v minulosti absolvovali. Studentům, kteří mají tento předmět aktuálně zapsaný, se změní počet kreditů. Studentům, kteří překročí kvůli této změně maximální počet kreditů, se musí změněný předmět odhlásit.

Architektury vybrané pro demonstraci

Pro implementaci demonstrační aplikace bylo vybíráno z architektur popsaných v předchozí kapitole. Následující seznam shrnuje všechny architektury popsané v předchozí kapitole.

- Vrstvená architektura
- Chytré UI
- Architektura transakčních skriptů
- Architektura aktivních záznamů
- Hexagonální architektura
- Cibulová architektura
- Architektura anemického doménového modelu

První vybranou architekturou je Cibulová, jelikož je jediná používající objektově orientované programování. Cibulová architektura také používá Hexagonální architekturu, můžeme tedy obě dvě považovat za vybrané. Architektura Aktivních záznamů je další vybranou architekturou, jelikož je velice populární a používá jí několik oblíbených framewokrů. Více informací je v kapitole 2.5. Architektura aktivních záznamů používá vrstvenou architekturu, proto můžeme i tu považovat za vybranou.

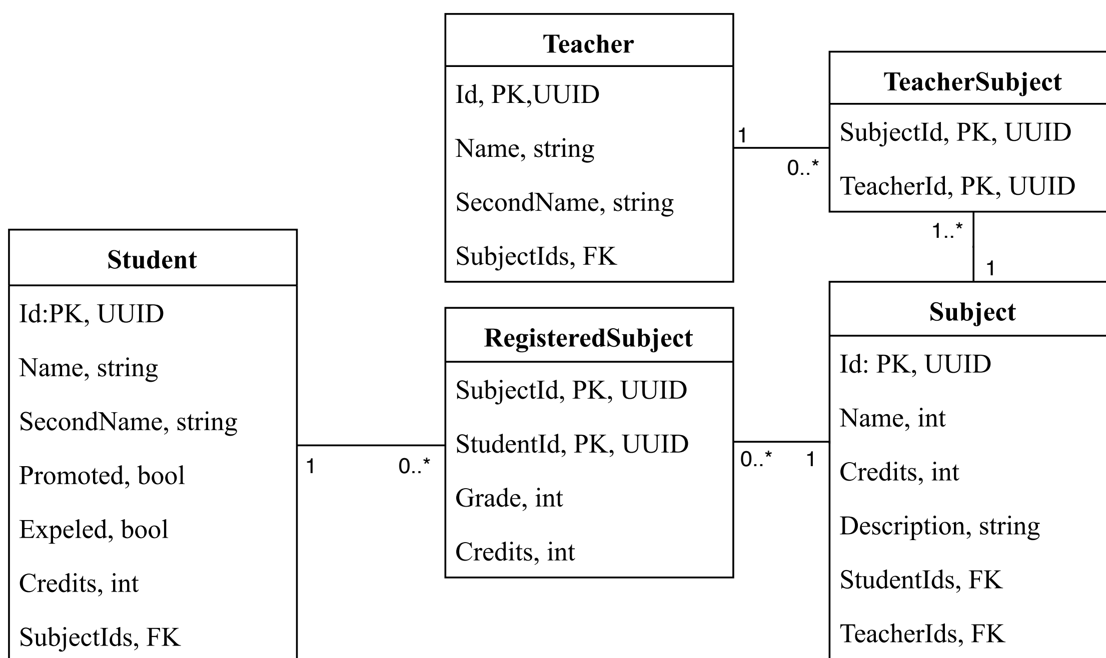
Ze tří zbývajících architektur byla vybrána pouze Architektura anemického doménového modelu. Anemický doménový model má mnoho variant popsaných v kapitole 2.10 a kvůli své popularitě byla vybrána varianta, ve které jsou použity datové objekty odpovídající tabulkám v databázi bez aplikační vrstvy. Tato varianta architektury je velice podobná Architektuře transakčních skriptů. Z tohoto důvodu a také kvůli nízké popularitě nebyla Architektuře transakčních skriptů vybrána.

Poslední architekturou je Chytré UI. Chytré UI nebylo vybráno, jelikož jeho obvyklé použití je velice odlišné od ostatních architektur. Všechny architektury popsané v této práci se obvykle používají na komplikované aplikace. Chytré UI se ale obvykle používá na jednoduché aplikace. Porovnání tedy nemá velký význam, jelikož rozhodnutí mezi použitím Chytrého UI nebo jinou architekturou je poměrně jednoduché. Toto rozhodování bylo krátce popsáno v kapitole 2.3.

3.2 Návrh aplikace v Architektuře aktivního záznamu

Architektura aktivních záznamů má dvě varianty popsané v kapitole 2.5. Pro implementaci ukázkové aplikace byla zvolena varianta, ve které řadiče neobsahují byznys logiku. Namísto toho je byznys logika obsažena v Transakčních skriptech. Jsou tedy použity tři vrstvy vrstvené architektury tak, jak jsou popsány v kapitole 2.2. Aktivní záznamy by podle definice mohly obsahovat byznys logiku pro demonstraci, ale byla zvolena varianta ve které žádnou neobsahují. Tím se zachová rozdělení zodpovědností vrstvené architektury.

Architektura aktivního záznamu je z velké části modelována podle toho, jak vypadá návrh relační databáze. Diagram databáze pro školní systém je ukázán na obrázku 3.1. Pro implementaci této architektury byl zvolen programovací jazyk Python a framework Django, jelikož C# nemá velkou podporu Aktivních záznamů.



Obrázek 3.1: Databázový diagram školního systému.

3.3 Návrh aplikace v Cibulové architektuře

Cibulová architektura byla popsána v podkapitole 2.9. Jednou z hlavních částí této architektury je použití DDD. Důležitou částí DDD je sdílený slovník popsaný v podkapitole 2.7.2, který obsahuje slova sdílená všemi skupinami podílejícími se na vývoji systému. Entity, hodnotové objekty a doménové služby jsou poté pojmenovány podle pojmů ze sdíleného slovníku. Následující seznam ukazuje slova ze sdíleného slovníku vybraná z předchozího popisu aplikace. Některá dále uvedená slova se nevyskytují v předchozím popisu, avšak pro potřeby této práce budeme předpokládat, že byla přidána v dalších diskuzích s doménovým expertem. V závorkách jsou uvedeny anglické překlady.

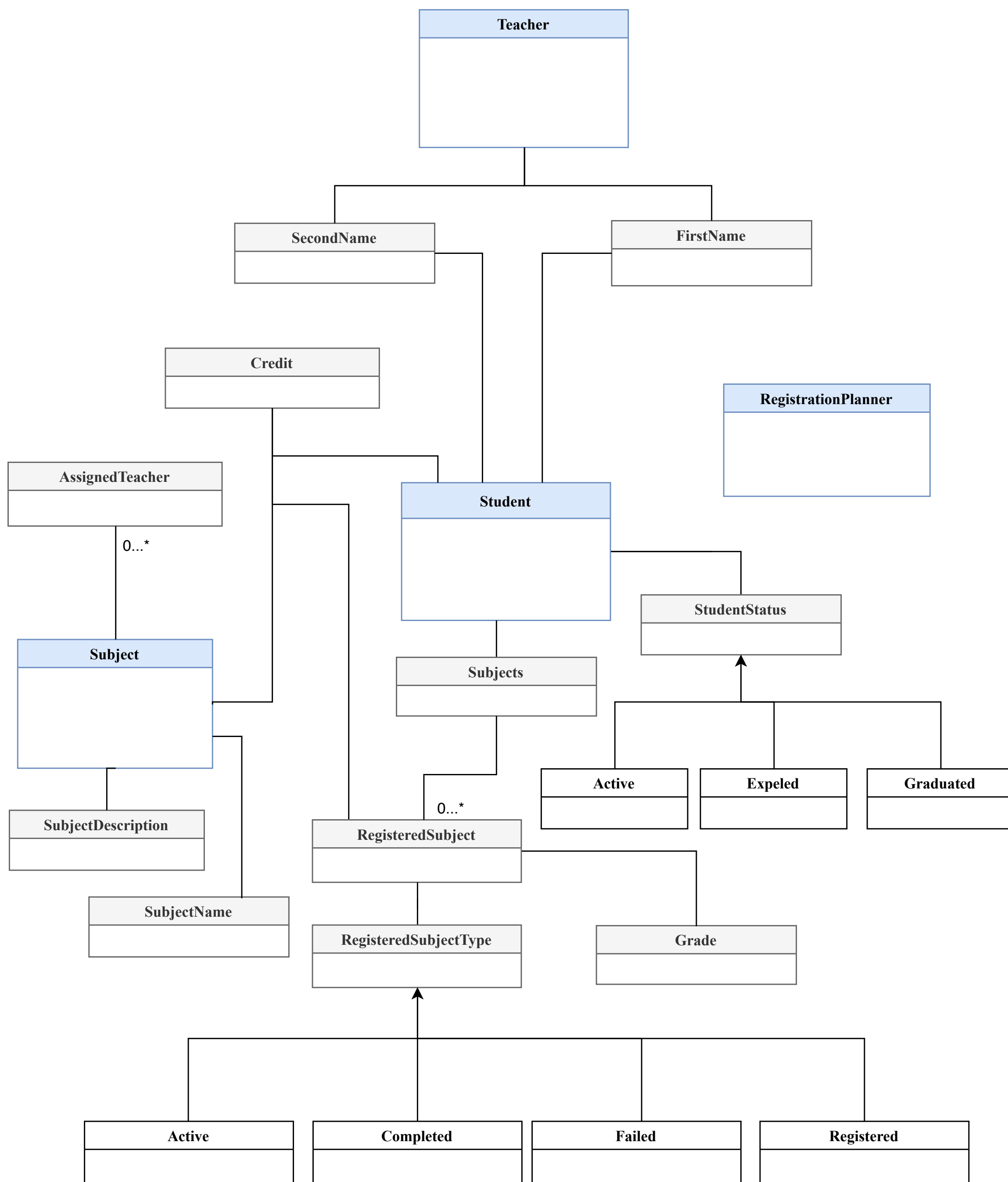
- Student (student),
- předmět (subject),
- učitel (teacher),
- škola (school),
- kredit (credit),
- registrace (registration),
- kategorie studenta (student category),
- promovat (graduate),
- ukončit studium (fail),
- stav (status) — určuje, zda student získal titul nebo byl vyloučen nebo stále studuje,
- kategorie (category) — určuje kategorii studenta podle průměru.

Doménový model školního systému

Dalším krokem je vytvoření návrhu aplikace, který používá sdílený slovník. Pro větší aplikace by mohl systém být rozdělen do více částí a každá část modelována samostatně, avšak v této jednoduché ukázce existuje v systému pouze jedna část, která je ukázaná na obrázku 3.2. V této části modelování je důležité nalézt agregáty, entity a hodnotové objekty. Tyto koncepty byly popsány v podkapitolách 2.8, 2.7.3 a 2.7.4. Po identifikování těchto konceptů již je možné implementovat aplikaci, jelikož už máme přesný model systému.

3.4 Návrh aplikace v Architektuře anemického modelu

Architektura anemického doménového modelu má mnoho variant popsaných v podkapitole 2.10. Pro demonstrační implementaci byla vybrána varianta s datovými objekty odpovídající databázovým tabulkám bez aplikační vrstvy. Tato varianta byla vybrána kvůli její popularitě. Pro tuto architekturu je stěžejní databázový model. Databázový model je stejný jako v Architektuře aktivních záznamů ukázaný na obrázku 3.1.



Obrázek 3.2: Diagram entit a hodnotových objektů školního systému. Modře jsou vyznačeny kořeny agregátů a šedě hodnotové objekty.

Kapitola 4

Implementace vybraných architektur

Implementace sebou nese mnoho problémů, které teoretický popis architektur nedokáže vyřešit. Často jsou to problémy vytvořeny omezeními programovacího jazyka nebo použitého frameworku. Následující kapitoly popisují ty nejzajímavější problémy, které se vyskytnou při implementaci vybraných architektur.

Nejvíce implementačních problémů sebou nese Cibulová architektura. Na rozdíl od zbylých dvou architektur musí programátor dobře znát objektové programování. Zároveň programátor musí dobře znát ORM, které používá, jelikož se setká s obtížemi při mapování doménového modelu na databázový. Implementace Architektury anemického doménového modelu a aktivního záznamu se zdají být jednodušší než implementace Cibulové architektury.

4.1 Uplatňování byznys pravidel a validací

Každá byznys aplikace určitým způsobem kontroluje, zda jsou splněny všechny podmínky potřebné pro vykonání byznys operace. Tuto kontrolu můžeme rozdělit na dvě části — validace a kontrola byznys pravidel. Validace kontrolují, zda jsou vstupní data platná a byznys pravidla poté kontrolují, zda je operace platná z byznys pohledu. Obvykle je však obtížné rozlišit, jaké operace jsou důležité z byznys pohledu a proto je hranice mezi validacemi a byznys pravidly velice tenká. Z tohoto důvodu nemá pro tuto část práce význam rozlišovat mezi byznys pravidly a validacemi. V této kapitole tedy budou oba pojmy považovány za ekvivalentní a zaměnitelné.

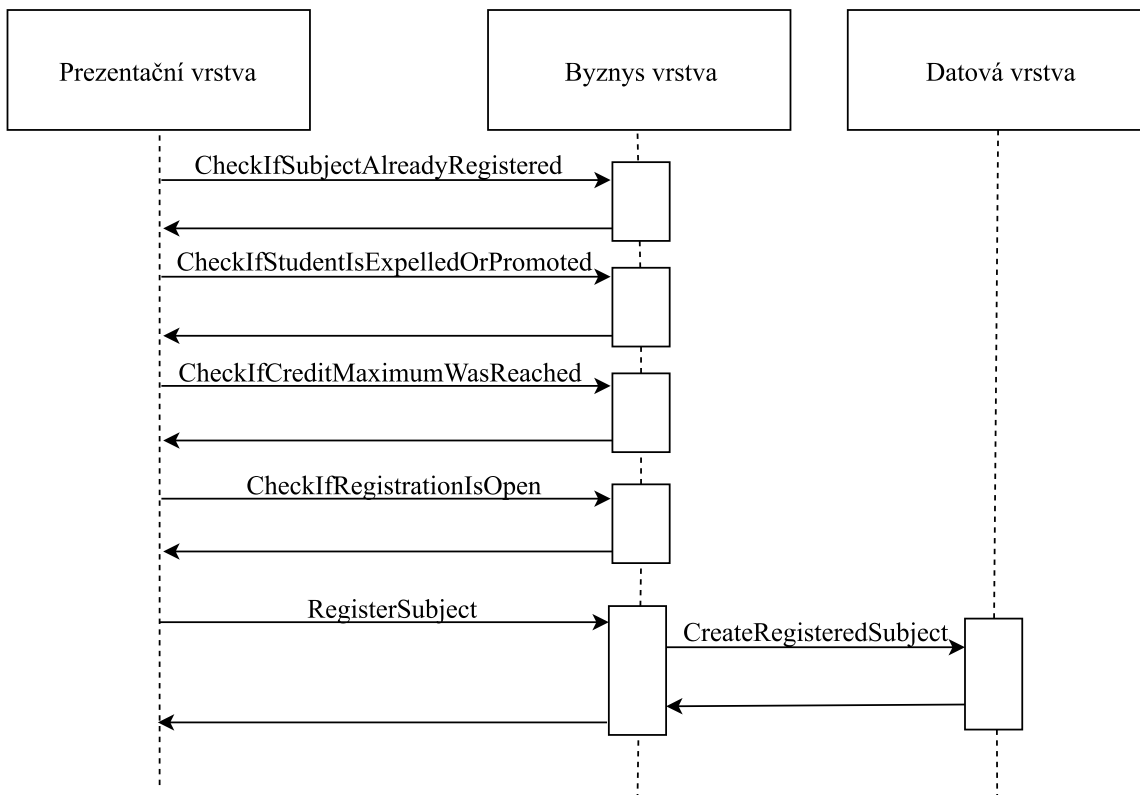
Validace je možné implementovat třemi způsoby — validace před provedením operace, validace uvnitř operace a kombinace těchto přístupů. Každá z těchto variant má výhody a nevýhody. Pro výběr mezi těmito variantami je tedy důležité zvážit složitost aplikace.

Pro zjednodušení následujícího textu je vhodné označit některé validace jako absolutní, takové validace musejí být vždy zkontrolovány jinak mohou způsobit selhání byznys operace. Příklad absolutní validace je kontrola, zda parametr neobsahuje hodnotu `null`. Jelikož hodnota `null` může způsobit vyhození výjimky, která způsobí neočekávané selhání operace.

Kontrola byznys pravidel před provedením operace

V případě této implementace prezenční vrstva nejdříve volá metody, které zkontrolují byznys pravidla a poté zavolá metodu, která provede byznys operaci samotnou. Sekvenční dia-

gram 4.1 ukazuje registraci předmětu studentem s použitím tohoto způsobu kontroly byznys pravidel. Validační metody leží v byznys vrstvě, stejně jako byznys operace samotná. Díky tomu se žádná byznys logika nedostává mimo byznys vrstvu aplikace.



Obrázek 4.1: Sekvenční diagram ukazující registraci předmětu studentem s kontrolou byznys pravidel před provedením byznys operace. Před provedením operace se provedou různé kontroly, jako například kontrola, zda student neregistruje předmět podruhé.

Díky oddělení validací od byznys operací může aplikace používat různé validace podle kontextu, ze kterého je operace volána. Příkladem může být registrace uživatele pomocí administrace a pomocí běžně přístupné webové stránky. Běžná registrace často požaduje jiné validace než registrace uživatele pomocí administračního rozhraní. Díky flexibilitě, kterou přináší kontrola byznys pravidel před provedením operace, může aplikace jednoduše zvolit různé validace na základě kontextu.

Problémem validace před provedením byznys operace je, že programátor nemá žádnou možnost, jak zjistit, které validace jsou absolutní. Jedinou možností je zjistit, jak funguje celá byznys operace. Před použití každé operace je tedy nutné zjistit její absolutní validace a ty zkontrolovat. Je zřejmé, že tento přístup k validaci snižuje znovupoužitelnost kódu.

Kontrolu byznys pravidel je také možné provést uvnitř objektu, který je poté předán jako parametr byznys operace. Příkladem tohoto přístupu jsou hodnotové objekty, které v konstruktoru kontrolují svá byznys pravidla. Ukázka takového hodnotového objektu je na obrázku 2.2 v kapitole o hodnotových objektech 2.7.4. Tento přístup je velice elegantní, jelikož jsou všechny validace přesně tam, kde by je čtenář programu očekával. Je velice intuitivní, když objekt `UserName` validuje, jaká pravidla musí splňovat uživatelské jméno.

Nevýhodou tohoto přístupu je, že byznys operace nemůže být volána z různých kontextů s různými byznys pravidly. Další nevýhodou je, že změna byznys pravidel v hodnotovém objektu může způsobit selhání některé z byznys operací.

Kontrola byznys pravidel uvnitř byznys operace

Přesunutí validací do byznys operace vede k lepší znovupoužitelnosti, jelikož se operace nikdy nemůže dostat do neočekávaného stavu. Nevýhodou ale je, že není možné použít operaci z jiných kontextů s různými validacemi.

Kontrola byznys pravidel uvnitř i před byznys operací

Další možností je kombinace předchozích variant. Před provedením operace aplikace zkontroluje neabsolutní pravidla a poté uvnitř operace zkontroluje absolutní pravidla. Tato varianta řeší všechny předchozí problémy za cenu obtížné implementace. Implementace této varianty je obtížná, jelikož programátor musí validace provádět na dvou místech a jejich výsledek zkombinovat tak, aby mohl být společně prezentován uživateli. V některých případech také může dojít k duplikaci validací, kvůli nepozornosti programátora.

Zhodnocení

Jaký způsob validace zvolit záleží hlavně na velikosti projektu. První dva způsoby jsou vhodné pro menší projekty a naopak poslední způsob umožňuje velkou flexibilitu za cenu vyšší komplexity. Demonstrační aplikace používají první dva způsoby validace. Validace před provedením byznys operace je ukázána v aplikaci používající Architekturu aktivního záznamu. Validace uvnitř byznys operací je ukázána na Architektuře anemického doménového modelu. Validace uvnitř hodnotových objektů, tedy před provedením byznys operace, je demonstrována na Cibulové architektuře.

4.2 Mapování doménového modelu na databázový pomocí Entity frameworku

Jedním z problémů implementace doménového modelu je mapování na databázový model. Databázový model často neodpovídá problému, který se doménový model snaží modelovat a ORM slouží jako mezi vrstva, která mapuje mezi těmito modely. Pro mapování v demonstrační aplikaci bylo použito nejznámější ORM v C# s názvem Entity framework. Entity framework se v minulosti zaměřoval především na mapování datových modelů přesně odpovídajících databázi, ale v posledních verzích začal podporovat i složitější scénáře potřebné pro mapování doménových modelů. Tato část dále popisuje řešení nejčastějších problémů, které je potřeba vyřešit při mapování doménového modelu na databázový.

4.2.1 Mapování privátních atributů

DDD popsané v kapitole 2.7.1 říká, že doménový model by neměl znát způsob, jakým je mapován na databázový model. Doménový model by měl obsahovat pouze věci, které jsou důležité z pohledu domény aplikace. Avšak toto pravidlo musí být v některých případech porušeno, aby bylo možné uložit doménový model do databáze. Entity framework umožňuje mapování privátních atributů do databáze, neumožňuje ale způsob, jak se na privátní

```
1  public class Student : Entity {
2      private List<RegisteredSubject> _subjects { get; }
3                                      = new List<RegisteredSubject>();
4
5      public static class DatabaseOperations {
6          public static
7              Expression<Func<Student, List<RegisteredSubject>>>
8              IncludeRegisteredSubjects(){
9              return (Student x) => x._subjects;
10         }
11     }
12     db.student.Include(
13                 Student.DatabaseOperations.IncludeRegisteredSubjects());
14 }
```

Výpis 4.1: Zjednodušená ukázka tabulky student a operace `join` na tabulkách student a registrovaný předmět. Registrovaný předmět obsahuje cizí klíč na tabulku Student. Tento vztah je v ORM Entity Framework vytvořen pomocí seznamu registrovaných předmětů v objektu student. Vnořená statická třída `DatabaseOperations` zpřístupňuje lambda funkci, která má přístup k privátní proměnné `_subjects` a umožňuje tak SQL operace nad tímto atributem. Na závěr je ukázán příklad SQL operace `join` tabulek student a registrovaný předmět pomocí silně typovaného rozhraní LINQ.

proměnné dotazovat bez použití názvu proměnné jako řetězce. To způsobuje, že programátor může přejmenovat proměnou ve třídě, ale nezměnit řetězec v dotazu na databázi, což způsobí selhání dotazu. Elegantním řešením tohoto problému je vnořená statická třída. Vnořené třídy v C# mají přístup ke všem privátním proměnným a umožňují tak programátorovi přístup ke všem atributům. Ukázka 4.1 ukazuje implementaci vnořené třídy.

4.2.2 Mapování M:N vztahu

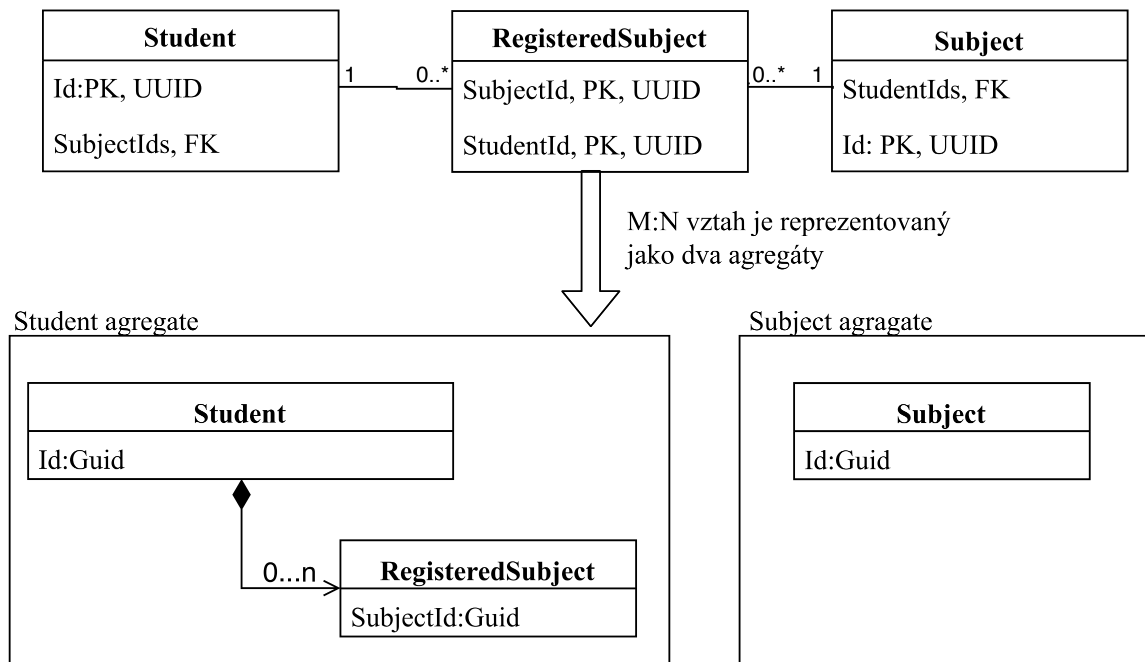
Při tvorbě doménového modelu se často stává, že spojovací tabulka vztahu M:N je reprezentována hodnotovým objektem oddělujícím dva agregáty. Diagram 4.2 ukazuje předmět, studenta a jejich spojovací tabulku `RegisteredSubject`. Hodnotový objekt popsaný v kapitole 2.7.4 by měl vždy podle definice být neměnný a neměl by obsahovat identifikátor. Pokud je ale nutné hodnotový objekt mapovat na spojovací tabulku ve vztahu M:N, je potřeba, aby měl identifikátor a také aby jeho hodnota mohla být změněna.

Tento ústupek ukazuje nevýhodu použití SQL databáze v kombinaci s doménovým modelem. V některých případech je mapování možné, ale obtížné. V takových případech je vhodné udělat ústupek databázi a upravit doménový model tak, aby se více podobal databázovému modelu, i přesto, že je to proti zásadám DDD.

Dalším řešením pro ukládání doménového modelu je použití dokumentové databáze, která dokáže velice elegantně uložit jednotlivé agregáty. Většina moderních dokumentových databází ale neumožňuje atomické transakce ve více než jednom dokumentu, což může v některých případech vést k problémům.

Poměrně nově se začínají objevovat relační databáze, které umožňují ukládat JSON. Tento způsob ukládání využívá například knihovna Marten [17], která se stará o ukládání

agregátů do databáze PostgreSQL. V demonstrační aplikaci ale byla použita klasická SQL databáze, jelikož Marten a další podobná řešení jsou stále poměrně nová.



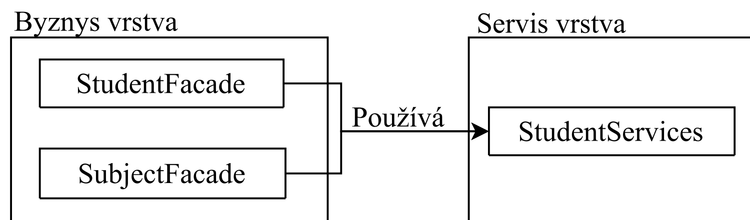
Obrázek 4.2: Vztah M:N reprezentovaný jako dva agregáty v objektovém modelu. V implementaci, která nedělá žádné ústupky kvůli mapování objektového modelu na databázový, by objekt `RegisteredSubject` měl být neměnný a neměl by obsahovat identifikátor.

4.3 Volání metod ze stejné vrstvy

Vrstvy ve vrstvené architektuře mohou být otevřené nebo uzavřené. Tato vlastnosti určuje, zda požadavky mohou obejít vrstvu nebo musí skrze vrstvu projít. Další informace o této vlastnosti jsou uvedeny v podkapitole 2.2.2. Otevřenost a uzavřenost je dobře zdokumentovaná v literatuře a proto je poměrně jednoduché rozhodnout, které vrstvy mají být uzavřené a které otevřené. Vrstvy mají ale i jinou velice důležitou vlastnost, která není téměř nikdy popisována. Tato vlastnost určuje, zda objekty v jedné vrstvě na sobě mohou mít závislost bez prostředníka, neboli přímou závislost. Přímá závislost je způsobena, pokud objekt volá metodu z jiného objektu.

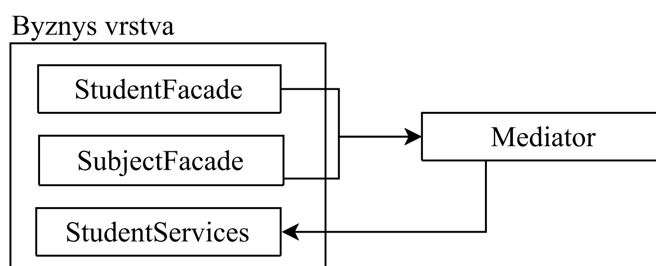
Přímá závislost mezi službami v jedné vrstvě může způsobit cyklickou závislost. Cyklické závislosti v kódu jsou obecně považovány za nežádoucí z mnoha důvodů. Více informací můžeme najít například zde [1]. Služby v jedné vrstvě by na sebe tedy nikdy neměli mít přímé závislosti. Avšak v některých případech je potřeba volat stejnou metodu z více míst v jedné vrstvě. Řešením je vyjmout sdílenou logiku do jiné vrstvy a tuto vrstvu volat z obou míst. Diagram tohoto oddělení je ukázán zde 4.3. Dalším řešením je použití návrhového vzoru prostředník. Řešení pomocí prostředníka ukazuje diagram 4.4.

Zda použít mediátor nebo vyjmutí služby do samostatné vrstvy záleží na architektuře a také na typu operace, kterou se snaží aplikace provést. Nevýhodou prostředníka je horší čitelnost kódu, jelikož služba, která prostředníka volá, nezná službu, která požadavek zpracuje. Při čtení kódu tedy není jasné, kdo operaci vykonává bez bližšího zkoumání prostřed-



Obrázek 4.3: Ukázka vyjmutí sdílené byznys logiky do servisní třídy.

níka. Výhodou prostředníka je možnost změny objektu, který operaci zpracovává za jiný bez nutnosti změny kódu, který volá prostředníka. Kterou variantu použít nelze obecně rozhodnout a je potřeba je posoudit pro každou operaci samostatně.



Obrázek 4.4: Ukázka použití prostředníka pro volání sdílené logiky.

Kapitola 5

Porovnání architektur

Porovnání architektur v této práci má za účel pomoci čtenářům vybrat, která z architektur je nejvhodnější pro jejich projekt. Pro splnění tohoto cíle byly vybrány následující čtyři metriky — *čitelnost*, *rozšiřitelnost*, *potřebné znalosti vývojářů* a *testovatelnost*. První tři metriky byly porovnávány na základě zpětné vazby od programátorů. Poslední metrika testovatelnost byla porovnána na základě informací o architekturách. Následující odstavce krátce popisují jednotlivé metriky a jejich zkoumání. Další části této kapitoly poté hlouběji popisují metriky a jejich testování.

Čitelnost a *rozšiřitelnost kódu* není možné přesně vypočítat algoritmem, proto bylo pro porovnání vybráno několik programátorů, kteří se snažili rozšířit demonstrační aplikaci o nová byznys pravidla a zároveň se vyznat v již existujících pravidlech. Na základě těchto pozorování pak byly architektury porovnány.

Potřebné znalost programátorů pro použití vybraných architektur byly porovnána na základě dotazníku, který programátoři vyplnili po školení na všechny vybrané architektury. Odpovědi z dotazníku pak ukázaly, která architektura je nejobtížnější na naučení.

Poslední zkoumanou metrikou byla *testovatelnost* pomocí jednotkových testů. Jednotkové testy jsou pro programátory důležité, jelikož umožňují rychle zkontrolovat, zda malé části aplikace fungují správným způsobem. Testovatelnost aplikace ale není obvykle ovlivněna architekturou, ale dodržením obecných principů psaní správného kódu. Výjimkou je Architektura aktivních záznamů, ve které obvykle není možné testovat byznys logiku pomocí jednotkových testů, jelikož je provázána s databázovou logikou aplikace.

5.1 Potřebné znalosti týmu vývojářů

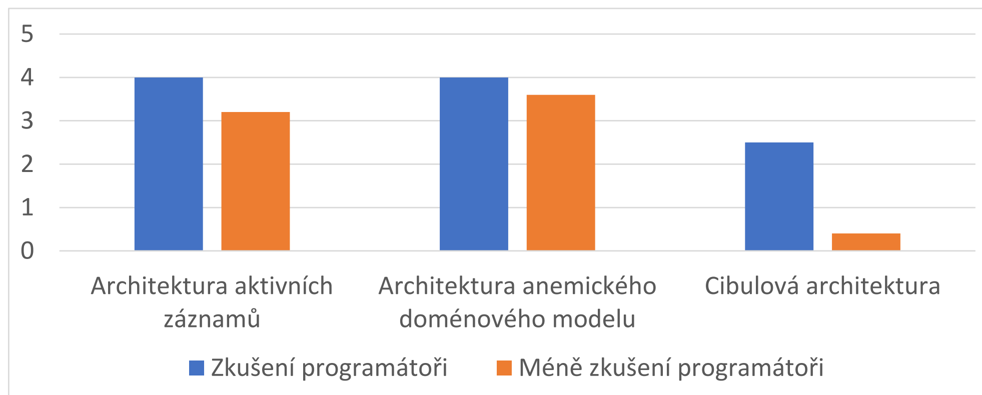
Už z popisu architektur je zřejmé, že Cibulová architektura vyžaduje nejzkušenější tým vývojářů. Cibulové architektury je obtížná na implementaci, jelikož vyžaduje tvorbu doménového a databázového modelu. Naopak Anemický doménový model a aktivní záznam vyžadují pouze databázový model. Pro porovnání architektur je důležité alespoň odhadnout, jak obtížné je pro tým vývojářů použít jednu z vybraných architektur. K vytvoření tohoto odhadu bylo vybráno 11 programátorů, kteří prošli školením o architekturách a poté vyplnili test, který ověřoval schopnost implementovat architektury.

Školení se snažilo napodobit reálné studium architektur programátory. Z tohoto důvodu bylo školení zaměřeno spíše teoreticky. Naopak následující test se snažil napodobit pokus o implementaci architektur, a proto byl zaměřen na praktické schopnosti. Příloha B obsahuje

otázky, správné odpovědi a výsledky programátorů. Při psaní testu bylo možné využívat internet, aby měli programátoři stejné možnosti jako při implementaci aplikace.

Programátoři byly rozděleni do dvou skupin pěti a šesti programátorů. První skupina obsahovala méně zkušené programátory a druhá skupina obsahovala pokročilé programátory. Pokročilí programátoři měli praktické zkušenosti s Architekturou anemického doménového modelu a méně zkušení programátoři neměli zkušenosti s žádnou z vybraných architektur.

Testové otázky byly hodnoceny pouze třemi hodnotami — správně, nesprávně a částečně správně. Částečně správně bylo uděleno za odpovědi, kterým chyběla pouze malá část do vyřešení zadaného problému. Tabulka 5.1 ukazuje výsledky obou skupin.



Obrázek 5.1: Graf ukazující průměrný počet bodů získaných z testu. Maximální možný průměr byl čtyři. Graf ukazuje, že Cibulová architektura je obtížnější na naučení.

Výsledky testu potvrzují, že Cibulová architektura je nejkomplikovanější architekturou. Pokročilý programátoři byly schopni alespoň částečně odpovědět na všechny otázky ohledně této architektury. Naopak méně pokročilý nebyly schopni odpovědět téměř na žádné otázky o Cibulové architektuře. Zbylé dvě architektury byly dobře pochopeny jak pokročilými, tak méně zkušenými programátory. Obě skupiny tedy byly schopny odpovědět správně na většinu otázek ohledně těchto architektur.

Zhodnocení

Cibulová architektura je obtížnější na naučení než Architektura anemického doménového modelu a Architektura aktivních záznamů. Pro programátory, kteří nemají předchozí zkušenosti s architekturou, je pochopení Cibulové architektury velice obtížné. Pro zkušenější programátory je Cibulová architektura méně náročná, ale přesto vyžaduje nemalé úsilí.

5.2 Čitelnost architektur

Čitelnost textu určuje, jak náročné je pro člověka pochopení textu [3]. Čitelnost softwarových aplikací je důležitá vlastnost, jelikož programátor při údržbě aplikace tráví až 70% času pouze čtením kódu [3]. Obecně také panuje přesvědčení, že čitelnost kódu je rozhodující v určení kvality kódu.

Existuje několik algoritmů, které zkoumají čitelnost kódu v malých částech programu, jako jsou funkce. Ukázkou takového algoritmu může být Cyklomatická složitost [19], která zkoumá větvení funkcí. Takové algoritmy jsou však nedostačující pro porovnání čitelnosti

architektur, jelikož pro architekturu nejsou detaily, jako je obsah jednotlivých funkcí, důležité. Z těchto důvodů bylo pro porovnání čitelnosti architektur zvolena metoda, ve které programátoři zkoumali byznys pravidla aplikace a následně byl vyhodnocen čas, jak dlouho jim orientace v aplikaci trvala.

Pro tuto metriku bylo použito jedenáct programátorů z předchozí části. Programátoři nejdříve prošli školením a poté vyplnili test o architekturách. V následující části byly opraveny chyby, které programátoři udělali a poté byly hlouběji seznámeni s jednotlivými architekturami. Následně byla každému programátorovi ukázána demonstrační aplikace v určité architektuře a úkolem programátora bylo popsat, co aplikace dělá a vyjmenovat všechna byznys pravidla vyskytující se v aplikaci.

Kvůli malému vzorku programátorů byla čitelnost porovnána pouze mezi dvěma architekturami — Anemickým doménovým modelem a Cibulovou. Tyto dvě architektury byly vybrány, jelikož Architektura aktivních záznamů byla napsána v programovacím jazyce Python. Zbylé dvě architektury byly napsány v programovacím jazyce C#, se kterým měli pokročilí programátoři zkušenosti. Výběr architektur napsaných v C# tedy zajistil výsledky, které nejsou ovlivněny znalostí pouze jednoho programovacích z jazyků.

Předpokládaným výsledkem testu bylo, že obě skupiny budou schopny lépe číst Architekturu anemického doménového modelu. Tento předpoklad byl založen na výsledcích předchozího testu a znalostech pokročilých programátorů. Předchozí test ukázal, že méně pokročilí programátoři nejsou plně schopni pochopit Cibulovou architekturu. Z tohoto důvodu by měla být Architektura anemického doménového modelu lépe čitelná. Pokročilí programátoři měli již před experimentem velké zkušenosti s Architekturou anemického doménového modelu, proto by se také dalo předpokládat že tato architektura bude lépe čitelná.

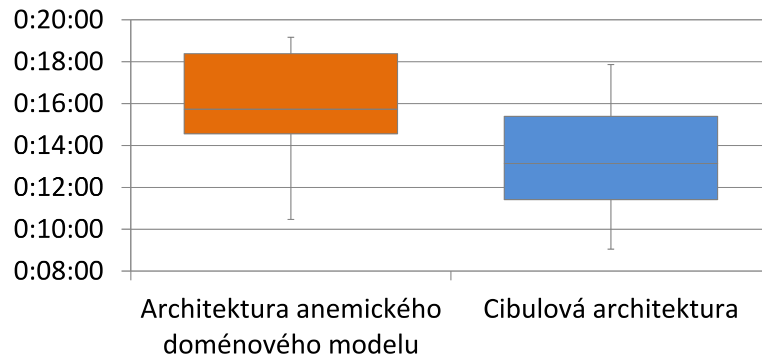
Výsledky experimentu ukázali, že předpoklady byly chybné. I přes neznalost Cibulové architektury téměř všechny výsledky ukázali, že Cibulová architektura má lepší čitelnost. Graf 5.2 ukazuje výsledky experimentu. Příloha C.1 obsahuje všechny naměřené časy v tomto testu. I přesto že Cibulová architektura měla mnoho zdánlivých nevýhod oproti Anemickému doménovému modelu, dokázala mít lepší čitelnost. Tento výsledek tedy znamená, že Cibulová architektura by měla být v praxi čitelnější než Anemický doménový model.

Výsledky experimentu jsou pravděpodobně způsobeny tím, že Cibulová architektura používá doménový model odpovídající reálnému světu. Tento model je obecně považován za lépe čitelný než model, který je vytvořen na základě databázového schéma. Více informací je napsáno v podkapitolách 2.7 a 2.7.1. Obecně tedy můžeme předpokládat, že Cibulová architektura je čitelnější i pro aplikace s větším množstvím byznys logiky.

5.3 Rozšiřitelnost architektury

Rozšiřitelnost aplikace je míra, která určuje, jak obtížné je rozšířit nebo upravit aplikaci tak, aby splňovala nové byznys požadavky. Rozšiřitelnost je důležitou vlastností většiny aplikací, jelikož byznys požadavky jsou často měněny. Pro aplikaci je tedy esenciální, aby byla schopna co nejrychleji reagovat na změny.

Obecně existuje několik metod, jejichž dodržení by mělo vést k lépe rozšiřitelnému kódu. Příkladem těchto metod může být SOLID [4], GRASP [33], zákon Demeter [14]. Nevýhodou těchto technik je, že jsou to pouhá doporučení, která nemusí vždy vést k lepšímu kódu. Zároveň je také možné použít téměř všechny tyto techniky na všechny vybrané architektury a proto nemají z pohledu porovnání architektur velký význam. Z těchto důvodů bylo pro porovnání rozšiřitelnosti stejně jako v předchozí části použito testování na programátorech.



Obrázek 5.2: Graf ukazující čitelnost Architektury anemického doménového modelu a Cibulové architektury. Na ose Y je uveden čas v minutách. Graf ukazuje, že Cibulová architektura má lepší čitelnost.

Pro porovnání bylo použito stejných jedenáct programátorů jako v předchozích experimentech. Tento úkol byl poslední, který programátoři prováděli.

Úkolem programátorů v tomto případě bylo rozšířit všechny demonstrační aplikace o nové byznys pravidlo. Jako metrika byl zvolen čas, který určoval jak dlouho úprava trvala. Byznys pravidlo, které měli programátoři implementovat, znělo takto: „Škola chce přidat sporty. Každý student si může zaregistrovat pouze jeden sport za celé studium. Za splněný sport dostane student pět kreditů. Sporty nemají žádného učitele a stačí když u nich bude evidován název a popis. Změna sportu není povolena.“ Pro jednoduchost nebylo potřeba implementovat prezentační vrstvu aplikace.

Při implementaci byznys požadavku byly všichni programátoři vedeni ke stejnému způsobu implementace, aby se nesnažili o různě obecná řešení, která by způsobila rozdílné časy. Každý programátor byl po čas implementace sledován a upozorněn, pokud vytvořil v programu chybu. Tento způsob byl zvolen, jelikož vytvoření chyb by mohlo vést k zrychlení implementace a tím i k ovlivnění výsledku měření.

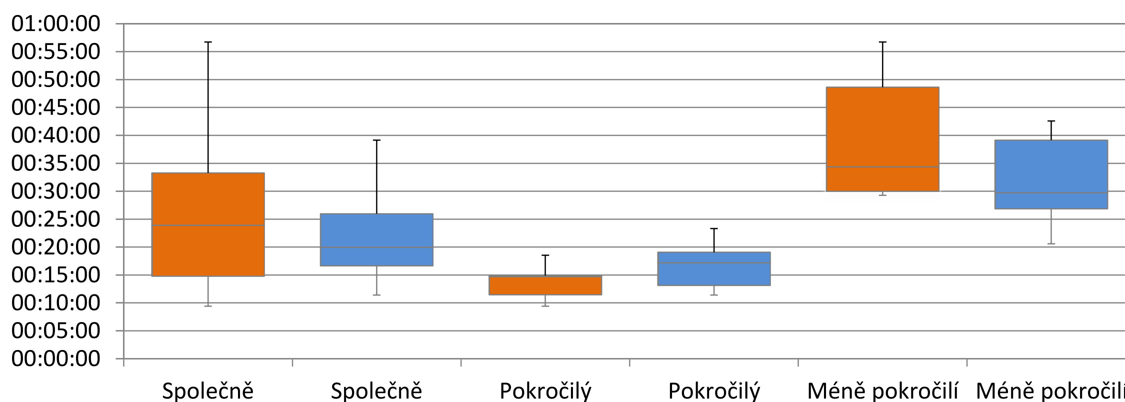
Při návrhu experimentu byla zvažována i varianta, ve které by programátoři nebyly opravováni a výsledný čas by byl upraven podle počtu chyb, které při implementaci vytvořili. Tento způsob testování ale nebyl použit, jelikož různé chyby v programu trvají různě dlouho opravit a bylo by potřeba vytvořit metriku, která určuje, jak dlouho by oprava jednotlivých chyb trvala. Namísto toho byl zvolen způsob, ve kterém byl programátor ponechán vytvořit chybu a až poté opraven, čímž se započítal i čas potřebný pro opravu chyby.

Předchozí školení na jednotlivé architektury a po něm následující test na znalosti programátorů ukázaly, ukázaly, že méně pokročilý programátoři nemají dostatečné znalosti pro práci s Cibulovou architekturou. Z tohoto důvodu byla rozšiřitelnost pro tuto skupinu testována pouze na Architektuře anemického doménového modelu a Architektuře aktivních záznamů. Skupina pokročilých programátorů byla testována na všech architekturách.

Každý programátor upravoval více než jednu architekturu. Je tedy očekáváno, že aplikace, kterou upravoval jako druhou nebo třetí, bude rychleji změněna. Z tohoto důvodu měl každý programátor přiřazené pořadí, ve kterém úpravu architektur prováděl. Pořadí bylo vytvořeno tak, aby každá architektura byla rovnoměrně zastoupena.

5.3.1 Výsledky testování

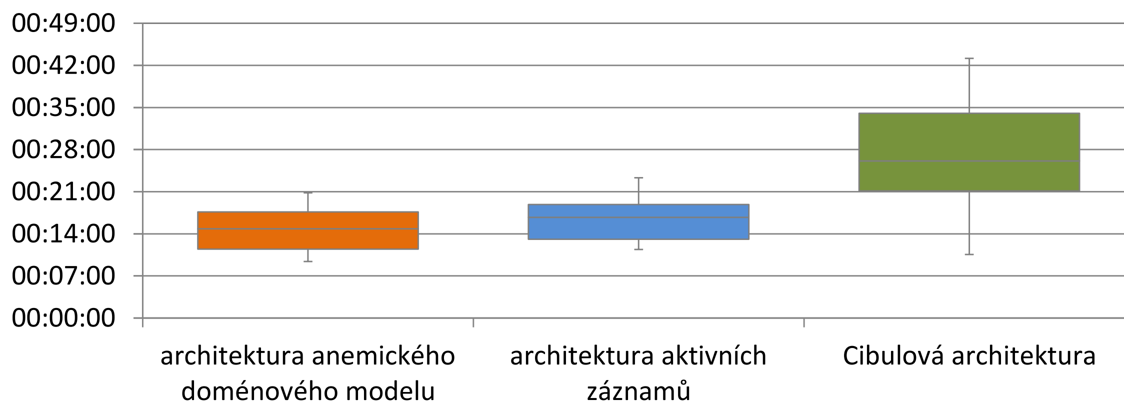
Graf 5.3 ukazuje výsledky testování Architektury anemického doménového modelu a Architektury aktivních záznamů. Pokročilý programátoři měli podle očekávání rychlejší čas v Architektuře anemického doménového modelu, jelikož tuto architekturu znají a používají. I přes tuto nevýhodu byla Architektura aktivních záznamů jen o něco málo pomalejší. Méně zkušení programátoři, kteří neměli předchozí znalost s žádnou architekturou, měli o něco rychlejší čas v Architektuře aktivních záznamů. Rozdíl mezi Architekturou anemického doménového modelu a Architekturou aktivních záznamů je ale velice malý. Můžeme tedy říct, že Architektura aktivních záznamů je o něco málo lépe rozšiřitelná, pokud obsahuje malé množství byznys pravidel. Zda by tento výsledek platil i pro aplikace s větším množstvím byznys pravidel, nemůžeme z výsledků určit. Příloha D obsahuje všechny naměřené časy.



Obrázek 5.3: Graf rozšiřitelnosti Architektury anemického doménového modelu a Architektury aktivních záznamů. Oranžově je označena Architektura aktivních záznamů a modře je označena Architektura anemického doménového modelu. Na ose Y je uveden čas v minutách. Graf ukazuje, že Architektura aktivních záznamů je o něco jednodušší na rozšíření.

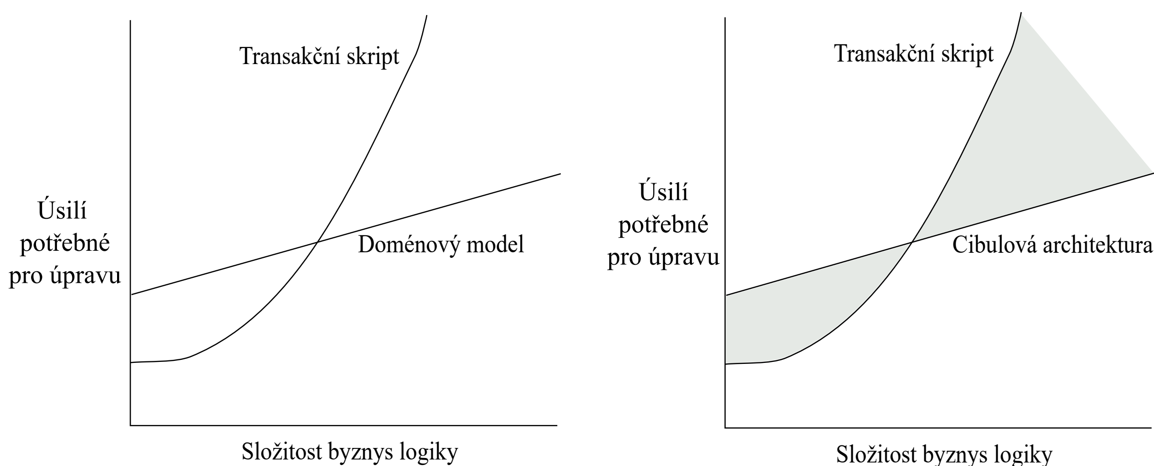
Cibulová architektura se ukázala jako nejobtížnější na úpravu. Tyto výsledky byly nezávislé na tom kolikátá byla Cibulová architektura upravována. Graf 5.4 ukazuje časy potřebné pro úpravu byznys pravidel. Tento graf obsahuje výsledky pouze skupiny pokročilých programátorů. Méně zkušení programátoři nebyly v grafu zahrnuti jelikož v rámci experimentu neupravovali Cibulovou architekturu a jejich výsledky u Architektur anemického doménového modelu a aktivních záznamů by mohli ovlivnit porovnání s Cibulovou architekturou. Nižší rozšiřitelnost Cibulové architektury je pravděpodobně z části způsobena neznalostí Cibulové architektury. Bylo by tedy vhodné výsledky potvrdit i jinými zdroji.

Martin Fowler v knize Patterns of Enterprise Application Architecture [10] představil graf, který ukazuje obtížnost úpravy podle toho, jakou má aplikace složitost. Zjednodušená verze tohoto grafu je ukázána na 5.5. Graf ukazuje pouze Architekturu transakčních skriptů a architekturu používající doménový model. Je tedy potřeba upravit tento graf tak, aby obsahoval architektury porovnávané v této práci. Cibulová architektura používá doménový model, proto můžeme předpokládat, že by v grafu byla reprezentována podobným způsobem. Aktivní záznam je autorem původního grafu považován za vylepšení Transakčního skriptu [10]. Architektura anemického doménového modelu je podobná Aktivnímu záznamu, proto můžeme předpokládat, že by v grafu obě tyto architektury byly mezi Transakčním skriptem a doménovým modelem. Upravený graf je ukázaný na diagramu 5.5.



Obrázek 5.4: Graf ukazující rozšiřitelnost všech tří vybraných architektur. Osa Y ukazuje čas v minutách. Tento graf obsahuje data pouze pokročilých programátorů a ukazuje, že Cibulová architektura má nejhorší rozšiřitelnost ze vše vybraných architektur.

Experiment s rozšiřitelností architektury podporuje část grafu 5.5, která popisuje aplikace s malým množstvím byznys logiky. Rozšiřitelnost Cibulové architektury je horší pro jednodušší aplikace. Můžeme tedy předpokládat, že experiment s programátory nebyl ovlivněn pouze neznalostí Cibulové architektury.



Obrázek 5.5: Graf ukazující rozšiřitelnost architektur podle složitosti byznys logiky. Graf vlevo je převzatý z [10]. Graf vpravo je založený na předchozím grafu a je upravený tak, aby obsahoval architektury popsané v této práci. Šedá část vyznačuje místo, ve kterém by se vyskytovala Architektura anemického doménového modelu a aktivních záznamů.

5.3.2 Zhodnocení

Výsledky z testů o rozšiřitelnosti architektur se shodují s vědecky nepodloženým grafem Martina Fowlera [10], který je ukázaný na obrázku 5.5. Graf 5.5 ukazuje stejný graf upravený tak, aby ukazoval architektury popsané v této práci. Podle tohoto grafu bychom mohli říct, že Cibulová architektura vyžaduje napsat větší množství kódu, který je ale později jednodušeji rozšiřitelný. Toto ale nemusí být jediná reprezentace tohoto grafu.

Výsledky zkoumání čitelnosti architektur naznačují, že Cibulová architektura je lépe čitelná než ostatní architektury bez ohledu na množství byznys pravidel. Předchozí graf tedy může být reprezentován tak, že rozšíření aplikace používající Cibulovou architekturu, je vždy obtížnější než u ostatních architektur. Složitější aplikace ale vyžadují pro úpravu více čtení již existujícího kódu a proto je výsledný čas potřebný pro úpravu Cibulové architektury nižší. Jinak řečeno pokud by programátor měl úplné znalosti o aplikaci a o všech jejích byznys pravidlech, bylo by vždy bez ohledu na množství byznys pravidel jednodušší rozšiřovat aplikaci používající Anemický doménový model nebo Aktivní záznamy. Programátor ale u složitých aplikací nikdy nemá úplné znalosti a vždy musí číst již existující kód a proto úprava v Cibulové architektuře bude trvat kratší dobu.

Pokud předpokládáme, že reprezentace grafu pomocí čitelnosti je správná, můžeme očekávat, že snížení čitelnosti Cibulové architektury vytvoří architekturu horší něž další dvě vybrané architektury. Naštěstí takový příklad můžeme nalézt. Cibulová architektura používá DDD popsané v kapitole 2.7.1. DDD popisuje dva druhy návrhových vzorů nazývané taktické a strategické [31]. Taktické návrhové vzory popisují jednotlivé části architektury aplikace jako jsou entity, hodnotové objekty, repozitáře. Všechny tyto koncepty byly popsány v kapitolách 2.7.3, 2.7.4, 2.8. Strategické návrhové vzory říkají, jak má být modelována byznys logika, aby co nejlépe odpovídala reálnému světu. Strategické návrhové vzory tedy přináší lepší čitelnost aplikace. Při použití DDD se v některých případech používají pouze taktické návrhové vzory bez strategických. Tento způsob implementace vede k Cibulové architektuře s horší čitelností. Předchozí teorie tedy říká, že by tato architektura neměla být lepší než zbylé dvě. Tato teorie je potvrzena, jelikož použití pouze strategických návrhových vzorů je obecně považováno za chybné použití DDD, které často vede k selhání projektu [40, 36, 31].

5.4 Testovatelnost Architektury aktivního záznamu

Definice Architektury aktivních záznamů říká, že entity, které obsahují byznys logiku aplikace obsahují i logiku, která se stará o přístup k databázi. Prakticky tedy není možné oddělit entity a tím i byznys logiku obsaženou v těchto entitách od databázových operací. Z tohoto důvodu není možné téměř v žádné z běžných implementací nahradit databázové operace za jiné testovací operace. Byznys logiku Architektury aktivních záznamů tedy není ve většině implementací možno otestovat jednotkovými testy.

Mnoho programátorů se snaží v těchto architekturách použít různé techniky, které odstiňují byznys logiku aplikace od Aktivních záznamů, což vede k architektuře podobné Anemickému doménovému modelu. Oddělení Aktivních záznamů od přístup k databázi ale způsobuje komplikovanější architekturu. V některých případech se může stát, že výsledná architektura je komplikovanější než původní architektura, která nebyla testovatelná [9].

Z pohledu porovnání architektur je důležité, že Architekturu aktivních záznamů není ve většině případů možné testovat pomocí jednotkových testů. Obecně tedy můžeme Architekturu anemického doménového modelu a Cibulovou architekturu považovat za jednodušeji testovatelné.

5.5 Celkové zhodnocení testovaní architektur

Při výběru architektury jsou dva důležité faktory — množství byznys pravidel a znalosti týmu vývojářů. Experimenty ukázaly, že méně zkušení programátoři nemají problém nastu-

dovat Architekturu anemického doménového modelu nebo Architekturu aktivních záznamů. Cibulová architektura je naopak pro tyto programátory velkým problémem a vyžaduje delší studium. Překvapivě i programátoři, kteří mají zkušenosti s Architekturou anemického doménového modelu, vyžadují poměrně velké úsilí, aby byli schopni použít Cibulovou architekturu. Kolik času by reálně týmy potřebovaly, je těžké odhadnout, ale autor se domnívá, že by nebylo by překvapující, kdyby to byl čas v řádu dní až měsíců pro pokročilý tým.

Obtížnost na naučení architektury je důležitým faktorem i pro nové členy týmu. Pokud tým používá Cibulovou architekturu, bude muset přijímat pouze zkušené vývojáře a nebo méně zkušené a poté je dlouhodobě zaučovat. Dvě jednodušší architektury umožňují poměrně rychlé zaučení všech programátorů, což vede k zrychlení vývoje aplikace.

Mezi Architekturami aktivních záznamů a Anemického doménového modelu nebyly zjištěny téměř žádné rozdíly ani v jedné ze zvolených metrik. Tento fakt je pravděpodobně způsoben tím, že jsou si architektury velice podobné a pro vyhodnocení malých detailů by byl potřeba mnohem větší vzorek programátorů. Jediný rozdíl je v testovatelnosti pomocí jednotkových testů. Architekturu anemického doménového modelu je možné testovat pomocí jednotkových testů a Architektura aktivních záznamů ve většině případů neumožňuje testování jednotkovými testy. Obecně tedy můžeme říct, že je o něco lepší Architektura anemického doménového modelu.

Výsledky z testování Cibulové architektury naznačují, že je obtížná na rozšíření, ale má lepší čitelnost. Důsledkem tohoto pozorování je, že architektura je horší pro malé projekty s malým množstvím byznys logiky, jelikož v těchto aplikacích není mnoho kódu, který je potřeba číst. S velikostí aplikace ale přibývá i množství kódu, který je potřeba číst, což ve výsledku znamená, že úpravy jsou pro složité aplikace rychlejší v Cibulové architektuře. Cibulová architektura je tedy vhodná pouze na velké projekty.

Vysoká čitelnost a nízká rozšiřitelnost dokázaly vysvětlit, proč jsou aplikace používající Cibulovou architekturu pouze s taktickými návrhovými vzory považovány za špatné použití DDD. Více o tomto problému je napsáno v podkapitole 5.3.2. Dalším důsledkem tohoto zjištění je, že špatné použití Cibulové architektury pravděpodobně povede k selhání projektu kvůli špatnému vyjádření byznys logiky.

5.6 Limitace porovnání architektur

Porovnání architektur v této práci je limitováno malým vzorkem programátorů. Jedenáct programátorů nemusí být považováno za dostatečně velký vzorek. I přesto že výsledky ve většině případů potvrdily předpoklady nebo závěry jiných odborníků, je možné, že tato zjištění byla způsobena pouze náhodnou shodou.

Měření znalostí potřebných pro implementaci bylo limitováno omezeným časem testovaných programátorů. Pro nejpřesnější výsledky by bylo vhodné nechat programátory několik dní až měsíců studovat jednotlivé architektury a následně je nechat implementovat reálnou aplikaci, která by ověřila nastudované znalosti. Pro takovou studii by však bylo potřeba velké množství času testovaných programátorů. V této práci byl tedy zvolen způsob, který je méně přesný, ale zároveň méně časově náročný.

Rozšiřitelnost architektur také byla ovlivněna časovým limitem programátorů. Jelikož programátoři prováděli pouze jednu úpravu, mohlo se stát, že tato úprava byla jednodušší v jedné z architektur. Kdyby nebylo časové limitace, mohli by programátoři provést více různě rozsáhlých úprav, které by pravděpodobně lépe ukazovaly rozšiřitelnost architektur.

Kapitola 6

Závěr

Tato práce popisuje často používané architektury a koncepty potřebné pro jejich pochopení. Z popsaných architektur byly vybrány tři, pomocí kterých byla navržena a implementována demonstrační aplikace. Demonstrační aplikace poté sloužila pro porovnání architektur na základě zvolených metrik. Architektury vybrané pro tvorbu aplikace a porovnání byly následující — Cibulová architektura, Architektura anemického doménového modelu a Architektura aktivních záznamů. Architektury byly zvoleny převážně na základě popularity

Navržená demonstrační aplikace je informační systém pro vysoké školy se zjednodušenými byznys pravidly. V této aplikaci si mohou studenti registrovat předměty, za které získávají kredity stejně jako na vysokých školách. Školní systém byl vybrán, jelikož je v něm jednoduché vytvořit byznys pravidla, které dobře ukazují výhody a nevýhody vybraných architektur. V navržené demonstrační aplikaci pak byla provedena implementace pomocí všech vybraných architektur.

Pro porovnání architektur byly vybrány tři metriky čitelnosti, rozšiřitelnosti, testovatelnost a potřebné znalosti programátorů pro použití architektury. Všechny metriky s výjimkou testovatelnosti byly porovnány na základě zpětné vazby od programátorů. Testovatelnost byla porovnána na základě nastudovaných informací.

Potřebné znalosti vývojářů byly porovnány na základě testu, který programátoři vyplnili, poté co prošli školení na vybrané architektury. Výsledky testu pak ukázaly, že Cibulová architektura je nejobtížnější na naučení. Mezi zbylými dvěma architekturami nebyl nalezen žádný rozdíl v obtížnosti na naučení. Výsledky také ukázaly, že méně zkušeným programátorům dělá velké problémy pochopit Cibulovou architekturu. Pro zkušenějším programátory byla Cibulová architektura také problémem, ale byli schopni většinou alespoň částečně odpovědět na všechny otázky.

Čitelnost byla testována tak, že byla programátorům představena demonstračních aplikace v jedné z architektur a programátoři měli za úkol popsat, co aplikace dělá a vyjmenovat všechna byznys pravidla, která se v aplikaci vyskytují. Jako metrika byl zvolen čas, který určoval, jak dlouho orientace v aplikaci programátorům trvala. Výsledky ukázaly, že Cibulová architektura má lepší čitelnost než Architektura anemického doménového modelu.

Dalším testem, který byl proveden, bylo porovnání rozšiřitelnosti architektur. V tomto případě měli programátoři za úkol rozšířit vybrané demonstrační aplikace o nové byznys pravidlo. Jako metrika byl zvolen čas, který určoval, jak dlouho úprava trvala. Tento test ukázal, že Cibulová architektura je nejobtížnější na rozšíření.

Vysoká čitelnost a nízká rozšiřitelnost Cibulové architektury ukázaly, že tato architektura je vhodná pouze na komplikované projekty. Ve složitých aplikacích každá úprava vyžaduje velké množství čtení již existujícího kódu, což způsobuje, že celkový čas na úpravu

složité aplikace v Cibulové architektury je kratší než u zbylých dvou architektur. Naopak jednoduché aplikace nevyžadují pro změnu mnoho čtení již existujícího kódu a proto jsou tyto úpravy pomalejší v Cibulové architektuře.

Poslední porovnávanou metrikou byla testovatelnost architektur. Testovatelnost byla porovnána na základě informací získaných o architekturách. Výsledkem porovnání bylo, že Architektura aktivních záznamů má často nižší testovatelnost než zbylé dvě architektury.

Tato práce nezodpověděla otázku, jak složitá aplikace musí být, aby se vyplatilo použití Cibulové architektury. Následující studie by se tedy mohla zabývat touto otázkou. Bylo by například zajímavé analyzovat aplikace různé velikosti a zjišťovat, kolik času z úpravy již existujícího kódu programátoři stráví čtením. Na základě takové analýzy by se dalo alespoň částečně odhadnou, jak složitá aplikace musí být, aby se vyplatila Cibulová architektura.

Dále tato práce nebyla schopna zjistit téměř žádné rozdíly mezi Architekturou anemického doménového modelu a Architekturou aktivních záznamů. Což naznačuje, že rozdíl mezi těmito architekturami je velice malý. Z toho důvodu by bylo vhodné provést další testy vztahující se k porovnání těchto architektur. Testy by mohly být provedeny stejným způsobem jako v této práci, ale na větším vzorku programátorů.

Literatura

[1] Aaronaught: *What's wrong with circular references?* [Online; navštíveno 13.4.2019]. URL https://softwareengineering.stackexchange.com/a/12030

[2] BOHMER, M.: *Zend Framework: programujeme webové aplikace v PHP*. Brno: Computer Press, 2010, ISBN 978-802512965-4.

[3] Buse, R. P.; Weimer, W. R.: *A Metric for Software Readability*. [Online; navštíveno 15.4.2019].
URL
https://web.eecs.umich.edu/~weimerw/p/weimer-issta2008-readability.pdf

[4] C, M. R.: *Clean architecture: a craftsman's guide to software structure and design*. London, England: Prentice Hall, 2018, ISBN 978-013449416-6.

[5] Django: *Django*. [Online; navštíveno 28.1.2019].
URL https://www.djangoproject.com/

[6] Evans, E.: *Domain-driven design: tackling complexity in the heart of software*. Boston: Addison-Wesley, 2003, ISBN 978-0-321-12742-6.

[7] Fowler, M.: *AggregateOrientedDatabase*. [Online; navštíveno 21.1.2019].
URL https://martinfowler.com/bliki/AggregateOrientedDatabase.html

[8] Fowler, M.: *AnemicDomainModel*. [Online; navštíveno 06.12.2018].
URL https://www.martinfowler.com/bliki/AnemicDomainModel.html

[9] Fowler, M.: *Is TDD Dead?* [Online; navštíveno 14.4.2019].
URL https://martinfowler.com/articles/is-tdd-dead/

[10] Fowler, M.: *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.., 2002, ISBN 978-0-321-12742-6.

[11] Gamma, E.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995, ISBN 0-201-63361-2.

[12] Graca, H.: *Ports & Adapters Architecture*. [Online; navštíveno 07.12.2018].
URL https://herbertograca.com/2017/09/14/ports-adapters-architecture/

[13] Grandnode: *Grandnode*. [Online; navštíveno 28.4.2019].
URL https://grandnode.com/

[14] Hunt, A.; Thomas, D.: *Tell, Don't Ask*. [Online; navštíveno 15.4.2019].
URL https://pragprog.com/articles/tell-dont-ask

[15] KABELOVÁ, A. a. L. D.: *Velký průvodce protokoly TCP/IP a systémem DNS. 5., aktualiz. vyd.* . Brno: Computer Press, 2008, ISBN 978-802512236-5.

[16] Laravel: *Laravel*. [Online; navštíveno 28.1.2019].
URL https://laravel.com/

[17] Marten: *Marten*. [Online; navštíveno 29.4.2019].
URL https://jasperfx.github.io/marten/

[18] MARTIN, R. C.: *Čistý kód: [návrhové vzory, refaktorování, testování a další techniky agilního programování]*. Brno: Computer Press, 2009, ISBN 978-802512285-3.

[19] McCABE, T. J.: *7 Reasons Why DDD Projects Fail*. [Online; navštíveno 11.5.2019].
URL http://www.literateprogramming.com/mccabe.pdf

[20] Mehdi Khosrow-Pour, D.: *Encyclopedia of Information Science and Technology, Third Edition* . Hershey, PA: Information Science Reference, 2014, ISBN 978-146665888-2.

[21] doc. Mgr. Miloš Kudělka, P.: *Vývoj informačních systémů*. [Online; navštíveno 29.1.2019].
URL https://homel.vsb.cz/~kud007/lectures/vis_02.pdf?fbclid=IwAR0RdHna41EEUqD5waraQTIou0-7rvbJNy11fy1F0eShVUdmmnJFi5bFpNg

[22] Minařík, R. P.: *Domain-Driven Design, Posudek oponenta*. [Online; navštíveno 29.1.2019].
URL https://is.muni.cz/th/y04ag/

[23] Mička, P.: *Architektura aplikací*. [Online; navštíveno 29.1.2019].
URL https://cw.fel.cvut.cz/old/_media/courses/a7b39wpa/architektura.pdf?fbclid=IwAR2ZgkObtXWdOWbz_5hvKkwUWHLm1oWaSPWMycDcZMMp4zZ-UccUnXLXVUo

[24] NopCommerce: *Source code organization. Architecture of nopCommerce*. [Online; navštíveno 06.12.2018].
URL http://docs.nopcommerce.com/pages/viewpage.action?pageId=1442491

[25] Palermo, J.: *The Onion Architecture : part 1*. [Online; navštíveno 11.12.2018].
URL https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[26] Palermo, J.: *The Onion Architecture : part 3*. [Online; navštíveno 4.1.2018].
URL https://jeffreypalermo.com/2008/08/the-onion-architecture-part-3/

[27] P.Matha, M.: *Object-oriented Analysis and Design Using Umla Introduction to Unified Process and Design Patterns*. Prentice Hall of India Private Ltd, 2008, ISBN 978-8-120-33322-2.

[28] Rails: *Active Record Basics*. [Online; navštíveno 05.12.2018].
URL https://guides.rubyonrails.org/active_record_basics.html

[29] Rails: *Ruby on Rails*. [Online; navštíveno 28.1.2019].
URL https://rubyonrails.org/

[30] Richards, M.: *Software Architecture Patterns*. O'Reilly Media, Inc., 2015, ISBN 978-1-491-97143-7.

[31] Scott, M.: *Patterns, principles, and practices of domain-driven design*. Wrox, 2015, ISBN 9-781-11871469-0.

[32] SimplCommerce: *SimplCommerce*. [Online; navštíveno 28.4.2019].
URL https://www.simplcommerce.com/

[33] Umair, M.: *GRASP*. [Online; navštíveno 15.4.2019].
URL https://dzone.com/articles/solid-grasp-and-other-basic-principles-of-object-o

[34] Vernon, V.: *Effective Aggregate Design*. [Online; navštíveno 22.1.2019].
URL http://dddcommunity.org/library/vernon_2011/

[35] Vernon, V.: *Effective Aggregate Design Part I: Modeling a Single Aggregate*. [Online; navštíveno 22.1.2019].
URL http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf

[36] VERNON, V.: *Implementing domain-driven design*. Upper Saddle River, NJ: Addision-Wesley, 2013, ISBN 032-183457-7.

[37] Wikipedia contributors: *Business logic*. [Online; navštíveno 27.4.2019].
URL https://en.wikipedia.org/wiki/Business_logic

[38] Wikipedia contributors: *Business software*. [Online; navštíveno 27.4.2019].
URL https://en.wikipedia.org/wiki/Business_software

[39] Wikipedia contributors: *Objektově relační mapování*. [Online; navštíveno 27.4.2019].
URL https://en.wikipedia.org/wiki/Object-relational_mapping

[40] Young, G.: *7 Reasons Why DDD Projects Fail*. [Online; navštíveno 8.5.2019].
URL https://vimeo.com/13824218

[41] Ševčík, D.: *Domain-Driven Design*. [Online; navštíveno 29.1.2019].
URL https://is.muni.cz/th/y04ag/

Přílohy

Příloha A

Obsah přiloženého paměťového média

- */ActiveRecord* — obsahuje implementaci demonstrační aplikace v architektuře aktivních záznamů
- */DomainModel* — obsahuje implementaci demonstrační aplikace v Cibulové archuitektuře
- */AnemicModel* — obsahuje implementaci demonstrační aplikace v anemického doménového modelu
- */CorrectSolutionToExtendability/ActiveRecord* — obsahuje implementaci demonstrační aplikace v architektuře aktivních záznamů rozšířenou o sport
- */CorrectSolutionToExtendability/DomainModel* — obsahuje implementaci demonstrační aplikace v Cibulové archuitektuře rozšířenou o sport
- */CorrectSolutionToExtendability/AnemicModel* — obsahuje implementaci demonstrační aplikace v anemického doménového modelu rozšířenou o sport
- */BP-skoleni.pptx* — obsahuje prezentaci ke školení architektur
- */BPLatex* — obsahuje tuto práci v latexu

Příloha B

Test znalostí programátorů

B.1 Cibulová architektura

1. Zrefaktorujte následující kód pomocí objektového programování. Použijte koncepty popsané Cibulovou architketurou jako jsou entity, value objecty, application services a další. Metody CalculateAverageColorRgb a ColorToHex jsou součásti byznys logiky aplikace. Při refaktoringu můžete změnit všechny metody, například není potřeba používat Active record pro přístup k databázi.

```
1 Product product = Product.GetById(id);
2 Image productImage = product.Image;
3 string averageColorRgb = service.CalculateAverageColorRgb(productImage);
4 string averageColorHex = service.ColorToHex(averageColorRgb);
5 return averageColorHex;
```

Řešení:

```
public class Product{
    private Image image;

    public Color CalculateAverageColor(){
        return image.CalculateAverageColor()
    }
}

public class Image{

    public Color CalculateAverageColor(){
        //return average color
    }
}

public class Color(){
    public string GetHexRepresentation(){
        //return hexRepresentation
    }
}

public class ProductApplicationService(){
    private ProductRepostiroty productRepository
            = new ProductRepository();
    public string GetProductImageAverageColorHex(int productId){
        var product = productRepository.GetById(productId);
        var color = product.CalculateAverageColor();
        return color.GetHexRepresentation();
    }
}
```

Hodnoceno bylo správné rozdělení do entit, hodnotových objektů, aplikační vrstvy a repozitáře. Za chybná jména proměnných nebo syntaktické chyby nebylo sníženo hodnocení.

Méně pokročilí programátoři často vůbec nevěděli co dělat. Pokročilý programátoři byly ve většině případů schopni dospět alespoň k částečnému řešení. Jeden pokročilý programátor byl schopen správně rozdělit kód ale nepoužil objekt color. Tato odpověď byla uznána jako chybná.

2. V jaké části Cibulové architektury vyřešíte autorizaci (zda má uživatel oprávněni provést operaci) a autentifikace uživatele?

Správnou odpovědí je, že autorizace je prováděna na aplikační vrstvě v některých případech pomocí volání doménové služby. Autentifikace je obvykle prováděna na prezenční vrstvě.

Odpovědi programátorů:

Začátečnici považovali entity za vhodné místo pro provádění validací.

3. Jak byste uložili doménový model do úložiště? Popište výhody a nevýhody vašeho přístupu.

Tato otázka má mnoho správných odpovědí. Důležité bylo si uvědomit výhody a nevýhody zvoleného přístupu.

Správné odpovědi:
Použití NoSQL databáze a serializovat agregáty.
Použití relační databáze a mít v aplikaci dva modely. Jeden odpovídající databázi a jeden odpovídající doméně. Poté mít nějakou vrstvu, která mezi těmito modely mapuje. V tomto případě je potřeba si uvědomit, že doménový model má porovnání proměnné a je potřeba použít něco jako visitor pattern pro zjištění hodnot.
Použití ORM pro uložení doménového modelu.
Méně pokročilí programátoři dokázali obvykle odpovědět správnou metodou, nedokázali však obhájit výhody a nevýhody přístupu.
4. Namodelujte aggregáty, aggregate rooty a entity pro následující systém. U entit stačí uvést jméno.
Klient požaduje běžný e-shop pro jeho obchod s auty. Zákazníci e-shopu musejí být schopni zobrazit auta, vložit auto do košíku, upravit množství aut v košíku a nakoupit zboží které již je v košíku. U každého uživatele je potřeba evidovat jméno, příjmení, věk, rodné číslo, telefon a další věci potřebné pro doručení. E-shop neobsahuje administraci, ale klient bude vkládat a upravovat zboží na E-shopu pomocí SQL dotazů. U nakoupených předmětů je potřeba evidovat cenu, která byla platná v době zakoupení. Klient také vyžaduje dashboard, na kterém se zobrazuje seznam všech uživatelů a aut v e-shopu. Klient může vložit maximálně 2 věci do košíku.
Správná odpověď:
Klient - aggregate root s jedninou entitou — Cart
Druhý aggregate Car
Otázka je naschvál položena tak, aby byly programátoři nuceni zjistit, co jsou byznys pravidla.

B.2 Anemic domain model

1. Napište kód pro odbavení zboží z košíku v architektuře anemického doménového modelu. Uživatel má kredity a nákupní košík. Košík obsahuje předměty, které je potřeba nakoupit. Každý předmět má cenu. Kredity uživatele slouží jako peníze, za které je zboží nakoupeno.
Funkce řadiče, která je volána, vypadá takto:

```
1 public void Checkout(int userId){}
```

Správná odpověď:

```
1  public void Checkout(int userId){
2      User user = userRepository.GetById(userId);
3      var cartItemsPrice = user.cart.Sum(x=>x.price)
4      if(cartItemsPrice < user.Credits){
5          throw Exception("nedostatek penez");
6      }
7      user.Credits -= carItemsPrice;
8      user.Cart.Clear();
9      userRepository.SaveChanges();
10 }
```

2. Jaký je rozdíl mezi ORM mapperem a repozitářem v architektuře anemického doménového modelu.

Správná odpověď:
Jednou ze zodpovědností repozitáře je oddělení byznys logiky od databázové logiky. ORM mapper ale plně neodděluje databázi od byznys vrstvy. Při potřebě změnit databázi tedy může být potřeba změnit ORM mapper, což může vést k provázání byznys logiky a databáze. Repozitář také zabraňuje opakování stejných dotazů na databázi.
Částečně bylo uděleno za uvědomění si, že repozitář zabraňuje duplikovaní kódu.
3. Popište k čemu se používá repozitář v architektuře anemického doménového modelu. Napište příklad repozitáře.
Správná odpověď:
Repozitář slouží pro přístup k databázi. Anemický doménový model obsahuje jeden repozitář pro jednu databázovou tabulku. Každý repozitář má také přiřazenou jednu entitu, která odpovídá dané databázové tabulce.

```
1  public class User{
2      //database columns
3      public string name;
4      public string secondName;
5  }
6
7  public class UserRepository{
8      public User GetById(int id);
9      public void Update();
10     public void Delete();
11     public void Insert();
12     //aditional database operations
13 }
```

4. Popište, jaké vrstvy běžně používá anemický doménový model a jaké mají zodpovědnosti.
Správná odpověď:
Architektura anemického doménového modelu obvykle používá tři vrstvy. Prezentační, byznys a datovou. Prezentační se stará o zobrazování dat uživateli. Byznys vrstva obvykle obsahuje transakční skripty, které volají repozitáře a provádí byznys logiku. Datová obsahuje pouze repozitáře.

B.3 Active record

1. Napište kód pro odbavení zboží z košíku v architektuře anemického doménového modelu. Uživatel má kredity a nákupní košík. Košík obsahuje předměty, které je potřeba nakoupit. Každý předmět má cenu. Kredity uživatele slouží jako peníze, za které je zboží nakoupeno.
Funkce řadiče, která je volána, vypadá takto:

```
1  public void Checkout(int userId){}
```

Správná odpověď:

```
1  public void Checkout(int userId){
2      User user = User.GetById(userId);
3      var cartItemsPrice = user.cart.Sum(x=>x.price)
4      if(cartItemsPrice < user.Credits){
5          throw Exception("nedostatek penez");
6      }
7      user.Credits -= carItemsPrice;
8      user.Cart.Clear();
9      User.Save();
10 }
```

2. Napište výhody a nevýhody vložení byznys logiky do aktivních záznamů.
Správná odpověď:
Přidání byznys logiky do aktivních záznamů způsobuje provázání databázové logiky s byznys logikou a tím zhoršuje oddělení zodpovědností a testovatelnost. Výhodou je, že je poté možno použít privátních proměnných a zapouzdření.
Částečně bylo uděleno za uvědomění si, že aktivní záznam s byznys logikou porušuje vrstvenou architekturu s vyjmenováním výhod.
3. Popište jak se v architektuře aktivních záznamů přistupuje k databázi. Napište ukázku třídy, která přistupuje k databázi.
Správná odpověď:
V aplikaci jsou vytvořeny entity, které přesně odpovídají tabulká v databázi. Tyto entity mají metody, které slouží pro přístup k databázi.

```
1  public class User{
2      //database columns
3      public string name;
4      public string secondName;
5
6      public static User GetById(int id);
7      public void Update();
8      public void Delete();
9      public void Insert();
10
11     //aditional database operations
12 }
```

4. Popište jaké vrstvy běžně používá aktivní záznam a jaké mají tyto vrstvy zodpovědnosti.
Správná odpověď:
Architektura aktivních záznamů obvykle používá tři vrstvy. Prezentační, byznys a datovou. Prezentační se stará o zobrazování dat uživateli. byznys vrstva obvykle obsahuje transakční skripty, které volají aktivní záznamy a provádí byznys logiku. Datová je reprezentována aktivními záznamy.

B.4 Bonusová otázka:

Myslíte si, že byste byly schopni váš příští projekt implementovat v Cibulové architektuře bez větších problémů?
Všichni odpověděli NE.
Myslíte si, že byste byly schopni váš příští projekt implementovat v jedné ze dvou zbývajících architektur bez větších problémů?
Všichni odpověděli ANO.

B.5 Výsledky testu

Otázka	Ano	Ne	Částečně
1.	0	4	1
2.	0	5	0
3.	0	4	1
4.	1	4	0

Tabulka B.1: Méně pokročilí — Cibulová architektura

Otázka	Ano	Ne	Částečně
1.	5	0	0
2.	1	0	4
3.	5	0	0
4.	5	0	0

Tabulka B.2: Méně pokročilí — Architektura anemického doménového modelu

Otázka	Ano	Ne	Částečně
1.	5	0	0
2.	0	1	4
3.	4	0	1
4.	5	0	0

Tabulka B.3: Méně pokročilí — Architektura aktivních záznamů

Otázka	Ano	Ne	Částečně
1.	3	1	2
2.	6	0	0
3.	4	0	2
4.	3	2	1

Tabulka B.4: Pokročilí — Cibulová architektura

Otázka	Ano	Ne	Částečně
1.	6	0	0
2.	6	0	0
3.	6	0	0
4.	6	0	0

Tabulka B.5: Pokročilí — Architektura anemického doménového modelu

Otázka	Ano	Ne	Částečně
1.	6	0	0
2.	6	0	0
3.	6	0	0
4.	6	0	0

Tabulka B.6: Pokročilí — Architektura aktivních záznamů

Příloha C

Výsledky měření čitelnosti architektur

C.1 Architektura anemického doménového modelu

začátečníci — 0:18:23, 0:19:10, 0:14:33
pokročilí — 0:15:44, 0:10:28, 0:14:10

C.2 Cibulová architektura

začátečníci — 0:15:48, 0:17:52
pokročilí — 0:9:03, 0:11:10, 0:12:07

Příloha D

Výsledky měření rozšiřitelnosti architektur

D.1 Architektura anemického doménového modelu

začátečníci – 0:48:38, 0:29:59, 0:34:22, 0:29:16, 0:56:44
pokročilí – 0:10:22, 0:14:43, 0:18:32, 0:14:57, 0:09:24, 0:14:20

D.2 Cibulová architektura

pokročilí – 0:22:30, 0:43:10, 0:10:34, 0:35:28, 0:20:41, 0:29:44

D.3 Architektura aktivních záznamů

začátečníci – 0:42:35, 0:39:09, 0:26:51, 0:20:05, 0:29:34
pokročilí – 0:16:06,0:11:24, 0:23:19, 0:19:20, 0:12:07, 0:18:15