Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00519/2010
Datum: 05.04.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **SIMON ŠTAMCAR**

Naslov: **PRISTOPI IN VZORCI ZA UPORABO PRI RAZVOJU APLIKACIJ**
APPROACHES AND PATTERNS FOR APPLICATION DEVELOPMENT

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Opravite raziskavo pristopov in vzorcev za uporabo pri razvoju aplikacij. Predstavite pomen različnih pristopov in vzorcev tako na teoretični kot tudi praktični osnovi. Podajte tudi konkretne napotke za hitrejši in lažji razvoj aplikacij.

Mentor:
doc. dr. Rok Rupnik

Dekan:
prof. dr. Franc Solina

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Simon Štamcar

PRISTOPI IN VZORCI ZA UPORABO PRI RAZVOJU APLIKACIJ

DIPLOMSKO DELO NA
VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: doc. dr. Rok Rupnik

Ljubljana, 2010

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani **Simon Štamcar**,

z vpisno številko **63000291**,

sem avtor diplomskega dela z naslovom:

PRISTOPI IN VZORCI ZA UPORABO PRI RAZVOJU APLIKACIJ

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Roka Rupnika

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 01.07.2010 Podpis avtorja: _____

ZAHVALA

„I can no other answer make, but, thanks, and thanks.”

~ W. Shakespeare

Zahvaljujem se svojemu mentorju, doc. dr. Roku Rupniku, za pomoč, potrpežljivost in spodbudo pri izdelavi diplomskega dela. Iskrena hvala tudi staršem za vso podporo v času študija. Hvala pa tudi vsem ostalim, ki ste mi vsa ta leta potrpežljivo stali ob strani.

KAZALO

SEZNAM UPORABLJENIH KRATIC IN SIMBOLOV

DIP	Dependency Inversion Principle – princip obračanja odvisnosti
DRY	Don't Repeat Yourself – ne ponavljaj se
DSL	Domain Specific Language – domensko specifični jezik
DTO	Data Transfer Object – objekt za prenos podatkov
HTML	HyperText Markup Language – jezik spletnih strani
HTTP	HyperText Transfer Protocol – protokol za prenos hiperteksta med strežniki in brskalniki
ISP	Interface Segregation Principle – princip ločevanja vmesnikov
LINQ	Language Integrated Query – del Microsoft .NET platforme, ki omogoča različne poizvedbe po podatkih
LSP	Liskov Substitution Principle – Liskovin princip zamenjave, ki se ukvarja s tipi in podtipi
MVC	Model View Controller – arhitekturni vzorec za razvoj aplikacij, kjer je uporabniški vmesnik ločen od poslovne logike
MVP	Model View Presenter – arhitekturni vzorec za razvoj aplikacij, temelji na MVC
OCP	Open/Closed Principle – princip odprto/zaprto
ORM	Object/Relational Mapping – objektno relacijska preslikava
SQL	Structured Query Language – jezik za upravljanje s podatki v relacijskih podatkovnih bazah
SRP	Single Responsibility Principle – princip ene odgovornosti
TDD	Test Driven Development – razvoj, temelječ na testih
XML	eXtensible Markup Language – format za opisovanje strukturiranih podatkov

Povzetek

Diplomsko delo vsebuje opis nekaterih pristopov in vzorcev, ki se jih lahko oziroma jih je priporočljivo uporabiti za reševanje problemov pri razvoju aplikacij. Namen diplomskega dela je predstaviti pomen različnih pristopov in vzorcev tako na teoretični kot tudi praktični osnovi. Delo vsebuje tudi nekaj konkretnih napotkov za hitrejši in lažji razvoj.

Predstavljene so tudi nekatere programske knjižnice in orodja, ki sledijo prej omenjenim pristopom in nam še dodatno olajšajo delo. Podani so tudi razlogi za izbiro orodja oziroma posameznega pristopa, kajti vedno se je potrebno zavedati, kakšne pozitivne in negativne posledice nosi izbira orodja oziroma pristopa. Uporaba orodij in pristopov je predstavljena tudi na konkretnih primerih.

Pri vsaki aplikaciji je pomembna tudi sama arhitektura aplikacije. Pravilna izbira arhitekture lahko namreč olajša, napačna izbira pa zelo oteži razvoj aplikacije. Tu poskušam poudariti pomembnost začetnih odločitev, ki potem vplivajo na celoten razvoj aplikacije.

KLJUČNE BESEDE:
načrtovalski vzorci, principi, razširljivost, večplastna arhitektura, objektno-relacijska preslikava

Abstract

This thesis describes some development principles and design patterns which can be used to solve various problems during application development. Purpose of the thesis is to present different principles, patterns and practices for application development in both theory and practice. We also add some examples for an easier and faster development.

The thesis also presents some program libraries and tools, which utilize aforementioned patterns and can additionally help with the development process. For each chosen library or tool we also describe a reason why the tool was chosen, since one always has to be aware of both advantages and disadvantages carried by choosing a pattern or tool. Usage of those patterns and tools is also demonstrated on practical examples.

Very important choice for each application is a choice of architecture. To use the right architecture for the application can mean an overall easier development process. But if one chooses architecture unsuited for specific application, development can become quite unpleasant and expensive. So we try to emphasize the importance of right decisions early in the application development.

KEYWORDS:
design patterns, principles, extensibility, multilayer architecture, object-relational mapping

Uvod

Smo v dobi, ko so računalniške aplikacije tako ali drugače del našega življenja. Vsaka aplikacija je specifična, vse pa gredo čez določene stopnje v svojem razvoju: načrtovanje, implementacija, testiranje in dokumentiranje ter seveda vzdrževanje. Pri tem je seveda izvedba vsake naslednje stopnje vsaj v določeni meri odvisna od prejšnjih. Ker pa je vzdrževanje zadnja stopnja v razvoju aplikacije, se ponavadi tam pokažejo vse napake skozi celoten razvoj aplikacije. Tako lahko slabo načrtovanje, neustrezna implementacija in druge napake privedejo do tega, da je aplikacijo zelo težko vzdrževati, kar prinese tudi veliko večje stroške.

S problemom razvoja aplikacij se srečujemo vsi razvijalci, na srečo pa so vsaj nekateri problemi v večji ali manjši meri rešeni. Obstajajo tako imenovane dobre prakse (*best practices*), vzorci (*design patterns*) in orodja, ki so nam lahko v pomoč pri samem razvoju ali pri reševanju določenega problema.

Namen je, seznaniti se s tem, kaj je pri načrtovanju aplikacij sploh pomembno in kakšni pristopi in vzorci so nam na voljo za reševanje problemov pri razvoju. Ob tem bomo spoznali tudi nekatera orodja, ki so zgrajena na podlagi določenih vzorcev oziroma olajšujejo vpeljavo vzorcev v našo aplikacijo.

Glavni cilj diplomske naloge je preučiti prej omenjene vzorce, pristope in orodja ter njihovo uporabo predstaviti na konkretnem primeru. S tem želim dati smernice ostalim razvijalcev, ki se lotevajo razvoja aplikacije, pa morda ne vedo, kako bi se lotili določenih problemov. Želim pa tudi poudariti dejstvo, da je neupoštevanje določenih vzorcev lahko zelo boleče pri vzdrževanju same aplikacije in da se večinoma ne obrestuje, če sami iščemo rešitev za nek splošno znan problem.

1. Razvoj aplikacij

Sam razvoj aplikacij predstavlja svojevrsten izziv. Glavna problema pri tem sta, kako aplikacijo razviti hitro in kvalitetno. Skozi čas se je razvilo precej različnih metodologij razvoja aplikacij, npr: slapovni model, prototipni model, inkrementalni razvoj, agilni razvoj itd. Vsem tem metodologijam pa so skupne posamezne aktivnosti znotraj modela, ki so bile omenjene že v uvodu: načrtovanje, implementacija in vzdrževanje.

1.1. Načrtovanje

Pri načrtovanju aplikacije igra načrtovanje zelo pomembno vlogo, saj napačno načrtovana aplikacija lahko zelo oteži tako izdelavo kot tudi njeno vzdrževanje. Po drugi strani pa dobro načrtovanje lahko izdelavo tudi poenostavi.
Pri načrtovanju aplikacij je potrebno upoštevati, katere lastnosti so za konkretno aplikacijo pomembne. Pomembnost lastnosti je odvisna od namena aplikacije. Nekatere lastnosti so:

- Združljivost: aplikacija je združljiva z drugimi produkti, ki so zasnovani za delo z različnimi aplikacijami.
- Razširljivost: aplikaciji lahko dodamo nove možnosti brez velikih sprememb v sami aplikaciji.
- Odpornost na napake: aplikacija je odporna na napake in se lahko ob odpovedi določenega dela sistema izvaja naprej.
- Modularnost: končni produkt je sestavljen iz posameznih, dobro definiranih komponent oz. modulov, kar prinese s sabo tudi lažje vzdrževanje.
- Zanesljivost: aplikacija lahko neko funkcijo izvaja določen čas pod določenimi pogoji.
- Varnost: aplikacija je odporna na napade od zunaj.
- Uporabnost: uporabniški vmesnik mora biti enostaven za uporabo v neki ciljni skupini.
- ...

Poleg teh lastnosti, ki vplivajo na samo zasnovo aplikacije, je pomembna ustrezna izbira arhitekture aplikacije. Na izbiro arhitekture do neke mere vplivajo tudi prej omenjene lastnosti: aplikacija, ki mora biti varna, združljivost pa ni pomembna, ima običajno vsaj nekoliko drugačno arhitekturo od aplikacije, ki mora biti združljiva s tremi drugimi produkti.
Kaj točno je aplikacijska arhitektura in zakaj je zares pomembna, na kakšne probleme lahko naletimo pri načrtovanju arhitekture in kako te probleme rešiti? Odgovore na to bom poskusil podati v poglavju 2.

1.2. Implementacija

Faza implementacije vsebuje tri aktivnosti: implementacijo, testiranje in dokumentiranje.
Implementacija je v našem primeru kodiranje aplikacije v izbranem programskem jeziku. Tu se lahko srečamo z izzivi kot so kako implementirati rešitev določenega problema, kako določen algoritem narediti bolj učinkovit, lažji za uporabo ipd.
Glavni problem je, kako pisati kodo, ki bo čim bolj zanesljiva. Tu pride v ospredje testiranje – to je, *testiranje* programske kode oziroma delov programske kode. Poznamo več vrst testov, eno so testi enot (*unit test*), ki testirajo posamezno komponento sistema. Drugi testi pa so integracijski (*integration test*), ki testirajo sodelovanje med komponentami. Tako testi enot kot integracijski testi spadajo pod ti. avtomatizirane teste, torej teste, ki se izvajajo avtomatsko (npr. na vsako uro, ob vsaki spremembi kode v glavni veji ipd.).Določenih delov (predvsem uporabniški vmesnik) se tako ne da testirati oziroma je avtomatsko testiranje precej težje, tu pride do izraza ročno testiranje. Večino naše kode pa bo testirane preko avtomatskih testov. Več ko imamo testov, z večjo gotovostjo lahko trdimo, da je koda stabilna

in lažje tudi zamenjano oziroma popravljamo določene module. Če za določen modul obstajajo testi in se spremeni implementacija modula, nam testi vseeno zagotavljajo, da se samo delovanje na zunaj ni spremenilo. Obratno seveda velja, če testov nimamo – vsaka sprememba lahko s sabo prinese veliko nepredvidljivih posledic. Feathers celo pravi, da je vsaka koda brez testov koda, ki ni vzdrževana, ti. *legacy code* [**2**].
Kot so pomembni testi, pa je za vzdrževanje pomembna tudi ustrezna dokumentacija kode. Tako je potrebno ustrezno dokumentirati programski vmesnik (API) nekega modula kot tudi njegovo interno delovanje. Brez dokumentacije je vzdrževanje kode veliko težje. Pri tem je pomembno tudi, da je dokumentacija ustrezna. Zato je ob spremembi delovanja potrebno popraviti tudi dokumentacijo, za kar pa je potrebno nekaj discipline. Potrebno se je zavedati, da je bolje pisati dobro in pregledno kodo, s čimer se določenim komentarjem lahko tudi izognemo.

1.3. Vzdrževanje

Po končani implementaciji in po tem, ko je celotna aplikacija ustrezno stestirana, se začne faza vzdrževanja. V tej fazi je potrebno odpraviti odkrite napake kot tudi dodajanje nove funkcionalnosti. Glede na raziskave je celo 80% vzdrževanja dodajanje nove funkcionalnosti (Pigosky, 1997).
Če ne prej, se tu pokaže veliko napak in napačnih pristopov med načrtovanjem in implementacijo aplikacije. Po eni strani nam tako velika količina napak lahko pove, da naša koda ni dovolj robustna in predvsem premalo testirana. Težave pri dodajanju nove funkcionalnosti pa kažejo na to, da naš sistem ni dovolj modularen in razširljiv. Težave z napakami lahko v veliki meri rešimo z ustreznim testiranjem, težave z dodajanjem funkcionalnosti pa z upoštevanjem določenih navodil in pristopov pri samem razvoju. Kakšni ti pristopi so in čemu so namenjeni, bom poskušal odgovoriti v 3. poglavju.

2. Arhitektura

Arhitekturo programa je potrebno določiti že v fazi načrtovanja in tako predstavlja osnovo za nadaljni razvoj. Torej preden lahko uporabimo vzorce in pristope, se je potrebno odločiti za arhitekturo. Tu pa se, tako kot pri pristopih, pojavlja več vprašanj.

2.1. Kaj sploh je arhitektura programa?

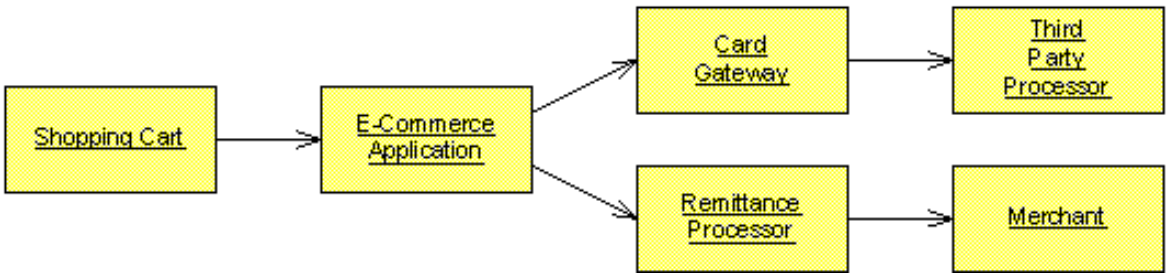
Neke univerzalne definicije, kaj je arhitektura programa (*software architecture*), ni. Je pa dejstvo, da ima vsaka programska rešitev arhitekturo. Razlika je samo v tem ali se definira vnaprej ali pa nastaja sproti, med samim razvojem. Potrebno se je zavedati, da je arhitektura zelo pomembna, kajti prava arhitektura tlakuje pot do uspeha, napačna pa v veliki večini primerov vodi do take ali drugačne katastrofe [**11**]. Zakaj? Arhitektura predstavlja prve načrtovalske rešitve. Posledično je te rešitve kasneje zelo težko (in drago) spreminjati.
Poenostavljeno rečeno pa lahko arhitekturo programa definiramo kot temeljno organizacijo sistema, sestavljenega iz delov sistema, povezav med deli in okoljem ter principov, ki usmerjajo načrtovanje in razvoj sistema [**8**]. Po pozornem branju se taka definicija lahko razbije na tri osnovne dele:

- Struktura - deljenje odgovornosti med posameznimi deli sistema,
- Vedenje (*behavior*) - razmerja in interakcije med deli sistema,
- Slog - principi in vodila, ki so bila vzrok za arhitekturo in bodo veljala tudi v naprej.

Poglejmo si vsakega od teh delov nekoliko podrobneje, tudi s pomočjo preprostih primerov.

2.1.1. Struktura

Srukturni elementi arhitekture predstavljajo statičen nabor delov, ki tvorijo sistem. Hkrati predstavljajo tudi odgovornosti teh delov in njihov položaj v celotni organizaciji sistema. Pomemben vidik strukture so odvisnosti med deli in zunanjimi sistemi, sama organiziranost arhitekture iz strukturnega vidika pa je pomembna tudi za načrtovanje projekta in koordinacijo dela. Projektna skupina je večinoma organizirana na podlagi strukture - vsaka skupina je odgovorna za svoj del sistema.

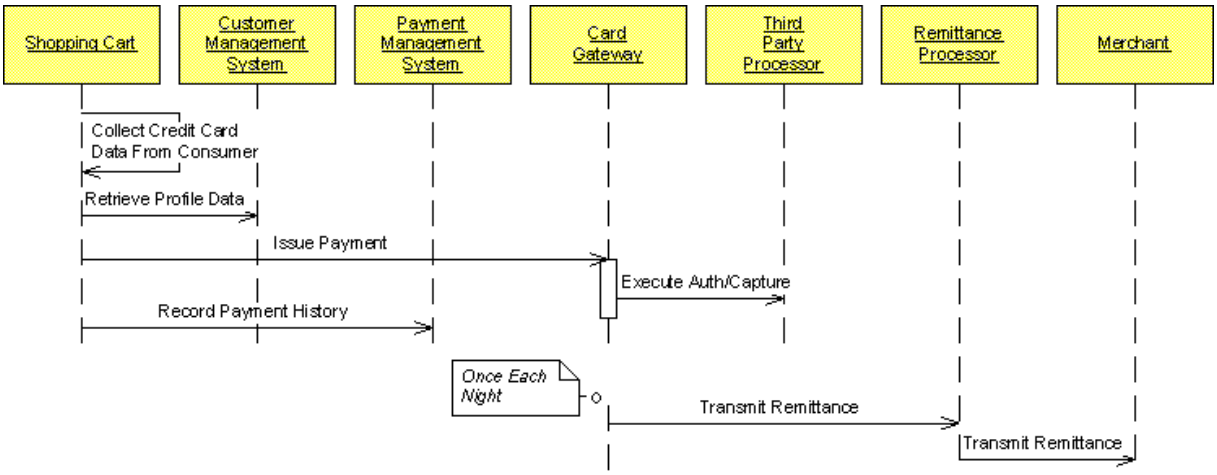


Slika 1: Primer statičnega dela arhitekture

Zgornji diagram prikazuje zelo splošen prikaz strukture teoretične elektronske trgovine, ki sprejema tudi kreditne kartice. Tak prikaz služi kot pomoč pri postavljanju samega sistema in pri sprejemanju odločitev glede arhitekture.

2.1.2. Vedenje

Vedenjski del arhitekture predstavlja načine, na katere lahko deli sistema komunicirajo med sabo, da bi bile dosežene zahteve sistema. Diagrami, kot je prikazan tudi spodaj, predstavljajo "sporočila" med deli sistema.



Slika 2: Primer dinamičnega dela arhitekture

Tu lahko spremljamo sistem v delovanju.

2.1.3. Slog

Slog zajema principe, vodila in vzorce, na podlagi katerih sta narejena tudi struktura in vedenje arhitekture. Slog zajema tudi koncepte kot so arhitekturni vzorci, tehnološki principi (npr. oddaljeni klici) in vrednote (npr. preprostost načrta).
Slog predstavi zbirko pravil (lahko tudi omejitev), ki vzpodbujajo učinkovito sprejemanje odločitev in lajšajo komuniciranje glede arhitekture. Zelo važno je, da je slog eksplicitno

definiran, saj nadaljni razvoj arhitekture temelji na razumevanju razlogov, ki so vplivali na trenutno arhitekturo.

2.1.4. Definiranje arhitekture v praksi

Grafi, omenjeni v prejšnjih točkah, služijo predvsem kot sredstvo za komunikacijo med programskim arhitektom in ekipo razvijalcev. Da pa se arhitekturo sploh lahko definira, mora arhitekt v praksi večinoma [**7**]:

- vedeti, katere so možnosti za dosego cilja (npr. poznavanje različnih tehnologij),
- izmed vseh možnosti izbrati tiste, ki največ prispevajo k rešitvi problema (vsaka možnost ima svoje prednosti in slabosti),
- izbrati ustrezno tehnologijo (najnovejša tehnologija ni nujno tudi najboljša za določen primer),
- paziti, da izbira tehnologije ne narekuje tudi definicije same arhitekture,
- pretehtati, če se določene dele sistema ali pa kar celotno ogrodje zares splača razvijati (obstaja že precej narejenih in stestiranih aplikacijskih ogrodij),
- definirati plasti za sistem (pripomore k večji modularnosti, ni pa vedno potrebno)
- slediti standardom, kjer je to smiselno (včasih standard narekuje tudi stvari, ki za določen projekt niso smiselne, npr. zapleteno dostopanje do datotečnega sistema, zapletena komunikacija navzven ipd.),
- poznati kontekst problema (rešitev bo težko dobra, če niso znane zahteve kot so performance, skalabilnost, varnost ipd.),
- težiti k enostavnim rešitvam.

2.2. Plasti

Koncept plasti se velikokrat uporablja namesto nivoja in obratno. Vendar sta si pojma različna, plast večinoma pomeni logično delitev elementov, ki tvorijo sistem, nivo pa pomeni fizično delitev sistema [**1**]. Vzorec plasti se uporablja za učinkovito upravljanje z odvisnostmi med deli sistema ter za izgradnjo razširljivih komponent. Praktična uporabnost večplastne aplikacije je, da z uporabo dobro definiranih vmesnikov in upravljanja medodvisnosti pridobimo na prožnosti same aplikacije. Največkrat uporabljan vzorec je tri-plastna aplikacija, ki definira:

- predstavitvena plast
- poslovna plast
- podatkovna plast

Prisotnih je sicer lahko še več plasti, so pa te tri plasti skoraj vedno prisotne v večjih poslovnih aplikacijah. Vsaka izmed naštetih plasti je opisana v naslednjih poglavjih.

2.2.1. Predstavitvena plast

Večina predstavitvenih plasti je sestavljena iz ti. obrazcev ali strani, preko katerih uporabnik komunicira s sistemom. Vsak obrazec vsebuje več polj, ki prikazujejo podatke iz nižjih nivojev in zbirajo uporabnikove vhodne podatke. V ogrodju .NET so za to običajno uporabljene forme iz System.Windows.Forms, za spletne aplikacije pa ASP.NET komponente.

2.2.2. Poslovna plast

Poslovna plast je običajno sestavljena iz konceptov poslovnih procesov in poslovnih komponent. Ti koncepti so realizirani preko različnih komponent, entitet, vmesnikov ipd.

Poslovne komponente

Poslovne komponente so glavna enota načrtovanja, implementacije in vzdrževanja v življenjskem ciklu poslovne aplikacije. Vsebujejo poslovno logiko - poslovna pravila. Pravila omejujejo potek dogodkov tako, da ustreza potrebam določenega podjetja. Npr., poslovno pravilo, ki določa, če je neka stranka upravičena do posojila, je lahko realizirano kot del poslovne komponente za delo s strankami.

Poslovni procesi

Poslovni procesi predstavljajo ponavljujoče se aktivnosti za določeno podjetje (npr. procesiranje naročila, pomoč strankam ipd.). Ti procesi so vsebovani v poslovnih *workflow* komponentah. Določena podpora procesom je v ogrodju .NET že prisotna (Windows Workflow Foundation), uporabi pa se lahko tudi BizTalk oziroma se podporo procesom implementira na novo.

Poslovne entitete

Poslovne entitete so nosilci podatkov, katerih namen je, da skrijejo konkreten format podatkov (npr. XML, podatkovna baza ipd.) in omogočijo lažji razvoj. Npr. določena komponenta lahko nastavi vrednost poslovne entitete, ki predstavlja stanje komponente.
Poslovne entitete so velikokrat uporabljene tudi kot nosilci podatkov preko plasti ali nivojev (DTO). Podatkovna plast večinoma vrača poslovne entitete in ne strukture, ki so specifične za podatkovno bazo. Tak pristop skrije podrobnosti, ki se tičejo samo podatkovne baze, tako da se tega višje plasti ne zavedajo.

Strežniški vmesniki

Aplikacija lahko določeno funkcionalnost odpre navzven drugim aplikacijam. Strežniški vmesnik predstavlja to funkcionalnost zunanjemu svetu. Če je mogoče, so podrobnosti implementacije skrite, tako da so preko vmesnika na voljo samo dobro definirane poslovne metode.

2.2.3. Podatkovna plast

Večina poslovnih aplikacij uporablja podatke, ki so shranjeni v podatkovni bazi, večinoma so to relacijske podatkovne baze. Komponente za dostop do podatkov so odgovorne za to, da poslovni nivo lahko dostopa do teh podatkov.
Komponente za dostop do podatkov so namenjene temu, da se podatkovni plasti ni potrebno zavedati podrobnosti podatkovne baze, kar ima naslednje prednosti:

- Minimizira vpliv sprememb ob menjavi podatkovne baze,
- Minimizira vpliv sprememb ob menjavi predstavitve podatkov (npr. sprememba podatkovne sheme),
- Združuje vso logiko za upravljanje s podatki določenega tipa na enem mestu. To precej olajša testiranje in vzdrževanje.

V ogrodju .NET je za dostop do podatkovnih baz na voljo tehnologija ADO.NET, lahko pa se npr. uporabi tudi O/RM orodje, več o tem v poglavju 3.7.

Poleg tri-plastne arhitekture pa se vedno več uporablja tudi štiri-plastna arhitektura z naslednjimi plastmi:

- Plast uporabniškega vmesnika (predstavitvena plast)
- Aplikacijska plast (tudi strežniška plast)
- Domenska plast (poslovna plast)
- Infrastrukturna plast (podatkovna plast).

Tukaj je poleg predstavitvene, poslovne in podatkovne plasti dodana še aplikacijska oziroma strežniška plast. Aplikacijska plast je velikokrat kar del poslovne plasti, včasih pa je lahko tudi ločena in je namenjena koordiniranju več različnih poslovnih komponent.
Vsaka plast se sicer lahko deli še naprej, npr. predstavitvena plast se lahko deli na prezentacijsko in plast uporabniškega vmesnika (tak pristop uporablja vzorec Model-View-Presenter), domenska plast lahko uporablja poslovno infrastrukturno plast kot vmesno plast med sabo in infrastrukturno plastjo (taka plast je ponavadi splošno uporabna) itd. Večinoma pa so dovolj tri oziroma štiri plasti.

3. Pristopi in vzorci

Kot že rečeno, so določeni problemi pri razvoju aplikacij precej splošni in za take probleme obstajajo različne rešitve. Rešitvam za določen problem, ki so splošno uporabne, pravimo vzorci ali tudi načrtovalski vzorci (*design patterns*).
Po drugi strani pa pristop k razvoju oziroma princip (*principle*) večinoma igra vlogo nasveta – torej za dosego nekega cilja je določen princip priporočljivo upoštevati. Obstaja pet principov, za katere je splošno sprejeto, da naj bi bili osnova razvijalcem. To so ti. SOLID principi [**16**].

	Kratica	**Razlaga**
S	SRP	Single responsibility principle - Princip ene odgovornosti
O	OCP	Open/closed principle - Princip odprto/zaprto
L	LSP	Liskov substitution principle - Razvoj po pogodbi
I	ISP	Interface segregation principle - Princip ločevanja vmesnikov
D	DIP	Dependency injection principle - Obračanje nadzora in vstavljanje odvisnosti

Tabela 1: SOLID principi

3.1. Princip ene odgovornosti

Eno izmed glavnih vodil dobrega programiranja je ti. princip ene odgovornosti *(single responsibility principle*). Princip pravi, da mora vsak objekt imeti samo eno odgovornost in da naj bo ta odgovornost v celoti zajeta v objektu.
Termin je predstavil Robert C. Martin, ki je v knjigi Principles of Object Oriented Design odgovornost definiral kot razlog za spremembo in iz tega izhaja, da naj bi modul imel en in samo en razlog, da se spremeni. Primer, npr. za modul, ki pripravi in tiska poročila. Tak modul se lahko spremeni iz dveh razlogov. Prvi razlog je, da se lahko spremeni vsebina poročila. Drugi pa, da se lahko spremeni oblika poročila. To sta dva precej različna razloga, dve različni odgovornosti. Tako bi torej morali biti ločeni v dva modula.
Razlog, zakaj je pomembno v glavi vedno imeti princip ene odgovornosti je, da je tak objekt precej bolj robusten in pripelje do ti. deljenja odgovornosti

3.2. Deljenje odgovornosti

Deljenje odgovornosti (*separation of concerns*) je proces ločevanja programa na ločene enote, ki se funkcionalno čim manj prepletajo. Deljenje odgovornosti je običajno doseženo preko modularnosti in enkapsulacije s pomočjo skrivanja informacij (skrivanje konkretne implementacije za vmesniki). Tudi večplastni dizajn je večinoma baziran na deljenju odgovornosti.
Deljenje odgovornosti je sicer pomemben princip tudi na drugih področjih, ne samo pri programiranju. Taka področja so npr. arhitektura, načrtovanje informacijskih sistemov, ... Cilj je načrtovati take sisteme, da je njihove funkcije mogoče optimizirati neodvisno od drugih funkcij. Tako napaka ene funkcije ne povzroči napak v drugih funkcijah, izboljša pa se tudi razumevanje in upravljanje kompleksnih sistemov.

3.3. Princip odprto/zaprto

Še eno izmed zelo pomembnih vodil pri razvoju aplikacij je princip odprto-zaprto [**13**], ki pravi, da mora biti objekt odprt za razširjanje in zaprt za spreminjanje. To pomeni, da naj bi bilo objekt možno razširiti brez sprememb v izvorni kodi objekta. Posebej uporabno se to izkaže v okolju, kjer spremembe v izvorni kodi pomenijo tudi pregledovanje kode, testiranje ipd. Pravilo oziroma princip je sicer precej preprosto, ampak je velikokrat ignorirano, kar ima za posledico grdo kodo, ki jo je težko vzdrževati.
V nadaljevanju sta podana dva primera, kjer se pravilo najpogosteje ignorira in tudi pravilen pristop z upoštevanjem pravila.

3.3.1. Primer 1 - Switch

Tipičen primer kršenja principa odprto-zaprto je uporaba switch ukaza. To sicer ne pomeni, da je switch ukaz sam po sebi "slab", je pa potrebno dobro premisliti, kdaj in kje ga uporabiti. Naslednja koda je lep primer objekta, ki ni razširljiv.

```
public enum AnimalType {
  Cat, Dog, Wolf, Bear
}

public class ClosedAnimal {
  private AnimalType _type;

  public AnimalType AnimalType {
    get { return _type; }
  }

  public ClosedAnimal(AnimalType type) {
    _type = type;
  }

  public string Noise() {
    switch (_type) {
      case AnimalType.Cat:
        return "Meow";
      case AnimalType.Dog:
        return "Bark";
      case AnimalType.Wolf:
        return "Howl";
      case AnimalType.Bear:
        return "Growl";
      default:
        throw new InvalidOperationException("Unrecognized type.");
    }
  }
}
```

Koda, ki ta razred uporablja, pa je naslednja:

```
static void Main(string[] args) {
  var animal = new ClosedAnimal(AnimalType.Dog);
  Console.WriteLine(animal.Noise());

  Console.ReadLine();
}
```

Problem zgornje kode je v tem, da če želimo dodati nov tip živali, bo potrebno spremeniti metodo Noise(), spremeniti pa jo bo potrebno ob vsakem dodajanju nove živali. Ob taki arhitekturi je tudi zelo verjetno, da bo poleg metode Noise() potrebno spremeniti še precej drugih metod, ki jih bo prav tako potrebno spremeniti. Če se pozabi na samo eno tako metodo, bo naš program v bistvu nedelujoč. Kot vidimo, tak pristop vsekakor ni odprt za razširitve.
Možna rešitev bi bila naslednja:

```
public abstract class OpenAnimal {
  public abstract string Noise();
}

public class Cat: OpenAnimal {
  public override string Noise() {
    return "Meow";
  }
}

public class Dog: OpenAnimal {
  public override string Noise() {
    return "Bark";
  }
}
```

Koda z uporabo pa je:

```
static void Main(string[] args) {
  var animal = new Dog();
  Console.WriteLine(animal.Noise());

  Console.ReadLine();
}
```

Torej kadarkoli je potrebno dodati novo žival, moramo samo implementirati nov razred, ki deduje iz razreda OpenAnimal in implementira potrebne metode. Tako se sprememba zgodi samo na enem mestu (npr. v OpenAnimalFactory, če uporabljamo vzorec tovarne), razred pa je odprt za razširitve.

3.3.2. Primer 2 - preverjanje tipov med izvajanjem

Drug, nekoliko težje opazen primer kršenja, je preverjanje tipov med izvajanjem programa. Sicer je v svojem bistvu podobna switch ukazu, le da so tu uporabljeni if-else konstrukti.

```
public class Dinosaur: OpenAnimal {
  public string Eat(OpenAnimal animal) {
    if (animal is Dog ||
        animal is Wolf ||
        animal is Bear)
      return "Yumyum";
    else if (animal is Cat)  // dinosaurs don't like cats
      return "Yuck";
    else
      return string.Empty;   // do dinos eat other dinos?
  }

  public override string Noise() {
```

```
    return "Roar";
  }
}
```

Kot je razvidno, bo zopet ob vsaki novi živali potrebno spremeniti tudi razred Dinosaur. Možna rešitev bi bila, da se v razred OpenAnimal doda še ena metoda, ki za vsako žival pove, če jo lahko poje velik plazilec.

```
public abstract class OpenAnimal {
  public abstract string Noise();
  public abstract bool IsTastyToLargeLizards { get; }
}

public class Cat: OpenAnimal {
  public override string Noise() {
    return "Meow";
  }

  public override bool IsTastyToLargeLizards {
    get { return false; }
  }
}

public class Dog: OpenAnimal {
  public override string Noise() {
    return "Bark";
  }

  public override bool IsTastyToLargeLizards {
    get { return true; }
  }
}

public class OpenDinosaur: OpenAnimal {
  public string Eat(OpenAnimal animal) {
    if (animal.IsTastyToLargeLizards)
      return "Yumyum";
    return "Yuck";
  }

  public override string Noise() {
    return "Roar";
  }

  public override bool IsTastyToLargeLizards {
    get { return false; }
  }
}
```

To so sicer precej poenostavljeni primeri, ki pa precej nazorno pokažejo, na kaj je pri razvoju razširljivih aplikacij potrebno (tudi) paziti.

3.4. Razvoj po pogodbi

Design by contract oziroma razvoj po pogodbi pravi, da naj bi razvijalci definirali formalne, natančne in preverljive specifikacije za vmesnike komponent. Te specifikacije so poznane kot pogodbe (*contracts*).

Glavna ideja razvoja po pogodbi je v prispodobi, kako naj bi elementi sistema sodelovali med seboj. Prispodoba prihaja iz poslovnega sveta, kjer stranka in ponudnik med seboj poslujeta glede na podpisano pogodbo, ki določa npr. katere produkte ponudnik nudi stranki ter za kakšno ceno. Podobno lahko določena metoda razreda v objektnem programiranju ponuja neko funkcionalnost, lahko pa pričakuje, da bodo vhodni parametri v nekem določenem formatu, zagotavlja določen format rezultata kot izhod ipd. Podedovani razredi lahko pogodbo spremenijo, vendar samo do določene mere. Lahko omilijo pogoje ter dodajo novo vrednost ob izvrševanju pogodbe. Npr. neka metoda lahko namesto števil od 1 do 10, kar določa pogodba osnovnega razreda, sprejme števila od 1 do 20 – pogoji so omiljeni. Ne sme pa podedovani razred spremeniti pogodbe v tolikšni meri, da pogodba nadrejenega razreda ne bi bila več veljavna.
Na bolj konkretnem primeru princip pravi naslednje za razreda Client in Supplier. Client razred lahko kliče metode Supplier razreda, Supplier razred pa ob tem klicu ne bo prešel v nedovoljeno stanje. Poleg tega je Supplier primoran kot rezultat vrniti neko stanje in podatke, ki so v skladu s pričakovanji Client razreda glede na pogodbo.
Če ima nek razred dobro definirano pogodbo, ga je posledično precej lahko testirati (glej poglavje 3.12). Tako s testi enote lahko testiramo posamezen modul in da ustrezno izpolnjuje pogodbo, z integracijskimi testi pa da moduli ustrezno sodelujejo med seboj.

3.4.1. Liskovin princip zamenjave

Princip zamenjave, ki ga je definirala Barbara Liskov, je precej podoben razvoju po pogodbi. Oba principa se ukvarjata s tem, kako naj razvijalec gleda na obnašanje nekega odvisnega razreda, če se zamenja tip tega odvisnega razreda. Konkretno, če je razred A odvisen od razreda B, tega pa dedujeta razreda C in D, kako naj zamenjava C z D vpliva na A in koliko se mora tega zavedati razred B.
Razlik med principom zamenjave in razvojem po pogodbi je ta, da je razvoj po pogodbi usmerjen bolj praktično, princip zamenjave pa bolj teoretično. Skupni cilj obeh pa je, da razvijalcu pomagata pri tem, da ne izdela razredov, ki bi implementirali funkcionalnost, ki bi bila v nasprotju s tisto, definirano z nadrejenim razredom.

3.5. Princip ločevanja vmesnikov

Princip se uporablja za ti. čisti razvoj in razvijalcu pomaga pri izogibanju temu, da je aplikacijo nemogoče ali vsaj zelo težko spremeniti. Sledenje principu pomaga pri tem, da sistem ostane nepovezan in posledično lažji za spreminjanje. Princip pravi, da ko vmesnik postane »predebel«, ga je potrebno razbiti na več manjših, bolj specifičnih vmesnikov. Tako se je uporabnikom tega vmesnika zavedati samo manjšega dela in tako ni potrebe o odvisnosti od delov, ki jih ne rabi.
Primer »debelega« vmesnika:

```
public interface ICar {
  void PaintDoors(Color color);
  void PaintRoof(Color color);
  void PaintSpoiler(Color color);
  void Accelerate(int delta);
  void StartEngine();
}
```

Tak vmesnik se lahko razbije na dva manjša:

```
public interface ICar {
  void Accelerate(int delta);
  void StartEngine();
}

public interface ICarPainter {
  void PaintDoors(Color color);
  void PaintRoof(Color color);
  void PaintSpoiler(Color color);
}
```

Tako je lahko razred, ki hoče samo pobarvati avto, odvisen samo od vmesnika ICarPainter, ne pa od celotnega vmesnika ICar.
Sledenje temu principu gre z roko v roki s principoma vstavljanja odvisnosti in ene odgovornosti.

3.6. Obračanje nadzora in vstavljanje odvisnosti

3.6.1. Obračanje nadzora

Obračanje nadzora (*inversion of control*) je princip, ki opisuje pristop k načrtovanju arhitekture, kjer je tok dogajanja v sistemu obrnjen glede na tok dogajanja pri proceduralnem programiranju. Običajno je tok dogajanja pod nadzorom glavne enote programa. Pri obračanju nadzora pa je drugače. Klicatelj bo sicer še vedno dobil odgovor, kako in kdaj pa ni več pod njegovim nadzorom. To se odloči klicani. To je podobno kot Hollywoodski princip "You do not call us, we will call you back". Ta princip služi za doseganje naslednjih ciljev:

- Ločevanje izvajanja neke naloge od implementacije,
- Vsak sistem se lahko osredotoči na to, za kar je bil narejen,
- Noben sistem se ne zanaša na to, kaj naj bi delali ostali sistemi,
- Zamenjava enega sistema nima nobenega učinka na ostale.

Obračanje nadzora se lahko implementira na različne načine. Nekateri izmed njih so:

- uporaba vzorca tovarne
- uporaba service locator
- uporaba vstavljanja v konstruktorju
- uporaba vstavljanja preko setterja
- uporaba vstavljanja preko vmesnika

Vsem metodam pa je skupno ime vstavljanje odvisnosti.

3.6.2. Vstavljanje odvistnosti

Vstavljanje odvistnosti je ime za vstavljanje zunanje odvisnosti neki komponenti. Sam pristop se, kot rečeno, večinoma uporablja za dosego obračanja nadzora.
Vzorec zajema vsaj tri elemente: odvisnika, odvisnosti in injektorja oziroma posredovalca. Odvisnik je uporabnik, ki mora izvršiti neko nalogo v programu. Da lahko nalogo izvrši, potrebuje pomoč različnih servisov, ki izvedejo razne podnaloge. Posredovalec pa je komponenta, ki lahko odvisnika in odvisnosti združi, poleg tega pa upravlja tudi z njihovim življenjskim časom. Tak posredovalec je lahko implementiran kot tovarna (*factory*), iskalnik servisov (*service locator*), celotno ogrodje ipd.

Primer: Avto (odvisnik) je odvisen od motorja (odvisnost), da se lahko premika. Motor je izdelan pri izdelovalcu motorjev (posredovalec). Avto ne ve nič o tem, kako namestiti motor vase. Izdelovalec motor namesti, avto pa ga samo uporabi za premikanje. Torej, recimo da imamo naslednji definiciji:

```
public interface IEngine {
  void SetFuelValveIntake(int gasPedalPressure);
  int GetTorque();
}

public interface ICar {
  int DoAcceleratorPedalStep(int gasPedalPressure);
}
```

Spodaj je primer, ki ne uporablja vstavljanja odvisnosti:

```
public class SimpleEngine: IEngine {
  private int torque = 0;
  private int gasPedalPressure = 0;

  public void SetFuelValveIntake(int gasPedalPressure) {
    this.gasPedalPressure = gasPedalPressure;
  }

  public int GetTorque() {
    return torque;
  }
}

public class SimpleCar: ICar {
  private readonly IEngine engine = new SimpleEngine();

  public int DoAcceleratorPedalStep(int gasPedalPressure) {
    engine.SetFuelValveIntake(gasPedalPressure);
    int torque = engine.GetTorque();
    int speed = ...; // Calculate speed from torque
    return speed;
  }
}

static void Main(string[] args) {
  var car = new SimpleCar();
  var speed = car.DoAcceleratorPedalStep(5);

  Console.WriteLine("The car speed is " + speed);
}
```

Kot je razvidno, Car rabi instanco Engine-a, da lahko izračuna hitrost glede na stopnjo pritiska na pedal za plin. Tako bi npr. za različne tipe motorja rabili različen tip avta.
Z uporabo vstavljanja odvisnosti pa bi bila lahko implementacija taka:

```
public class Car: ICar {
  private readonly IEngine engine;

  public Car(IEngine engine) {
```

```
    this.engine = engine;
  }

  public int DoAcceleratorPedalStep(int gasPedalPressure) {
    engine.SetFuelValveIntake(gasPedalPressure);
    int torque = engine.GetTorque();
    int speed = ...; // Calculate speed from torque
    return speed;
  }
}

public class CarFactory {
  public static ICar BuildCar() {
    return new Car(new SimpleEngine());
  }
}

static void Main(string[] args) {
  var car = CarFactory.BuildCar();
  var speed = car.DoAcceleratorPedalStep(5);

  Console.WriteLine("The car speed is " + speed);
}
```

V zgornjem primeru CarFactory prevzame nalogo izdelave motorja, avta in vstavljanja motorja v avto. To avtu omogoči, da se mu ni več treba zavedati, kako izdelati motor, ampak ga lahko samo še uporablja. Res pa je, da se mora sedaj tovarna zavedati, kako izdelati motor. Lahko bi sicer izdelali še EngineFactory, ampak posledično to pomeni medodvisnost tovarn.
Obstaja tudi boljši način, ki še bolj avtomatizira proces in zmanjša medodvisnosti, ti. injektor, ki se običajno uporablja z ogrodji. Primer uporabe bi bil:

```
static void Main(string[] args) {
  var injector = new DependencyInjector();
  injector.Bind(typeof(ICar), typeof(Car));
  injector.Bind(typeof(IEngine), typeof(SimpleEngine));
  injector.Init();

  var car = injector.GetObject<ICar>(); // Asks injector for a car.
  var speed = car.DoAcceleratorPedalStep(5);

  Console.WriteLine("The car speed is " + speed);
}
```

V tem primeru je uporabljen injektor, ki ima definirane odvisnosti med vmesnikom in konkretno implementacijo razreda. Po inicializaciji injektor ve, da ko dobi zahtevo za avto, opazi da mora vrniti instanco tipa Car. Nadalje opazi, da le-ta zahteva instanco motorja, IEngine, za kar ima registriran tip SimpleEngine. Tako naredi instanco SimpleEngine in to poda naprej instanci Car, katero potem tudi vrne.
Tako obnašanje je tipično za večino injektor ogrodij. Pri bolj kompleksnih primerih se hitro pokaže, da je z uporabo takega ogrodja potrebno minimalno kode za upravljanje z veliko objekti in njihovimi odvisnostmi.

3.7. Objektno-relacijska preslikava

Objektno-relacijska preslikava oziroma na kratko O/RM je tehnika, namenjena lažjemu dostopu do podatkov, ki se nahajajo v relacijski bazi. Ponuja "preslikavo" med podatki v

relacijski bazi in objekti, uporabljenimi v katerem izmed objektno usmerjenih programskih jezikov. Problem je namreč v tem, da aplikacije tipično obdelujejo kompleksne objekte, večina relacijskih podatkovnih baz pa hrani skalarne vrednosti (npr. številke, znakovne nize ipd.). Tako je za uporabo podatkov v bazi le-te potrebno preslikati v polja prej omenjenih kompleksnih objektov oziroma v sami aplikaciji uporabljati samo preproste skalarne tipe. Objektno-relacijska preslikava se uporablja za reševanje prvega problema, torej preslikavo podatkov na polja posameznega objekta (in obratno).
Nekaj prednosti oziroma razlogov za uporabo O/RM:

- **Pohitritev razvoja** - Orodje O/RM sicer ne ponuja nobenih zelo naprednih pristopov in tehnologij, funkcionalnost, ki jo ponuja, pa se vsekakor lahko tudi implementira. Vendar je implementacija take funkcionalnosti večinoma precej nezanimivo in dolgotrajno delo. Namesto pisanja celotne kode je potrebno samo definirati ustrezne preslikave, kar vzame precej manj časa.
- **Bolje načrtovana koda** - Sicer je res, da koda, ki jo generira oziroma uporablja orodje O/RM ni popolna, sledi pa najbolj priznanim pristopom in uporablja vzorce, kjer je to potrebno, tako da je taka koda vedno konsistentna.
- **Ni potrebno biti strokovnjak za nek jezik za uporabo O/RM** - Koda, ki ponuja dostop do baze podatkov, je kritična za performance celotnega sistema. Če je ta koda slabo napisana, lahko močno vpliva na delovanje celotnega sistema. Tako samostojni razvoj dostopa do baze podatkov zahteva precej poznavanja jezika in dobrih praks. Z uporabo orodja O/RM pa se lahko osredotočimo na samo logiko persistiranih objektov in interakcije med njimi, orodje O/RM pa naredi ostalo.
- **Privarčevanje na času testiranja** - Ob razvoju lastne kode za dostop do baze podatkov, je le-to potrebno tudi temeljito testirati. Orodje O/RM pa je že stestirano, prav tako pa ga vsakič znova testirajo ostali uporabniki. Tako so tudi napake, ki jih sami tudi s testiranjem ne bi našli, že odpravljene.
- **Poenostavljen razvoj** - Z uporabo samostojnega pristopa za dostop do podatkovne baze smo obsojeni tudi na podrobnosti, kot so ADO.NET, COM+ ipd. Za uporabo O/RM orodja pa je potrebno poznati samo nekaj splošnih vmesnikov.

3.8. Učinkoviti vmesnik

Učinkoviti vmesnik (*fluent interface*) je način implementacije, ki ima za cilj bolj čitljivo kodo. Običajno je implementiran z nizanjem metod (method chaining), čeprav je učinkoviti vmesnik več kot to.
Spodaj je primer navadne implementacije:

```
public interface IConfiguration {
  void SetColor(Color color);
  void SetHeight(int height);
  void SetWidth(int width);
}

public class Configuration: IConfiguration {
  private Color _color;
  private int _height;
  private int _width;

  public void SetColor(Color color) {
    _color = color;
  }

  public void SetHeight(int height) {
```

```
    _height = height;
  }

  public void SetWidth(int width) {
    _width = width;
  }

}

public class Program {
  public static void Main(string[] args) {
    IConfiguration config = new Configuration();
    config.SetColor(Color.Blue);
    config.SetWidth(100);
    config.SetHeight(150);
  }
}
```

Enak primer z učinkovitim vmesnikom pa bi izgledal tako:

```
public interface IConfiguration {
  IConfiguration SetColor(Color color);
  IConfiguration SetHeight(int height);
  IConfiguration SetWidth(int width);
}

public class Configuration: IConfiguration {
  private Color _color;
  private int _height;
  private int _width;

  public IConfiguration SetColor(Color color) {
    _color = color;
    return this;
  }

  public IConfiguration SetHeight(int height) {
    _height = height;
    return this;
  }

  public IConfiguration SetWidth(int width) {
    _width = width;
    return this;
  }

}

public class Program {
  public static void Main(string[] args) {
    IConfiguration config = new Configuration()
                              .SetColor(Color.Blue)
                              .SetWidth(100)
                              .SetHeight(150);
  }
}
```

Tako je koda lahko precej bolj berljiva, sploh pri kakih bolj kompleksnih primerih. Tak pristop tudi ustvari interni (glede na razred) jezik (*Domain Specific Language*), ki še dodatno izboljša berljivost in kodo naredi bolj razumljivo. Prav DSL je razlog za uporabo termina fluent oziroma učinkovit. Več o tem na [**3**].

3.9. Vzorec shrambe

Ideja shrambe (*repository*) je v tem, da predstavlja vmesni člen med domeno in podatkovno bazo oziroma nivojem mapiranj ter ponuja metode za dostop do domenskih objektov. Vzorec je sicer najbolj primeren za kompleksne domenske modele, kjer sistem pridobi na enostavnosti, saj shramba skrbi za vso potrebno poizvedbeno kodo.
Shramba je na zunaj vidna kot zbirka objektov, shranjenih v spominu, objekti pa se lahko dodajajo, spreminjajo in odstranjujejo iz shrambe. Konceptualno gledano, vsebuje shramba nabor objektov, persistiranih v podatkovno bazo in omogoča izvajanje operacij nad njimi, kar ponuja bolj objektno usmerjen pogled na podatkovni nivo. Druga prednost shrambe pa je ločitev med domeno in podatkovno bazo oziroma nivojem mapiranj, kar pomeni, da se podatkovna baza oziroma orodje za mapiranje lahko zamenjava brez sprememb na višjih nivojih. To se pokaže kot prednost tudi pri testiranju kode, ko za potrebe testa lahko implementiramo svojo testno shrambo.
Primer shrambe za razred Product:

```
public interface IProductRepository {
  Product GetById(int productId);
  void Add(Product product);
  void Remove(Product product);
}

public class Program {
  public static void Main(string[] args) {
    var repository = ObjectFactory.GetInstance<IProductRepository>();
    var product = repository.GetById(1);
    Console.WriteLine(product.Name);
  }
}
```

Kot je razvidno iz primera, je običajna praksa, da se do shrambe dostopa preko tovarne. Prav uporaba tovarne nam omogoča morebitno zamenjavo podatkovne baze oziroma orodja za mapiranje z drugim orodjem, brez da bi bilo potrebno spreminjati zgornje nivoje.

3.10. Enota dela

Enota dela je eden izmed pogosteje uporabljanih vzorcev za persistiranje podatkov. Martin Fowler enoto dela definira tako: »Hrani listo objektov, uporabljenih v poslovni transakciji in koordinira zapisovanje sprememb ter rešuje težave s konkurenčnostjo«.
Enota dela ni nujno nekaj, kar razvijemo sami eksplicitno, vzorec se pokaže pri uporabi skoraj vsakega orodja za persistiranje podatkov – pri NHibernate je to ITransaction, pri Linq to SQL je to DataContext ipd.
Včasih pa je bolje implementirati enoto dela, ki je specifična za neko aplikacijo in interno vsebuje enoto dela uporabljenega orodja za persistiranje. Razlogov za implementacijo svoje enote dela je lahko več, npr. beleženje, upravljanje z napakami, upravljanje s transakcijami ipd.

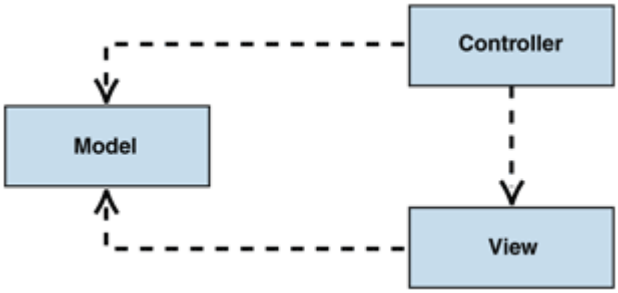
3.11. Model-View-Controller

Večina poslovnih aplikacij deluje tako, da sistem pridobi podatke iz podatkovne baze in jih prikaže uporabniku. Ko uporabnik spremeni podatke, sistem spremembe zopet shrani v podatkovno bazo. Ker je glavni tok podatkov med podatkovno bazo in uporabniškim vmesnikom, se lahko pojavi skušnjava, da se ta dva nivoja poveže in tako zmanjša količino kode in izboljša performance. Vendar pa tak pristop lahko povzroči precej problemov. En problem je, da se uporabniški vmesnik velikokrat spreminja, precej večkrat kot podatkovni nivo. Drug problem pa je da povezovanje podatkovnega nivoja in uporabniškega vmesnika posledično povzroči dodajanje poslovne logike na uporabniški vmesnik.
Ker se uporabniški vmesnik velikokrat spreminja, je zato bolje, da poslovna logika ni vključena v sam uporabniški vmesnik, temveč je od njega ločena. Če sta uporabniški vmesnik in poslovna logika v enem objektu, je namreč ob spremembi enega potrebno spremeniti tudi drugega. To potem lahko povzroči napake drugje v aplikaciji in zahteva testiranje celotne poslovne logike po vsaki spremembi.
Za uporabniški vmesnik je značilno tudi, da so za njegovo oblikovanje potrebna drugačna znanja kot za implementacijo poslovne logike. Tako je vsaj na večjih projektih redko, da ena oseba implementira tako poslovno logiko kot tudi uporabniški vmesnik.
Uporabniški vmesnik je tudi precej bolj odvisen od naprave, kot poslovna logika. Če je potrebno aplikacijo prenesti iz npr. brskalnika na PDA, je potrebno spremeniti precej kode uporabniškega vmesnika, kode poslovne logike pa pri tem ni potrebno spreminjati. Če sta torej ta dva nivoja ločena, je tak prenos precej bolj enostaven in predvsem hitrejši, prav tako pa tudi minimizira možnost vnosa napak na poslovni nivo.
Še en pomemben vidik pa je testabilnost - izdelava avtomatiziranih testov za uporabniški vmesnik je večinoma precej bolj zahtevna kot izdelava avtomatiziranih testov za poslovno logiko. Torej je zmanjšanje količine kode, ki je direktno odvisna od uporabniškega vmesnika, na mestu, saj precej olajša testiranje aplikacije.

Vse zgornje probleme lahko rešimo z uporabo vzorca Model-View-Controller. Vzorec ločuje modeliranje domene, prezentacijo in akcije glede na vnose uporabnika v tri ločene razrede:

- Model - Model upravlja z vedenjem in podatki aplikacijske domene, se odziva na poizvedbe o stanju in zahteve o spremembah stanja. Ob spremembah tudi obvesti vse poglede, da se ti lahko osvežijo (uporaba observer vzorca).
- View - View prikazuje stanje določene informacije.
- Controller - Nadzornik interpretira uporabniške vnose in obvešča model o potrebnih spremembah.

Slika prikazuje razmerja in odvisnosti med vsemi tremi objekti.



Slika 3: MVC razredna struktura

Pri tem je pomemben podatek, da sta tako pogled kot nadzornik odvisna od modela, model pa je od obeh neodvisen. To je ena glavnih prednosti ločevanja, saj dovoljuje da je model zgrajen in testiran neodvisno od vizualne prezentacije.

Sam vzorec je precej popularen in je en osnovnih vzorcev za ločevanje uporabniškega vmesnika in poslovne logike. Vendar pa se je ravno zaradi popularnosti vzorca pojavilo več napačnih opisov naziva "nadzornik". Na srečo pa uporaba vzorca v kontekstu web aplikacij pomaga pri reševanju tega problema, saj je meja med pogledom in nadzornikom zelo očitna.

3.11.1. Model-View-Presenter

Na podlagi vzorca MVC se je razvil nov vzorec, MVP. MVC ima namreč dve slabosti:

- Večja kompleksnost zaradi uporabe vzorca observer - da lahko nadzornik sporoči spremembe pogledu, mora za to spremeniti model, ki nato osveži pogled.
- MVC ne upošteva smernic pri modernem razvoju uporabniškega vmesnika. Danes namreč same kontrole skrbijo za ustrezen odziv na uporabniške akcije.

MVP ti dve slabosti rešuje tako, da lahko nadzornik komunicira neposredno s pogledom in tako sam povzroči osvežitev pogleda, ko je to potrebno, brez posredovanja modela. Hkrati pa MVP dopušča pogledom, da sami sprejemajo uporabniški vnos. MVP pozna dve variaciji, pasivni pogled in supervising controller. Več o teh variacijah je na voljo na [**19**].

3.12. Testi kot pomoč pri implementaciji

Testiranje programske kode, ki jo napišemo, je prav tako pomembno, kot sama programska koda. Pri tem obstajajo različni testi ter različni pristopi. V tem poglavju bom opisal vrsto testov, ki je po mojem mnenju najpomembnejša (vendar ne tudi edina pomembna) – unit testi oziroma testiranje enote (glej tudi poglavje 1.2).
Pri testiranju enote se za enoto smatra najmanjši del aplikacije, ki se lahko testira. V proceduralnem programiranju je to običajno ena funkcija, v objektnem pa razred.
Dobro napisani testi pripomorejo k hitrejšemu razvoju (time to market), testabilno napisana koda pa je tudi kvalitetnejša in veliko lažje razširljiva. Razširljivost je posledica dobro testabilne kode, kajti vsak test enote naj bi bil neodvisen od ostalih testov. Tak cilj pa je najlaže doseči z ustreznim dizajniranjem aplikacije - deljenjem odgovornosti med posameznimi razredi ter uporabo vmesnikov. Tak pristop se imenuje dizajniranje po pogodbi, kjer pogodbo predstavlja vmesnik. Testi se potem lahko naslanjajo na vmesnik, velikokrat pa s pomočjo testov lahko vmesnik tudi definiramo.
Prednost uporabe vmesnikov pri testiranju je v tem, da se ena enota oziroma v našem primeru razred, pogosto naslanja tudi na druge razrede. Če bi tako tak razred hoteli testirati, bi posledično morali testirati tudi vse razrede, od katerih je testirani razred odvisen. Če pa uporabimo ustrezen pristop in programiramo proti pogodbi oziroma vmesnikom, lahko pri testiranju namesto konkretnih razredov uporabimo nadomestke, ki te vmesnike implementirajo. Te nadomestke potem lahko nadzorujemo v samem testu. Takemu pristopu se reče mocking, nadomestnim razredom, ki implementirajo ustrezne vmesnike in ki jih podtaknemo razredu namesto pravih, pa mocki. Pristop, kjer programiramo proti vmesnikom in ne proti konkretnim razredom in kjer razred dobi ustrezno implementacijo od zunaj, pa je že opisan – obračanje nadzora in vstavljanje odvisnosti, poglavje 3.6.
Če vzamem primer iz poglavja 3.3, če bi hoteli testirati razred SimpleCar, bi hkrati s tem testirali tudi razred SimpleEngine. Tega ne želimo, saj hočemo testirati samo avto, neodvisno od motorja. Ob uporabi vstavljanja odvisnosti pa to lahko naredimo, torej testiramo razred Car tako:

```
// Najprej naredimo nadomestek za motor.
var engine = new Mock<IEngine>();
// Nadomestek bo pri klicu metode IEngine.GetTorque() kot rezultat vrnil 2
engine.Setup(e => e.GetTorque()).Returns(2);
// Sedaj naredimo instanco razreda, ki ga bomo testirali.
```

```
var car = new Car(engine.Object);

// Testiramo metodo DoAcceleratorPedalStep.
var result = car.DoAcceleratorPedalStep(5);

// Preverimo rezultat.
Assert.AreEqual(..., result);
// Preverimo lahko tudi, če je klic metode DoAcceleratorPedalStep povzročil
// tudi klic metode IEngine.SetFuelValveIntake
engine.Verify(e => e. SetFuelValveIntake(5));
```

4. Primer implementacije

V tem poglavju želim na konkretnem primeru prikazati uporabo pristopov in vzorcev, opisanih v prejšnjem poglavju. Določeni pristopi so uporabljeni preko orodij - konkretno sta to orodje NHibernate za objektno-relacijsko preslikavo in ASP.NET MVC, ki je ogrodje s podporo MVC.
Pri implementaciji sem uporabil več različnih vzorcev in se tako seznanil z njihovimi prednostmi, pa tudi slabostmi. Treba pa se je zavedati, da vsi tukaj omenjeni pristopi niso primerni za vsako aplikacijo. Primernost tehnologij in pristopov je potrebno upoštevati pri načrtovanju vsakega projekta oz. aplikacije posebej.

4.1. Osebni online katalog

Osebni online katalog je spletna aplikacija, ki ponuja katalog različnih zbirk. Ta aplikacija ponuja nekaj osnovnih katalogov - katalog filmov in glasbe. Poleg osnovnih katalogov pa je aplikacijo mogoče razširiti z dodajanjem novih katalogov. Za to je sicer potrebno nekaj programerskega znanja, vendar ne bi smelo biti pretežko - osnovna kataloga sta namreč narejena po enakem principu, kot naj bi bili tudi ostali in tako služita kot primer.
Ker je to osebni katalog, ki pa je dostopen preko spleta in tako viden vsem, seveda potrebuje vsaj osnovno avtorizacijo. Tako je npr. za dodajanje novih vnosov potrebno imeti ustrezne pravice, prav tako pa tudi za urejanje in brisanje teh vnosov. Posebno mesto ima seveda administrator, ki poleg ostalega lahko tudi briše posamezne uporabnike. Običajno bo administrator kar lastnik kataloga.
Za zagotavljanje avtorizacije seveda potrebujemo tudi avtentikacijo - ugotavljanje istovetnosti osebe. Tu aplikacija ponuja standardno avtentikacijo preko uporabniškega imena in gesla, poleg tega pa še avtentikacijo z uporabo OpenID, ki postaja vse bolj razširjen način avtentikacije na internetu (glej www.openid.net).
Vsak prijavljen uporabnik ima svoj profil, preko katerega lahko nastavlja določene lastnosti, npr. jezik aplikacije.

4.2. Uporabljena orodja

Pri implementiranju aplikacije sem uporabil tudi nekaj orodij, ki omogočajo lažjo uporabo določenih pristopov. V naslednjih podpoglavjih so ta orodja natančneje opisana, prav tako pa tudi njihova uporaba.

4.2.1. NHibernate

Eno izmed O/RM orodij je NHibernate in je namenjeno razvoju v ogrodju .NET. Ogrodje .NET sicer za delo s podatkovno bazo ponuja ADO.NET in DataSet, vendar ima ta precej pomankljivosti. Ena je predstavitev podatkov v tabelarični obliki, naša želja in cilj pa je seveda uporaba objektov. Z ustreznimi pristopi lahko sicer to težavo nekoliko omilimo (npr. uporaba tipiziranih datasetov), vseeno pa je velika verjetnost, da se bo vsaka sprememba strukture v podatkovni bazi močno odražala v sami kodi.

Drug problem dataseta je, da naslanjanje neposredno na dataset ruši transparentnost. V sami kodi je namreč razvidno, kakšen mehanizem za shranjevanje podatkov je uporabljen, to pa vpliva tudi na samo pisanje kode.
Rešitev teh problemov v ogrodju .NET je, kot že rečeno, uporaba O/RM orodja, v mojem primeru je to NHibernate. Za začetek uporabe tega orodja je potrebnih nekaj korakov:

1. Ustrezna nastavitev orodja.
2. Kreiranje domenskega modela in podatkovne baze.
3. Izdelava mapiranj med domenskimi objekti in podatkovno bazo.
4. Uporaba v kodi

Vsak od teh korakov je bolj podrobno opisan v nadaljevanju.

Nastavitve NHibernate orodja
Za nastavljanje NHibernate se običajno uporablja nastavitvena datoteka. Za delovanje je potrebno določiti vsaj katera podatkovna baza se uporablja (Oracle, MSSQL, MySQL itd.) ter povezavni niz za to bazo. Primer nastavitvene datoteke (glavne nastavitve so odebeljene):

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
  <session-factory>
    <property name="connection.provider">
      NHibernate.Connection.DriverConnectionProvider
    </property>
    <property name="dialect">NHibernate.Dialect.MsSql2000Dialect</property>
    <property name="connection.driver_class">
      NHibernate.Driver.SqlClientDriver
    </property>
    <property name="connection.connection_string">
      Data Source=.\SQLEXPRESS;Initial Catalog=DVDCatalog;Integrated Security=SSPI
    </property>
    <property name="connection.release_mode">on_close</property>
    <property name="show_sql">true</property>
    <property name='proxyfactory.factory_class'>
      NHibernate.ByteCode.Castle.ProxyFactoryFactory, NHibernate.ByteCode.Castle
    </property>
  </session-factory>
</hibernate-configuration>
```

Kreiranje domenskega in podatkovnega modela
Domenski in podatkovni model sta opisana v poglavju 4.3.1.
Osnova vsem entitetam je razred EntityBase.

```
public abstract class EntityBase {
  protected virtual int EntityID { get; set; }

  public override bool Equals(object obj) {
    var entity = obj as EntityBase;
    return (entity != null) && GetType().IsAssignableFrom(entity.GetType()) &&
      EntityID.Equals(entity.EntityID);
  }

  public override int GetHashCode() {
    return (GetType().Name + EntityID).GetHashCode();
  }
}
```

Iz te entitete pa dedujejo vse ostale entitete, npr. Role:

```
public class Role: EntityBase {
  public virtual int RoleID {
    get { return EntityID; }
    set { EntityID = value; }
  }

  public virtual string RoleName { get; set; }

  public virtual bool BuiltIn { get; set; }
}
```

Pomembna entiteta je še Content, ki je osnovna entiteta za vse ostale entitete, ki jih katalog zna prikazati (npr. Album, Movie).

```
public class Content: EntityBase {
  public virtual int ContentID {
    get { return EntityID; }
    set { EntityID = value; }
  }

  public virtual MediaType MediaType { get; set; }

  public virtual string Title { get; set; }

  public virtual DateTime ReleaseDate { get; set; }

  public virtual DateTime CreationDate { get; set; }

  public virtual Artist Artist { get; set; }

  public virtual IList<Category> Categories { get; set; }

  public virtual IList<Picture> Pictures { get; set; }
}
```

Definicije ostalih razredov so na voljo v priloženi izvorni kodi.

Izdelava mapiranj med domenskimi objekti in podatkovno bazo
Na podlagi mapiranj NHibernate zna napolniti domenske objekte s podatki iz podatkovne baze. Vsak domenski objekt oziroma razred v kodi zahteva svoje mapiranje. Mapiranje poveže razred s persistiranimi lastnostmi v bazi. Razred ima seveda lahko tudi lastnosti, ki se ne persistirajo, v tem primeru se v mapiranju preprosto ignorirajo. Primer mapiranja za razred Role:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
  <class name="AVCat.Core.Model.Role, AVCat.Core" table="Role">
    <property name="RoleID" column="RoleID" type="Int32" unsaved-value="0">
      <generator class="native" />
    </property>
    <property name="RoleName" column="RoleName" type="String(50)" />
    <property name="BuiltIn" column="BuiltIn" type="Boolean" />
  </class>
</hibernate-mapping>
```

To mapiranje orodju NHibernate pove, da se lastnosti razreda Role (v imenskem razredu AVCat.Core.Model) nahajajo v tabeli Role, ter da se tri lastnosti razreda mapirajo v ustrezne stolpce v tabeli. Za lastnost RoleID je določen tudi generator, ki pove, kako naj se primarni ključ oziroma identiteta razreda generirata. Native v tem primeru določa, da orodje samo izbere primeren generator, glede na izbrano bazo. Več o tem na [**6**].
Ker pa je delo z mapiranji zamudno in je velika verjetnost napačnega vnosa za kako mapiranje, saj ni preverjanja tipov, sem se sam odločil za drugačen pristop - uporabo FluentNHibernate. To je knjižnica, ki z uporabo učinkovitega vmesnika (glej poglavje 3.8) omogoča lažje nastavljanje mapiranj kar iz kode. Še enkrat primer za razred Role, tokrat z uporabo FluentNHibernate.

```
public sealed class RoleMap : ClassMap<Role> {
  public RoleMap() {
    Id(x => x.RoleID)
      .GeneratedBy.Native()
      .UnsavedValue(-1);
    Map(x => x.RoleName, "Name")
      .Not.Nullable()
      .Unique();
    Map(x => x.BuiltIn)
     .Not.Nullable()
     .ReadOnly();
  }
}
```

Kot je razvidno, je veliko manj možnosti za napako, saj se veliko stvari preveri že ob prevajanju kode. Prav tako prevajalnik opozori na napako, če se npr. spremeni razred Role oziroma taka uporaba omogoča uporabo naprednih orodij - npr. Refactor.
Primer je sicer zelo enostaven, pri kompleksnejših primerih pa je potrebno biti precej bolj pozoren. Tako je npr. treba biti pozoren pri mapiranju lastnosti, ki so zbirke oziroma seznami, lastnosti, ki kažejo na druge objekte ipd.
Drug primer pa je bolj kompleksen. Velikokrat imamo namreč v kodi dedovane razrede, v mojem primeru npr. Tako Video kot Music dedujeta iz Content. Imata namreč nekaj skupnih lastnosti, nekaj pa je tudi drugačnih. Tako se mapirata v različen *podrazred* glede na vrednost, ki tak podrazred določa. Tu je možnosti več:

- Ena tabela za celotno razredno hierarhijo (table per class hierarchy). Pri tem pristopu je celotna razredna hierarhija shranjena v eni sami tabeli, konkreten tip razreda pa je določen preko *discriminator* stolpca – vsak razred ima svojo vrednost.
- Ena tabela za vsak podrazred (*table per subclass*). Pri tem pristopu je za vsak razred ena tabela, ena tabela pa je za glavni razred. Odvisne tabele so na glavno povezane tako, da ima vsaka odvisna tabela za primarni ključ enako vrednost kot glavna tabela.
- Ena tabela za vsako implementacijo (*table per concrete class*). Pri tem pristopu ne rabimo tabele za glavni razred, je pa potrebno drugačno mapiranje glavnega razreda.

Z uporabo Fluent NHibernate se predvsem druga točka močno olajša, tako da je za mapiranje glavnega Content razreda potrebno sledeče:

```
public sealed class ContentMap : ClassMap<Content> {
  public ContentMap() {
    Id(x => x.ContentID, "ContentID")
      .GeneratedBy.Native()
      .UnsavedValue(0);
```

```
    Map(x => x.CreationDate)
      .Not.Nullable();
    References(x => x.MediaType, "MediaTypeID");
    Map(x => x.ReleaseDate)
      .Not.Nullable();
    Map(x => x.Title)
      .Not.Nullable();
    References(x => x.Artist, "ArtistID");
    HasManyToMany(x => x.Categories)
      .Table("Content_Category")
      .ParentKeyColumn("ContentID")
      .ChildKeyColumn("CategoryID")
      .LazyLoad();
    HasMany(x => x.Pictures)
      .Inverse()
      .KeyColumn("ContentID")
      .Cascade.AllDeleteOrphan()
      .LazyLoad();
  }
}
```

Dedovani razredi pa se mapirajo tako:

```
public sealed class AudioMap : SubclassMap<Album> {
  public AudioMap() {
    Table("Audio");
    KeyColumn("ContentID");
    HasMany(x => x.Tracks)
      .Inverse()
      .KeyColumn("ContentID")
      .Cascade.AllDeleteOrphan();
  }
}
```

In če potem iz kode želimo instanco razreda Content z identifikacijsko številko 1, dobimo nazaj objekt konkretnega tipa (npr. Audio), ki ima tako identifikacijsko številko.
V zadnjem primeru je prikazano tudi mapiranje referenciranih objektov (References) ter relacij (HasMany, HasManyToMany).
Več o NHibernate mapiranjih je na voljo na NHibernate spletni strani.

Uporaba v kodi
Tako kot pri vsakem delu s podatkovno bazo je tudi pri delu z NHibernate pomembno, da smo pozorni na transakcije – torej da se posamezne enote kode, ki predstavljajo zaključeno celoto, izvedejo v eni transakciji. Vzorec splošne uporabe je naslednji:

```
ISession session;
ITransaction transaction;

session = factory.OpenSession();
transaction = session.BeginTransaction();
try {
  // do database work
  transaction.Commit();
  session.Close();
}
catch (Exception ex) {
```

```
  transaction.Rollback();
  session.Close();
  throw;
}
```

Konkretna implementacija za pridobitev seznama vseh vlog pa bi bila taka:

```
public IList GetRoles() {
  IList roles = null;
  ISession session;
  ITransaction transaction;

  session = factory.OpenSession();
  transaction = session.BeginTransaction();
  try {
    roles = session.CreateCriteria(typeof(Role)).List();
    session.Close();
  }
  catch (Exception ex) {
    transaction.Rollback();
    session.Close();
    throw;
  }
  return roles;
}
```

Metoda CreateCriteria sprejme parameter tipa Type, ki NHibernate pove, da naj uporabi tabelo, mapirano na ta tip. Metoda List pa vrne nazaj vse zapise iz te tabele, seveda kot objekte.
Sam sem se odločil za nekoliko drugačen pristop k uporabi NHibernate in sicer z uporabo Linq. Linq to NHibernate zaenkrat sicer še ne podpira čisto vseh operacij, vendar za moje potrebe zadosti. To mi je precej poenostavilo razvoj, saj npr. zgorjno kodo lahko napišem v taki obliki:

```
public IQueryable<Role> GetRoles() {
  IQueryable<Role> roles = null;
  ISession session;
  ITransaction transaction;

  session = factory.OpenSession();
  transaction = session.BeginTransaction();
  try {
    roles = from role in session.Linq<Role> select role;
    session.Close();
  }
  catch (Exception ex) {
    transaction.Rollback();
    session.Close();
    throw;
  }
  return roles;
}
```

Razlika tu še ni očitna, vendar pa že to, da metoda vrača IQueryable, omogoča uporabo Linq nad takim objektom, recimo tako:

```
var role = (from r in GetRoles
            where r.Name == “Administrator”
            select r).Single();
```

Tak pristop ima dve prednosti:

- Vmesnik med podatkovnim in poslovnim nivojem je tanek, omogoča pa veliko prilagodljivost. Tako npr. ni potrebno imeti več različnih metod za pridobivanje podatkov, ampak je teoretično dovolj samo GetRoles, razen za bolj kompleksne primere.
- Uporaba Linq omogoča ti. *lazy load*, to je izvedba poizvedbe šele takrat, ko je to potrebno. Tako se v tem primeru na podatkovno bazo pošlje poizvedba samo za vlogo Administrator in ne za vse vloge.

Pri tem je potrebno paziti še na eno stvar – *lazy load* deluje samo v primeru, če je seja dostopna. Pozoren bralec bo tako lahko ugotovil, da se prejšnji primer v bistvu ne bi izvedel, ampak bi vrnil napako. Namreč ob koncu metode GetRoles se seja zapre in tako potem ni več dostopna v času klica metode Single, ki pa zares sproži poizvedbo. Te težave se rešuje z različnimi pristopi, sam pa sem izbral pristop enote dela (*unit of work*), ki je opisana v poglavju 3.10. Glede na to, da gre za spletno aplikacijo, sem za enoto dela vzel kar eno poizvedbo. Tako se seja kreira ob začetku, zaključi pa na koncu poizvedbe.

4.2.2. ASP.NET MVC

Kaj točno je MVC, je razloženo v poglavju 3.11. ASP.NET MVC pa je MVC implementacija v okolju ASP.NET. S tem torej dobimo porazdeljenost vlog (*separation of concerns*), SRP in DRY. Microsoft je pred ASP.NET MVC propagiral ASP.NET Webforms, ki pa ga sedaj potihem opušča. ASP.NET MVC se od ASP.NET Webforms razlikuje v več stvareh:

ASP.NET Webforms	ASP.NET MVC
• Avtomatsko upravljanje stanja • Hiter razvoj (RAD) • Podpora drag & drop • Skrivanje kompleksnosti (kontrole) • Enak razvojni model kot WinForms	• Boljše deljenje odgovornosti • Polna kontrola nad generiranim HTML • Lažje testiranje (TDD). • Polna razširljivost • Upošteva naravno spleta »brez stanja« (stateless)

Tabela 2: Razlike med WebForms in MVC

Tako MVC kot Webforms pa od ASP.NET dobita keširanje, usmerjanje (routing), lokalizacijo, avtentikacijo, profile ipd.
ASP.NET MVC ponuja različne funkcionalnosti, ki dodatno pomagajo pri razvoju:

- akcijski filtri
- različni formati izhodov
- prilagajanje prikazovanja
- ModelBinder
- Druge funkcionalnosti, npr. validacija modelov, tovarna nadzornikov (*ControllerFactory*), ...

Vsaka funkcionalnost je podrobneje razložena v naslednjih poglavjih.

Akcijski filtri

Akcijski filter je atribut, ki se ga lahko uporabi na metodi nadzornika ali pa kar na celotnem nadzorniku. Filter vpliva na potek izvedbe metode oziroma akcije. Nekaj akcijskih filtrov, vključenih v ASP.NET MVC:

- OutputCache: filter kešira izhod akcije za določen čas.
- HandleError: filter obravnava napake, ki se zgodijo med izvajanjem akcije
- Authorize: filter omogoča omejitev dostopa do akcije samo določeni vlogi ali uporabniku.
- AcceptVerbs: filter omogoča, da se določena akcija kliče samo z določenimi HTTP metodami (GET, POST, DELETE ipd).
- ValidateInput: filter omogoča validacijo vhodnih podatkov za akcijo.
- ValidateAntiForgeryToken: filter ki v navezi z metodo Html.AntiForgeryToken() omogoča preverjanje za cross-site request napade.

Po potrebi pa lahko tak akcijski atribut implementiramo tudi sami.

Različni formati izhodov

ASP.NET MVC omogoča, da določena metoda (lahko tudi vse) vrača drugačen format od standardnega ViewResult, ki prikaže zahtevani pogled. Nekaj drugih možnih izhodov, ki jih ponuja:

- JsonResult: rezultat, ki se uporablja za pošiljanje JSON formata kot odgovor.
- FileResult: rezultat, ki se uporabi za pošiljanje binarnih datotek.
- RedirectResult: nazdira procesiranje akcije tako, da preusmerja na drug naslov.
- EmptyResult: rezultat, ki ne naredi ničesar, podobno kot akcija, ki ne vrne ničesar.
- ContentResult: uporabniško definiran rezultat.
- JavaScriptResult: se uporablja za pošiljanje javascript vsebine.

Obstaja še nekaj drugih formatov izhodov, lahko pa implementiramo tudi svoje.

Prilagajanje prikazovanja

ASP.NET MVC vsebuje privzet motor prikazovanja (ViewEngine) ter privzeto logiko za lociranje pogledov (ViewLocator), v primeru drugačnih potreb pa je zelo enostavno implementirati svoj motor oziroma lokator pogledov. Tako lahko preko svoje implementacije lokatorja določimo, kje se iščejo pogledi in druge kontrole, kakšen pogled se uporabi ipd.

ModelBinder

ModelBinder omogoča, da neka akcijska metoda sprejme kompleksne parametre in ne samo parametre enostavnih tipov, kot so string in integer. Celoten sistem deluje tako, da se za nek kompleksen tip izdela ModelBinder, na akcijski metodi pa se preko atributa pove, da naj se ta ModelBinder uporabi. Primer kompleksnega tipa in ModelBinder-ja zanj:

```
public class MyType {
  public string Name { get; set; }
}
public class MyModelBinder: IModelBinder {
  public object GetValue(ControllerContext context, string modelName,
    ModelStateDictionary modelState) {
    var instance = new MyType();
    instance.Name = context.HttpContext.Request[“Name”];
    return instance;
  }
}
```

Preproste kompleksne razrede sicer privzeti ModelBinder sam pravilno zgradi, če pa je kompleksen razred npr. dedovan ali pa vsebuje več drugih kompleksnih razredov, potem privzeti ModelBinder ne deluje več.
ModelBinder se lahko s tipom asociira na sami metodi, ali pa na tipu, ki ga predstavlja.

```
[ModelBinder(typeof(MyModelBinder))]
public class MyType ...

public ActionResult Test([ModelBinder(typeof(MyModelBinder))] MyType myType) ...
```

Obstaja pa še tretja možnost. Če npr. ni možnosti na razred dodati dodatnega atributa, vseeno pa želimo, da določen ModelBinder velja na vseh metodah, lahko uporabimo tudi sledeč pristop:

```
protected void Application_Start() {
  RegisterRoutes(RouteTable.Routes);
  // Register custom model binders.
  ModelBinders.Binders[typeof(MyType)] = new ContentModelBinder(
    new MyModelBinder());
}
```

Torej binder lahko registriramo tudi pri zagonu aplikacije, kjer dodamo mapiranje med tipom in binderjem v statično zbirko.

Poleg omenjenih funkcionalnosti je prednost ASP.NET MVC tudi v tem, da se tudi uporabniški vmesnik dokaj enostavno testira, kar npr. za ASP.NET WebForms ne velja. Testiranje uporabniškega vmesnika je v splošnem problematično, tako da je to še en velik plus za MVC tehnologijo.

4.2.3. Enota dela

Kaj točno enota dela je in kaj je njen namen, je opisano v poglavju 3.10. Tukaj pa bi rad predstavil svojo implementacijo enote dela. Glavni razlog, zakaj sem enoto dela implementiral sam je bil, ker sem hotel specifiko uporabljenega orodja za persistiranje (NHibernate) ločiti od ostale aplikacije. To omogoča lažje testiranje ter tudi zamenjavo orodja za persistiranje, če bi se to izkazalo za potrebno.
Konkretno implementacijo sem povzel po enoti dela, kot jo je implementiral Oren Eini v svoji knjižnici *Rhino.Commons*.

```
public interface IUnitOfWork: IDisposable {
  bool IsInActiveTransaction { get; }
  IGeneralTransaction BeginTransaction();
  IGeneralTransaction BeginTransaction(IsolationLevel isolationLevel);

  void Flush();
  void TransactionalFlush();
  void TransactionalFlush(IsolationLevel isolationLevel);
  void InitializeCulture(CultureInfo cultureInfo);

}
```

Kot je razvidno enota dela skrbi za upravljanje s transakcijami, poleg tega pa moja enota dela skrbi še za izvajanje enote v pravilnem jeziku – namreč določeni podatki v podatkovni bazi so lokalizirani in enota dela mora vedeti, v katerem jeziku podatke prebrati iz baze.

Konkretna implementacija enote dela se pridobi s pomočjo vzorca tovarne, izdelan pa je tudi pomožni razred, preko katerega se potem enota dela lahko uporabi v kodi.

```
public static class UnitOfWork {

  private const string CurrentUnitOfWorkKey = "CurrentUnitOfWork.Key";

  public static bool IsStarted {
    get {
      return (Local.Data[CurrentUnitOfWorkKey] != null);
    }
  }

  public static IUnitOfWork Current {
    get {
      if (!IsStarted)
        throw new InvalidOperationException("Unit of work not started.");
      return (IUnitOfWork)Local.Data[CurrentUnitOfWorkKey];
    }
    private set {
      Local.Data[CurrentUnitOfWorkKey] = value;
    }
  }

  public static object CurrentSession {
    get {
      return ObjectFactory.GetInstance<IUnitOfWorkFactory>().CurrentSession;
    }
  }

  public static void DisposeUnitOfWork(IUnitOfWorkImpl unitOfWork) {
    if (unitOfWork == null)
      throw new ArgumentNullException("unitOfWork");
    Current = unitOfWork.Previous;
  }

  public static IUnitOfWork Start() {
    return Start(null, true /* returnExisting */);
  }

  public static IUnitOfWork Start(IDbConnection connection, bool returnExisting) {
    IUnitOfWorkImpl existingUnitOfWork =
      (IUnitOfWorkImpl)Local.Data[CurrentUnitOfWorkKey];
    if ((existingUnitOfWork != null) && returnExisting) {
      existingUnitOfWork.IncrementUsages();
      return existingUnitOfWork;
    }
    Current = ObjectFactory.GetInstance<IUnitOfWorkFactory>().Create(connection,
      existingUnitOfWork);
    return Current;
  }

}
```

Uporaba enote dela pa je sledeča:

```
UnitOfWork.Start();
try {
  using (var transaction = UnitOfWork.Current.BeginTransaction()) {
    // database operations
    // ...
    transaction.Commit();
  }
} finally {
  UnitOfWork.Current.Dispose();
}
```

Klic metode UnitOfWork.Start kreira novo enoto dela, s tem da če že obstaja kakšna enota dela, se samo poveča število uporabnikov enote preko klica metode IncrementUsages, v nasprotnem primeru pa se kreira nova enota preko tovarne. Tu je potrebno upoštevati da se tudi tovarna, ki je odgovorna za enote dela, pridobi iz tovarne, in sicer ObjectFactory. To je tovarna orodja StructureMap, ki jo uporabljam v celotni aplikaciji. StructureMap orodje je orodje za vstavljanje odvisnosti, več o tem je na voljo v poglavju 3.6.
V aplikaciji se sicer klic UnitOfWork.Start in UnitOfWork.Current.Dispose ne pojavljata velikokrat. Ker je to spletna aplikacija in je enota dela izenačena z eno zahtevo, se enota dela kreira takoj na začetku zahteve, na koncu zahteve pa se uniči.

```
public class MvcApplication : System.Web.HttpApplication {

  private static void Application_BeginRequest(object sender, EventArgs e) {
    UnitOfWork.Start();
  }

  private static void Application_EndRequest(object sender, EventArgs e) {
    if (UnitOfWork.IsStarted) {
      UnitOfWork.Current.Flush();
      UnitOfWork.Current.Dispose();
    }
  }

}
```

Če je potrebno v transakciji izvesti več poizvedb, je pa to seveda potrebno narediti eksplicitno.

4.2.4. StructureMap

StructureMap je orodje za obračanje nadzora in vstavljanje odvisnosti. Ob pametni uporabi lahko precej poveča možnosti ponovne uporabe kode (*code reuse*), saj omogoča minimiziranje direktnih povezav med posameznimi razredi.
Pri uporabi orodja oziroma nasploh pri uporabi principa za obračanje nadzora se je potrebno zavedati, da prevelika količina abstrakcije v majhnih projektih ni preveč zaželjena in je lahko celo škodljiva. Do izraza pride pri večjih, kompleksnejših projektih.

Nastavitve StructureMap orodja

Za pravilno delovanje orodja je pomembno predvsem, da so pravilno nastavljene preslikave med vmesniki in implementacijami le-teh. Kako oziroma kje pa se te preslikave nastavijo, je odvisno od potreb aplikacije. Orodje ponuja kot možnost nastavitve v .config nastavitveni

datoteki ali pa kar v kodi. Posledično lahko kak drugačen tip nastavitve sprogramiramo tudi sami.
Primer nastavitvene datoteke:

```
<xml version="1.0" charset="utf-8">
<StructureMap MementoStyle="Attribute">
  <DefaultInstance
    PluginType="MyAssembly.IColorPicker, MyAssembly"
    PluggedType="MyAssembly.ColorPicker, MyAssembly" />
</StructureMap>
```

PluginType predstavlja vmesnik, za katerega želimo ustvariti preslikavo, PluggedType pa predstavlja konkretno implementacijo vmesnika. Za nek vmesnik lahko definiramo tudi več preslikav, le da namesto *DefaultInstance*, ki predstavlja privzeto instanco, uporabimo *AddInstance* in ji določimo še ključ. V kodi potem lahko pri zahtevi uporabimo tudi ta ključ, da dobimo ustrezno implementacijo vmesnika.

```
<xml version="1.0" charset="utf-8">
<StructureMap MementoStyle="Attribute">
  <AddInstance
    Key="solid"
    PluginType="MyAssembly.IColorPicker, MyAssembly"
    PluggedType="MyAssembly.SolidColorPicker, MyAssembly" />

  <AddInstance
    Key="system"
    PluginType="MyAssembly.IColorPicker, MyAssembly"
    PluggedType="MyAssembly.SystemColorPicker, MyAssembly" />
</StructureMap>
```

Drug način pa je nastavitev v kodi.

```
ObjectFactory.Configure(x => {
  x.ForRequestedType<IColorPicker>().TheDefaultIsConcreteType<ColorPicker>();
});
```

Če pa imamo več logično ločenih nastavitev, pa si lahko pomagamo z razredom *Registry*.

```
public class ColorPickerRegistry: StructureMap.Configuration.DSL.Registry {

  protected override void configure() {
    ForRequestedType<IColorPicker>().TheDefaultIsConcreteType<ColorPicker>();
  }

}

ObjectFactory.Configure(x => {
  x.AddRegistry(new ColorPickerRegistry());
});
```

Metoda ObjectFactory.Configure ponuja tudi bolj napredne načine nastavitev, npr. nastavitev tako iz nastavitvene datoteke kot iz registrov, pri čemer lahko tudi avtomatsko najde ustrezne registre. Več o tem je na voljo na spletni strani structuremap.sourceforge.net/ConfiguringStructureMap.htm.

Uporaba v kodi

Uporaba StructureMap v kodi je precej preprosta. Vse kar želimo je, da nam tovarna za podan vmesnik vrne ustrezno implementacijo le-tega.

```
var colorPicker = ObjectFactory.GetInstance<IColorPicker>();
```

Če tovarna ni ustrezno nastavljena, dobimo izjemo, drugače pa neko konkretno implementacijo vmesnika, odvisno od nastavitev.
Prav tako lahko dobimo tudi ustrezno instanco tudi z uporabo ključa:

```
var colorPicker = ObjectFactory.GetNamedInstance<IMyInterface>("solid");
```

4.3. Implementacija

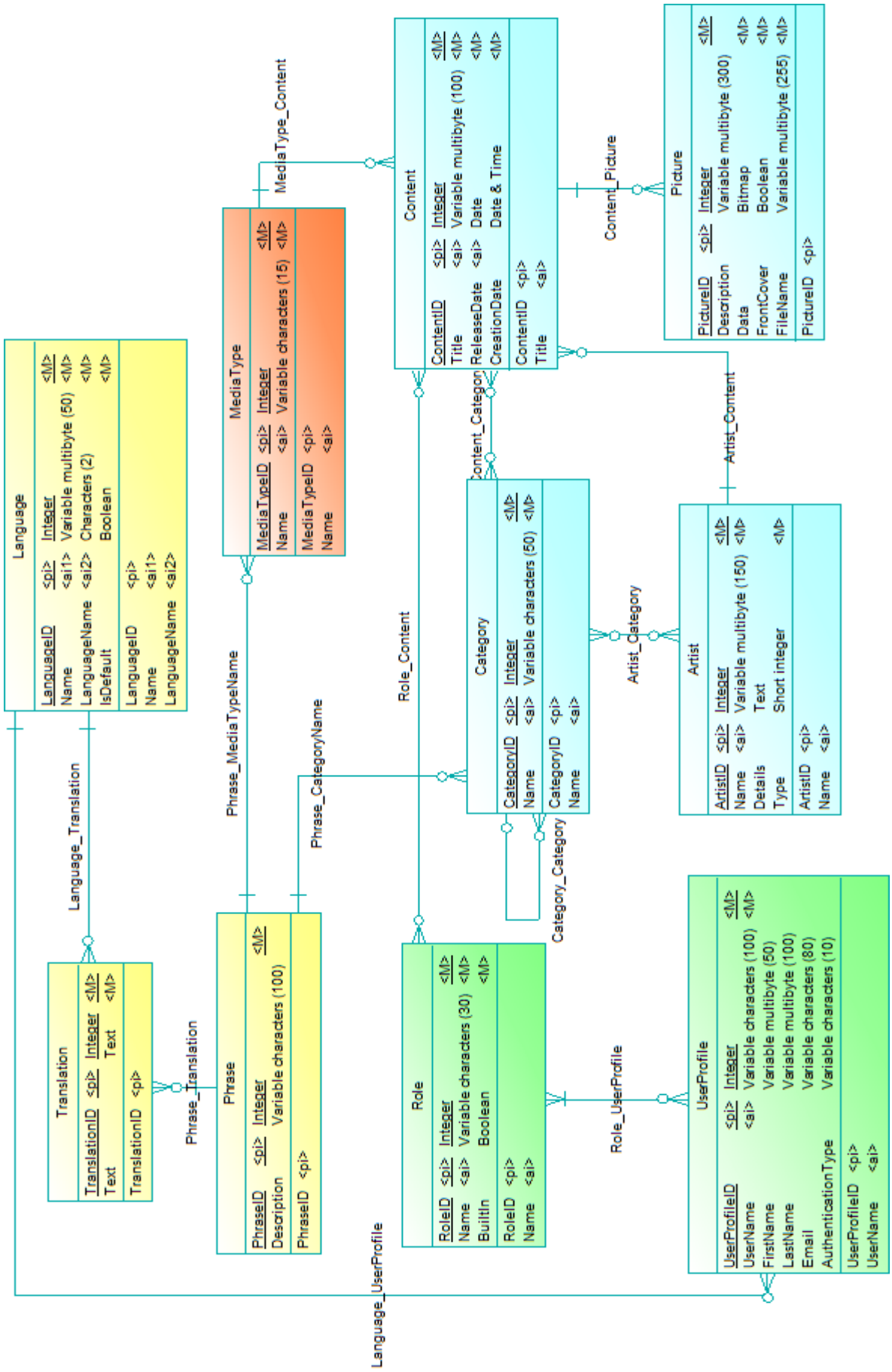
Določeni primeri implementacije vzorcev in uporabe orodij so opisani v prejšnjem poglavju, tukaj pa bom v grobem predstavil aplikacijo in njeno arhitekturo. Odločil sem se za klasično tri-plastno arhitekturo, ki omogoča učinkovito upravljanje z odvisnostmi med deli sistema. Če pa bi govorili o nivojih, je to klasična dvo-nivojska (client-server) aplikacija, kjer je en nivo sestavljen iz predstavitvene in poslovne logike, strežnik pa je podatkovna baza. To je poenostavilo celotno rešitve do te mere, da sem se v veliki meri lahko izognil uporabi DTO objektov ter poenostavil servisne metode. Z nekaj dela pa bi aplikacijo seveda lahko implementiral tudi kot tri-nivojsko.
V naslednjih poglavjih bom predstavil rešitve za vsako plast posebej, posebno poglavje pa je namenjeno domeni aplikacije.

4.3.1. Domenski model

Kot je bilo omenjeno že v prejšnjih poglavjih, je aplikacija namenjena online katalogu različnih tipov zbirk. Vsak tip zbirke je označen s kategorijo (npr. Audio, Video, Books, ...), vsaka kategorija pa ima lahko tudi eno ali več podkategorij (npr. zvrst glasbe). Vsak vnos lahko pripada eni ali več kategorijam (npr. film je lahko po eni strani akcijski, po drugi pa zgodovinski), vnos pa ima lahko tudi več slik, od teh je ena slika lahko naslovna. Vsak vnos se lahko pri uporabniku nahaja na različnih medijih, npr CD, DVD, knjiga ipd.
V samo aplikacijo je dodana tudi podpora večjezičnosti, tako da so določeni deli aplikacije lokalizirani. Uporabniški vmesnik ima svoj sistem lokalizacije, potrebno pa je lokalizirati tudi šifrante – v tem primeru so to kategorije in tip medija oziroma nosilca. Lokalizacija je sestavljena iz treh komponent:

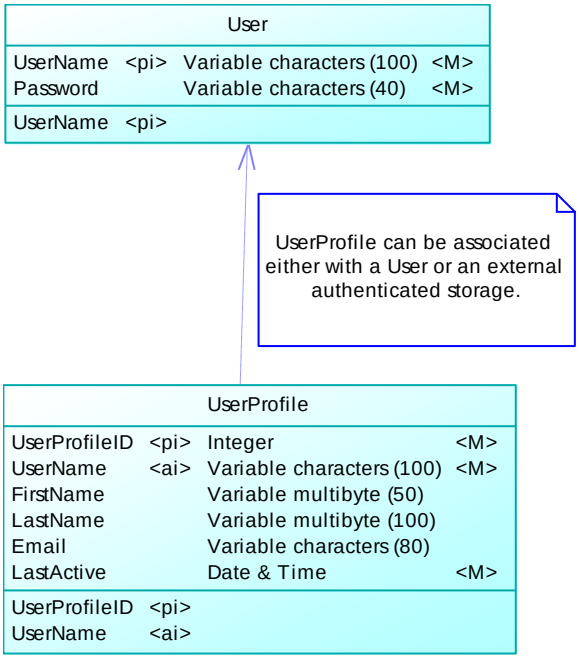
- fraza – označuje posamezno frazo, ki je lahko prevedena v več jezikov,
- jezik – jezik, ki je uporabljen za prevod neke fraze,
- prevod – prevod neke fraze v določenem jeziku.

Za vsak vnos v šifrantu je seveda potrebno določiti nek ključ, preko katerega se lahko sklicujemo nanj, večinoma je to univerzalno ime tistega vnosa (CD, DVD), na uporabniškem vmesniku pa je prikazan prevod tega vnosa (npr. zgoščenka za CD). Včasih je nek izraz univerzalen, v tem primeru je dovolj, da je preveden samo v privzeti jezik.
Lokalizacija pride v veljavo samo, če si uporabnik lahko spremeni prikazani jezik.To lahko stori preko svojega profila, na katerega so vezane tudi različne pravice. Tako lahko npr. katalog pregledujejo vsi, urejanje kataloga pa je dovoljeno samo z ustrezno pravico.



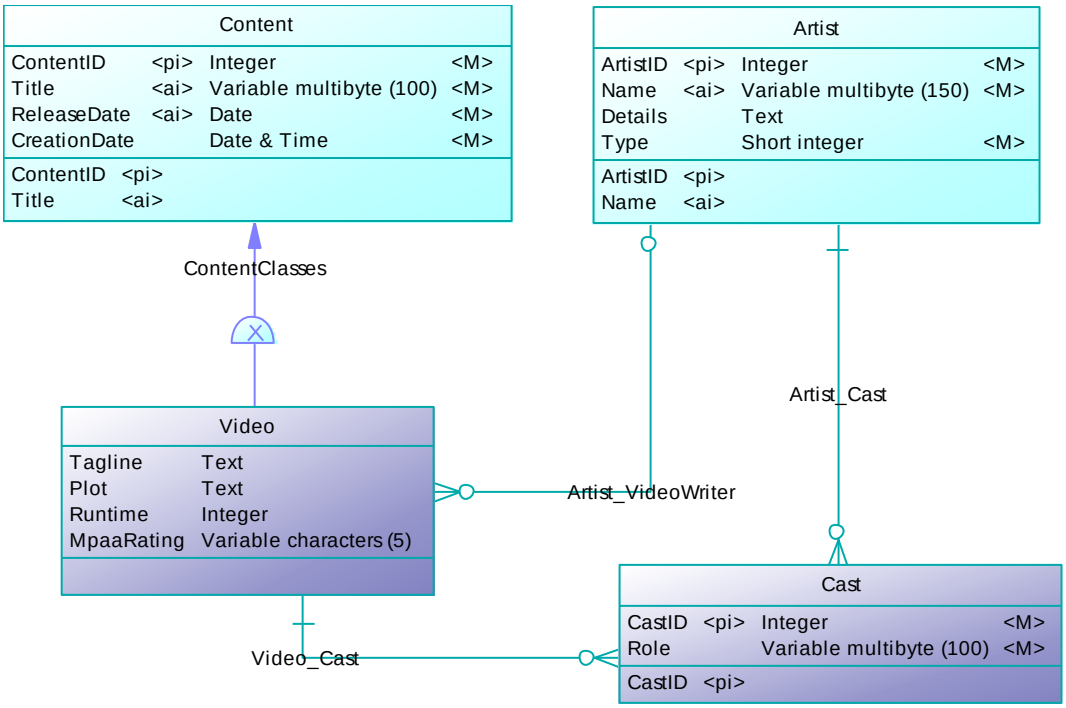
Slika 4: Konceptualni model

Slika prikazuje konceptualni model, ki je zelo podoben domenskemu. V mojem primeru se domenski model skoraj direktno preslika na konceptualnega, tako da je dovolj samo eden. Uporabljen pa je konceptualni model, ker je le-tega brez večjih težav mogoče preslikati v fizični model za konkretno podatkovno bazo.
Da uporabnika sistem prepozna, se mora le-ta najprej prijaviti. Sistem podpira različne vrste prijave, en izmed njih pa je tudi prijava preko uporabniškega imena in gesla. V ta namen je narejena tabela User, ki se potem veže na uporabniški profil.



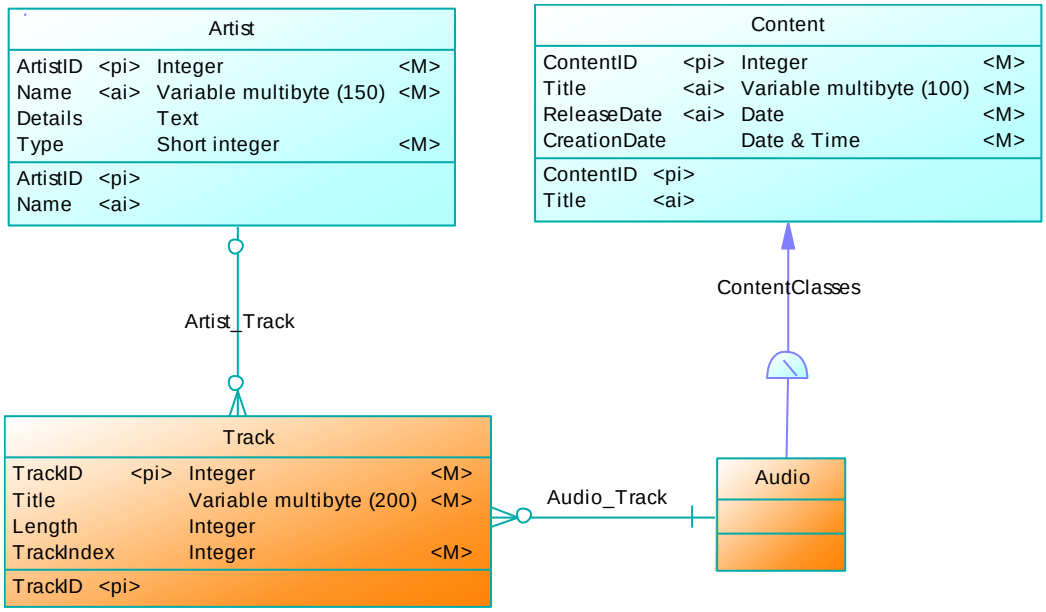
Slika 5: Avtentikacija preko imena in gesla

Aplikacija ponuja dva vgrajena modula, enega za filme in drugega za glasbo. Oba modula se navezujeta na glavni sistem, saj se mora vsak modul navezovati vsaj na Content tabelo, po potrebi pa se lahko seveda tudi na druge.
Modul za filme vsebuje dve tabeli - glavna tabela predstavlja zapis za en film ter osnovne podatke o njem, vsak film pa vsebuje tudi več igralcev, vsak igralec ima seveda v filmu določeno vlogo.



Slika 6: Modul "Filmi"

Modul za glasbo pa prav tako vsebuje dve tabeli, kjer glavna tabela predstavlja zapis za določen album, vsak album pa vsebuje tudi eni ali več skladb. Poleg tega ima vsaka skladba lahko tudi določenega izvajalca. Če skladba nima določenega izvajalca, se smatra, da je to izvajalec albuma.



Slika 7: Modul "Glasba"

Na podoben način se lahko doda tudi drugačne module.

4.3.2. Podatkovna plast

Kot že omenjeno v poglavju 2.2.3, je podatkovna plast namenjena temu, da poslovna plast nemoteno in predvsem transparentno lahko dostopa do objektov, shranjenih v podatkovni bazi. V poglavju 4.2.1 sem opisal, da sem se, da bi si olajšal in predvsem pospešil razvoj

logike za dostop do podatkovne baze, odločil za orodje NHibernate. Da pa bi izbira tega orodja bila čim bolj transparentna navzven, sem uporabo orodja skril z uporabo dveh vzorcev – enota dela in shramba. Konkretna implementacija shrambe izgleda tako, da se najprej definira nek splošen vmesnik zanjo.

```
public interface IRepository<T> {
  long Count();
  T Get(object id);
  IQueryable<T> GetAll();
  T Load(object id);

  T Save(T entity);
  T SaveOrUpdate(T entity);
  void Update(T entity);

  void Delete(T entity);
  void DeleteAll();
}
```

Kot je razvidno iz vmesnika, shramba omogoča pridobivanje števila zapisov za nek razred, pridobivanje instance razreda z neko identifikacijo, pridobivanje vseh instanc, shranjevanje in posodabljanje instanc ter brisanje vseh instanc iz baze. Tu velja opozoriti na metodo za pridobivanje vseh instanc iz baze, GetAll. Metoda vrača IQueryable, kar omogoča uporabo linq. Če konkretna implementacija shrambe podpira linq, se tako precej poenostavi delo s podatki. Npr. za pridobitev vseh vgrajenih vlog je potrebno sledeče:

```
IRepository<Role> roleRepository = ...
var builtInRoles = from r in roleRepository.GetAll()
                   where r.BuiltIn
                   select r;
```

Tako se lahko elegantno izognemo uporabi filtrov in kriterijev, ki bi bili mogoče specifični za posamezno uporabljeno orodje.
Konkretna instanca shrambe za nek tip pa se dobi preko statičnega Repository razreda, ki za pridobitev instance uporablja orodje StructureMap (glej 4.2.4).

```
public static class Repository<T> {

  private static IRepository<T> InternalRepository {
    get { return ObjectFactory.GetInstance<IRepository<T>>(); }
  }

  public static long Count() {
    return InternalRepository.Count();
  }

  public static T Get(object id) {
    return InternalRepository.Get(id);
  }

  ...

}
```

Tako bi zgornji primer za pridobivanje vseh vgrajenih vlog izgledal tako:

```
var builtInRoles = from r in Repository<Role>.GetAll()
                   where r.BuiltIn
                   select r;
```

Seveda je potrebno shrambo tudi implementirati. Za uporabo NHibernate izgleda shramba tako:

```
public class NHRepository<T>: IRepository<T> {

  protected ISession Session {
    return (ISession)UnitOfWork.CurrentSession;
  }

  public virtual long Count() {
    var criteria = DetachedCriteria.For<T>().GetExecutableCriteria(Session);
    criteria.SetProjection(Projections.RowCountInt64());
    return Convert.ToInt64(criteria.UniqueResult());
  }

  public virtual T Get(object id) {
    return Session.Get<T>(id);
  }

  public virtual IQueryable<T> GetAll() {
    // Linq2NHibernate
    return Session.Linq<T>();
  }

  ...

}
```

Ker je uporaba konkretnega orodja skrita za enoto dela in shrambo, se po potrebi lahko O/RM orodje kadarkoli zamenja oziroma se uporabi kak drug pristop za pridobivanje objektov iz podatkovne baze. Vedno bo potrebno samo napisati konkretno implementacijo shrambe in enote dela, implementacije registrirati v tovarni za vstavljanje odvistnosti (v mojem primeru StructureMap), spremembe na višjih plasteh pa niso potrebne. Po drugi strani pa velja omeniti, da se potreba po zamenjavi O/RM orodja pojavi zelo redko, zato si večinoma to delo lahko prihranimo. V tem primeru lahko vlogo nekakšne shrambe igra kar *Session* objekt iz NHibernate.

4.3.3. Poslovna plast

Poslovna plast vsebuje celotno poslovno logiko aplikacije, do podatkov pa dostopa preko podatkovne plasti.
V konkretnem primeru je poslovna plast implementirana kot storitev, definirana preko vmesnika, konkretna implementacija pa je odvisna od potreb. Večinoma so storitve implementirane z uporabo shrambe. Taka uporaba omogoča, tako kot pri podatkovni plasti, enostavno nadomestitev storitev z drugimi, ki mogoče ne uporabljajo shrambe.
Storitev ustrezno shrambo dobi preko konstruktorja in se seveda ne zaveda konkretne implementacije shrambe. Konkretno implementacijo storitve pa dobimo preko tovarne za vstavljanje odvisnosti, le-ta pa poskrbi tudi za to, da storitev dobi ustrezne shrambe.

Pomembnejše storitve

V tem poglavju so naštete storitve, ki so pomembne za samo delovanje sistema.

AuthenticationService
Storitev je namenjena avtentikaciji uporabnikov z uporabo uporabniškega imena in gesla. Omogoča registracijo novega uporabnika, prijavo obstoječega uporabnika in zamenjavo gesla.

```
public interface IAuthenticationService {
  bool Register(string userName, string password, string email);
  bool Validate(string userName, string password);
  bool ChangePassword(string userName, string currentPassword,
    string newPassword);
}
```

Konkretne implementacije storitve so lahko različne, glede na to, kje so shranjeni podatki o upotabniških računih. Aplikacija privzeto podpira upotabniške račune, shranjene v podatkovni bazi. Avtentikacija OpenID, omenjena v poglavju 4.1, pa se odvija samo na nivoju uporabniškega vmesnika in storitev zanjo ni potrebna.
Uporabnik se preko uporabniškega imena in gesla prijavi v sistem, ime in geslo pa sta shranjena v podatkovni bazi. Tako ta storitev za delovanje uporablja shrambo, preko katere pridobi potrebne podatke za preverjanje uporabnika. Preverjanje uporabnika v tej storitvi izgleda tako:

```
var user = GetUser(userName, password);
return (user != null);
```

Metoda GetUser pa preko shrambe vrne uporabnika, ki ustreza podanemu imenu in geslu.

```
private User GetUser(string userName, string password) {
  var user = (from u in _userRepository.GetAll()
              where u.UserName == userName)
              select u).FirstOrDefault();
  if ((user != null) && (string.Compare(user.Password, password,
    StringComparison.OrdinalIgnoreCase) == 0))
    return user;
  return null;
}
```

RoleService
Ko se uporabnik enkrat prijavi v sistem, pridobi določene pravice. Tako so npr. za urejanje podatkov potrebne večje pravice kot npr. za pregledovanje le-teh. Pravice se urejajo s pomočjo storitve za urejanje pravic.

```
public interface IRoleService {
  bool IsUserInRole(string userName, string roleName);
  Role[] GetRolesForUser(string userName);
  void CreateRole(string roleName);
  bool DeleteRole(string roleName);
  bool RoleExists(string roleName);
  void AddUsersToRoles(string[] userNames, string[] roleNames);
  void RemoveUsersFromRoles(string[] userNames, string[] roleNames);
  string[] GetUsersInRole(string roleName);
  Role GetRole(string roleName);
  IEnumerable<Role> GetAllRoles();
```

```
  string[] FindUsersInRole(string roleName, string usernameToMatch);
  void SaveRole(Role role);
}
```

Vmesnik je delno povzet po RoleProvider [**14**], ker je le-ta precej splošno uporaben, kot drugo pa se ustrezna implementacija RoleProvider razreda, ki uporablja storitev IRoleService, uporablja tudi v aplikaciji.
Konkretna implementacija zopet sloni na uporabi shrambe.

```
public class RoleService: IRoleService {

  ...

  public bool IsUserInRole(string userName, string roleName) {
    var roles = (from r in _roleRepository.GetAll()
                 join map in _roleRepository.GetUserRoleMap()
                   on r.RoleID equals map.RoleID
                 join u in _profileRepository.GetAll()
                   on map.UserProfileID equals u.UserProfileID
                 where u.UserName == userName && r.RoleName == roleName
                 select r);
    return (roles.Count() > 0);
  }

  ...

}
```

Kot je razvidno iz primera kode, so vloge vezane na uporabniški profil. Zakaj je temu tako, je razloženo v naslednjem poglavju.

PersonalizationService
Ne glede na to, kakšno metodo avtentikacije izbere uporabnik, se zanj ob prvi prijavi vedno kreira uporabniški profil. Profil je skupna točka, kjer si potem uporabnik lahko naknadno uredi določene nastavitve tako, kot mu najbolj ustrezajo. Ker je profil pod nadzorom aplikacije, avtentikacijske storitve pa ne nujno, je profil tista točka, na katero se vežejo tudi uporabniške pravice. Če ima torej uporabnik pravico urejati podatke, je to razvidno iz njegovega profila.
Omejitev profila je ta, da en profil ustreza natanko eni metodi avtentikacije. Torej en uporabnik ne more uporabljati istega profila in različne načine avtentikacije, npr. enkrat preko uporabniškega imena in gesla, drugič pa preko OpenID. Če en profil že ima in nato uporabi drug način avtentikacije, se avtomatično ustvari nov profil. Tukaj je vsekakor še prostor za izboljšavo, to je dodati podporo, da se lahko en profil uporabi preko različnih načinov avtentikacije.
Storitev za podporo profilom oziroma personalizaciji je precej preprosta.

```
public interface IPersonalizationService {
  IEnumerable<UserProfile> GetList();
  UserProfile Get(string userName);
  void Delete(string userName);
  void Save(UserProfile profile);
}
```

Storitev torej vsebuje osnovne metode za branje, brisanje in shranjevanje uporabniškega profila. Konkretna implementacija s shrambo je prav tako preprosta.

```
public class PersonalizationService: IPersonalizationService {

  ...

  public IEnumerable<UserProfile> GetList() {
    return _profileRepository.GetAll();
  }

  public void Delete(string userName) {
    _profileRepository.Delete(userName);
  }

  public UserProfile Get(string userName) {
    return _profileRepository.Get(userName);
  }

  public void Save(UserProfile profile) {
    if (profile.Language == null) {
      if (profile.UserProfileID > 0)
        throw new InvalidOperationException("Language for the profile is not
          defined.");
      profile.Language = _languageService.GetDefaultLanguage();
    }
    _profileRepository.SaveOrUpdate(profile);
  }

}
```

TranslationService in *LanguageService*

Uporabnik si v nastavitvah za profil lahko izbere tudi jezik, v katerem bo prikazan uporabniški vmesnik in določeni šifranti (npr. imena kategorij). Za ustrezno delovanje tega sta potrebni dve storitvi, ena ki ponuja upravljanje z mogočimi jeziki, druga pa s prevodi.

```
public interface ILanguageService {
  IEnumerable<Language> GetList();
  Language Get(int languageID);
  Language Get(string languageName);
  Language GetCurrent();
  void SetDefaultLanguage(int languageID);
}

public interface ITranslationService {
  IEnumerable<Translation> GetList();
  IEnumerable<Translation> GetListWithDefault(string languageName);
  int AddDefaultTranslation(string text);
  void SaveOrUpdate(Translation translation);
}
```

Storitev za upravljanje z jeziki je precej preprosta, vsebuje pa tudi dve bolj specifični metodi. GetCurrent vrača jezik, ki ustreza trenutno aktivno kulturi (glej System.Globalization.CultureInfo.CurrentCulture) oziroma če tak jezik ne obstaja, vrne privzeti jezik. Druga metoda, SetDefaultLanguage, pa nastavlja privzeti jezik za aplikacijo. Privzeti jezik je uporabljen, če uporabnik ne nastavi svojga jezika oziroma če iz kakršnegakoli

razloga izbrani jezik ni na voljo. V enem trenutku je samo en jezik lahko nastavljen kot privzet.

```
public Language GetCurrent() {
  var languages = (from l in GetList()
                   where l.IsDefault ||
                     (l.LanguageName ==
                       CultureInfo.CurrentCulture.TwoLetterISOLanguageName
                     )
                   select l).ToList();
  var currentLanguage = languages.Find(l => l.LanguageName ==
    CultureInfo.CurrentCulture.TwoLetterISOLanguageName);
  if (currentLanguage == null)
    currentLanguage = languages.Find(l => l.IsDefault);
  return currentLanguage;
}

public void SetDefaultLanguage(int languageID) {
  var languages = GetList();
  languages.ForEach(language => language.IsDefault = false);
  var defaultLanguage = languages.Find(language => language.LanguageID ==
    languageID);
  if (defaultLanguage == null)
    throw new ArgumentException("Cannot find the language with id " + languageID);
  defaultLanguage.IsDefault = true;
  using (var transaction = UnitOfWork.Current.BeginTransaction()) {
    languages.ForEach(language => _languageRepository.Update(language));
    transaction.Commit();
  }
}
```

Storitev za prevode pa vsebuje metode za dodajanje prevoda v privzetem jeziku, posodabljanje nekega prevoda ter pridobivanje vseh prevodov. Nekoliko posebna je metoda GetListWithDefault, ki vrne vse prevode za nek jezik kot tudi prevode v privzetem jeziku.

```
public IEnumerable<Translation> GetListWithDefault(string languageName) {
  return (from t in _translationRepository.GetAll()
          where t.Language.LanguageName == languageName ||
            t.Language.IsDefault
          orderby t.Phrase.PhraseID, t.Language.IsDefault
          select t);
}
```

EntityService

Ta storitev vsebuje vse metode za ostale entitete v aplikaciji - umetnike, kategorije, slike itd. Sama storitev je zopet precej preprosta.

```
public interface IEntityService<T> where T: EntityBase {
  T Get(int entityID);
  IEnumerable<T> GetList();
  void Delete(T entity);
  void SaveOrUpdate(T entity);
}
```

Konkretna implementacija je prav tako precej preprosta in samo kliče ustrezne metode shrambe. Določene storitve, ki dedujejo iz te, lahko seveda dodajo tudi bolj napredne metode, tako kot npr. storitev za pridobivanje umetnikov.

```
public interface IArtistService: IEntityService<Artist> {
  IEnumerable<Artist> GetList(string category);
  void SaveByName(Artist artist);
}
```

Prva metoda vrača seznam umetnikov za določeno kategorijo, druga pa shrani umetnika in kot ključ, po katerem poišče umetnika, uporabi njegovo ime in ne identifikacijske številke.
Storitve, ki EntityService uporabljajo kot izhodišče so naslednje:

- ArtistService - služi za upravljanje z umetniki oziroma avtorji določenega vnosa.
- CategoryService - služi za upravljanje s kategorijami.
- ContentService - služi za upravljanje s posameznimi vnosi (npr. albumi, filmi).
- ImageService - služi za upravljanje s slikami vnosov.
- MediaTypeService - služi za upravljanje z različnimi tipi medijev (knjiga, CD, DVD itd.).

4.3.4. Prezentacijska plast

Prezentacijska oziroma predstavitvena plast sloni na poslovni plasti. Dočim poslovni in podatkovno plast lahko razvijamo skoraj hkrati, je prezentacijsko plast najbolje razvijati takrat, ko je poslovna plast v večini že končana in testirana. Tako se izognemo nepotrebnemu popravljanju oziroma dopolnjevanju poslovne plasti in se lahko osredotočimo samo na izdelavo prezentacije. Pri več razvijalcih, kjer nekaj razvijalcev razvija uporabniški vmesnik, del pa poslovno plast, pa je potrebno doreči vmesnik (API) poslovne plasti, nato pa se lahko uporabniški vmesnik razvija na tem vmesniku, implementacija poslovne plasti pa poteka vzporedno.

Priprava okolja

V mojem primeru za prezentacijsko plast uporabljam tehnologijo ASP.NET MVC, prilagojen za delo z vstavljanjem odvisnosti. Tako se vsak nadzornik kreira preko orodja StrucrureMap, kar se doseže tako, da se implementira svojo tovarno nadzornikov in se jo registrira v ASP.NET MVC sistem.

```
internal class StructureMapControllerFactory: DefaultControllerFactory {
  protected override IController GetControllerInstance(Type controllerType) {
    if (controllerType != null)
      try {
        var controller = (IController)ObjectFactory.GetInstance(controllerType);
        return controller;
      }
      catch (StructureMapException) {
        Debug.WriteLine(ObjectFactory.WhatDoIHave();
        throw;
      }
    return null;
  }
}
protected void Application_Start() {
  ControllerBuilder.Current.SetControllerFactory(new // registracija tovarne
    StructureMapControllerFactory());
}
```

Poleg ustrezne uporabe tovarne nadzornikov je potrebno v sistem dodati še podporo za enoto dela in inicializirati orodje StructureMap. Vsa inicializacija sistema se dogaja v Global.asax.cs.

```
public class MvcApplication: System.Web.HttpApplication {
  public MvcApplication() {
    PreRequestHandlerExecute += Application_BeginRequest;
    PostRequestHandlerExecute += Application_EndRequest;
  }

  protected void Application_Start() {
    RegisterRoutes(RouteTable.Routes);
    // StructureMap initialization.
    Bootstrapper.ConfigureStructureMap();
    // Custom model binders.
    ModelBinders.Binders[typeof(Content)] = new ContentModelBinder(
      ObjectFactory.GetAllInstances<ContentModelBinderBase>());
    // Use StructureMap for controller creation.
    ControllerBuilder.Current.SetControllerFactory(new
      StructureMapControllerFactory());
  }

  private static void Application_BeginRequest(object sender, EventArgs e) {
    UnitOfWork.Start();
  }

  private static void Application_EndRequest(object sender, EventArgs e) {
    if (UnitOfWork.IsStarted) {
      UnitOfWork.Current.Flush();
      UnitOfWork.Current.Dispose();
    }
  }

}
```

Pri pripravi sistema je potrebno opozoriti tudi na uporabo posebnega model binderja za Content. Le-ta omogoča, da nadzorniške metode sprejmejo parametre tipa Content, ki pa so seveda lahko različnih konkretnih tipov (npr. Album, Movie ipd.).
Kot je razvidno iz kode, se StructureMap inicializira preko bootstrapper razreda, ta pa uporablja ti. register.

```
public static class Bootstrapper {
  public static void ConfigureStructureMap() {
    var registry = new DefaultServiceRegistry();
    ObjectFactory.Configure(x => {
      x.AddRegistry(registry);
      x.ForRequestedType<IImageManager>()
        .TheDefaultIsConcreteType<WebImageManager>();
      x.ForRequestedType<IOpenIdService>().
        .TheDefaultIsConcreteType<OpenIdService>();
      x.IncludeConfigurationFromConfigFile = true;
    });
  }
}
```

Pri inicializaciji StructureMap orodja omogočimo tudi inicializacijo preko nastavitvene datoteke StructureMap.config.

Varnost

Kot je opisano v poglavju 4.3.3 (poglavje RoleService), mora uporabnik za izvedbo nekaterih akcij imeti ustrezno vlogo. Katera vloga je potrebna za neko akcijo, lahko določimo preko atributa na metodi nadzornika ali na nadzorniku samem (glej 4.2.2, Akcijski filtri). Za pravilno delovanje pa je potrebno tudi, da sistem lahko pridobi ustrezne informacije o uporabnikih in vlogah. Za ta namen je potrebno implementirati RoleProvider in ga registrirati v RoleManager. RoleProvider za delovanje uporablja RoleService.

```
public class AVCatRoleProvider: RoleProvider {

  ...

  public override bool IsUserInRole(string userName, string roleName) {
    return _roleService.IsUserInRole(userName, roleName);
  }

  public override string[] GetRolesForUser(string userName) {
    var roles = from r in _roleService.GetRolesForUser(userName)
                select r.RoleName;
    return roles.ToArray();
  }

}
```

Registracija v RoleManager pa poteka preko konfiguracijske datoteke.

```
<configuration>
  ...
  <system.web>
    <roleManager enabled="true" defaultProvider="AVCatRoleProvider">
      <providers>
        <clear/>
        <add applicationName="/" name="AVCatRoleProvider"
          type="AVCat.Web.Security.AVCatRoleProvider, AVCat.Web"/>
      </providers>
    </roleManager>
  ...
```

To je vse, kar je potrebno narediti za delujoč sistem avtorizacije.

Uporabniške kontrole

Ker se informacije v določeni obliki večkrat prikazujejo na uporabniškem vmesniku, je za določene tipe smiselno izdelati posebno kontrolo ali pa pogled, ki skrbi za prikaz teh informacij. Tako sem za potrebe te aplikacije izdelal več kontrol in pogledov oziroma predlog.

Kontrole

Ena kontrola skrbi za ostranjevanje (*paging*) – za podan izvor podatkov izpiše trenutno stran, število prikazanih in vseh zapisov ter možnost navigacije med stranmi. Sama kontrola je precej preprosta, brez večjih posebnosti.

Druga kontrola pa je namenjena prikazu podatkov v tabelarični obliki – *grid*. Kontrola je nekoliko povzeta po Telerik Grid (www.telerik.com/products/aspnet-mvc/grid.aspx) in prilagojena za potrebe moje aplikacije. Kontrola za razliko od Telerik kontrole ne podpira sortiranja, grupiranja itd., za ostranjevanje pa uporablja kontrolo, omenjeno v prejšnjem odstavku. Pri izdelavi kontrole je uporabljenih več pristopov in vzorcev – vzorec tovarne, princip ene odgovornosti, učinkoviti vmesnik, razvoj po pogodbi itd., tako da si jo je vredno ogledati nekoliko bližje.
Grid kontrola je sestavljena iz dveh delov – konfiguracija oziroma nastavitve kontrole ter generiranje HTML kode na podlagi konfiguracije. Za konfiguracijo skrbijo ti. *builder* razredi, preko katerih lahko celotno kontrolo s pomočjo učinkovitega vmesnika konfiguriramo tako:

```
Html.Grid(Model).
  .Name("Grid")
  .Columns(columns => {
    columns.Bind(category => category.Name);
    columns.Command(commands => commands.Delete());
  })
  .DataKeys(keys => keys.Add(category => category.CategoryID))
```

Grid v zgornjem primeru je ti. ekstenzijska oziroma razširitvena metoda (*extension method*) nad razredom HtmlHelper. Metoda vrne prej omenjeni *builder* razred, preko katerega se »zgradi« grid kontrola.

```
public static GridBuilder<T> Grid<T>(this HtmlHelper html, IEnumerable<T> source){
  return new GridBuilder<T>(new Grid<T>(html.ViewContext, source));
}
```

Preko builder razreda se nato lahko konfigurira grid.

```
public class GridBuilder<T> {
  ...
  public GridBuilder<T> Name(string name) {
    _grid.Name = name;
    return this;
  }

  public GridBuilder<T> Actions(Action<GridActionSettingsBuilder> action) {
    action(new GridActionSettingsBuilder(_grid.Actions));
    return this;
  }
  ...
}
```

Razlogov za tak pristop je več. En je sledenje principu ene odgovornosti kot tudi deljenja odgovornosti. Tako je Grid razred zadolžen za generiranje HTML kode, GridBuilder razred pa za konfiguracijo. Tako tudi uporabnik ni obremenjen s podrobnostmi o sami kontroli, temveč se mora posvečati le pravilnim nastavitvam kontrole. Kot je razvidno, je enak pristop uporabljen tudi za dele Grid kontrole, npr. akcije, stolpce ipd.

Predloge pogledov
Za ASP.NET MVC 2 lahko za vsak tip definiramo svojo predlogo za pregledovanje in za urejanje – *Display Template* in *Editor Template*. Sam sem definiral posebne predloge za izvajalca, žanr, tip nosilca ter datum. Tako se potem npr. pri urejanju namesto navadnega vnosnega polja pojavi predloga, ki je lahko tudi precej kompleksna.

Predloge za pregledovanje so v mojem primeru preproste, samo izposovanje datuma, imena izvajalca ipd. Predloge za urejanje pa so precej bolj kompleksne. Tako je pri predlogi za datum poleg vnosnega polja na voljo tudi gumb, ki prikaže koledar, predloga za tip nosilca pa prikaže vse nosilce v izbirnem polju. Bolj kompleksni sta ostali dve predlogi – predloga izvajalca poleg izbire obstoječega dopušča tudi vnos novega izvajalca, predloga žanrov pa izbrane žanre prikaže na seznamu.

Večjezičnost

Podporo za večjezičnost je potrebno zagotiviti na dveh mestih - na uporabniškem vmesniku in na poslovni logiki. Uporabniški vmesnik v tem primeru predstavlja ASP.NET MVC pogled (View), poslovna logika pa se vedno kliče samo preko akcij na ASP.NET MVC nadzorniku (controller). Tako sem izdelal dva razreda, ki služita kot osnova za vse ostale razrede. Za uporabniški vmesnik je to ViewPageBase, za nadzornike pa BaseController.

```
public class BaseController: Controller {

  ...

  protected override void Initialize(RequestContext requestContext) {
    base.Initialize(requestContext);
    CultureHelper.InitializeLanguage(requestContext.HttpContext);
  }

}

public class ViewPageBase<TModel>: ModelViewPage<TModel> where TModel: class {

  ...

  protected override void InitializeCulture() {
    base.InitializeCulture();
    CultureHelper.InitializeLanguage(Context);
  }

}
```

Kot je razvidno, se vsa inicializacija jezika dogaja v pomožnem razredu CultureHelper.

```
internal static class CultureHelper {

  internal static void InitializeLanguage(HttpContextBase context) {
    InitializeLanguageInternal(
      cookieName => context.Request.Cookies.Get(cookieName), // getCookieFunc
      () => context.User.Identity,                           // getIdentityFunc
      cookie => context.Response.Cookies.Add(cookie),        // setCookieProc
      () => context.Request.UserLanguages                 // getUserLanguagesFunc
    );
    // Set the culture for currently active unit of work-used for data access.
    if (UnitOfWork.Current != null)
      UnitOfWork.Current.InitializeCulture(CultureInfo.CurrentCulture);
  }

  internal static void InitializeLanguage(HttpContext context) {
    InitializeLanguageInternal(
      cookieName => context.Request.Cookies.Get(cookieName), // getCookieFunc
      () => context.User.Identity,                           // getIdentityFunc
```

```
      cookie => context.Response.Cookies.Add(cookie),        // setCookieProc
      () => context.Request.UserLanguages                  // getUserLanguagesFunc
    );
  }

  private static void InitializeLanguageInternal(Func<string, HttpCookie>
    getCookieFunc, Func<IIdentity> getIdentityFunc,
    Action<HttpCookie> setCookieProc, Func<string[]> getUserLanguagesFunc) {
    var defaultLanguage = "en";
    var cookie = getCookieFunc("language");
    // No cookie (ignore expired cookies).
    if ((cookie == null) || (cookie.Value == null) ||
      (cookie.Expires < DateTime.UtcNow)) {
      var identity = getIdentityFunc();
      // No cookie but user is logged in
      if (identity.IsAuthenticated) {
        // Get the user's language and set the cookie for future use.
        var profile =
          ObjectFactory.GetInstance<IPersonalizationService>()
          .Get(getIdentityFunc().Name);
        cookie = new HttpCookie("language", profile.Language.LanguageName) {
          Expires = DateTime.Now.AddDays(7)
        };
        setCookieProc(cookie);
      }
    }
    if (((cookie == null) || string.IsNullOrEmpty(cookie.Value)) &&
      (getUserLanguagesFunc().Length > 0))
      defaultLanguage = getUserLanguagesFunc()[0];
    // Get the language name.
    var cultureName = ((cookie != null) && !string.IsNullOrEmpty(cookie.Value))
      ? cookie.Value : defaultLanguage;
    // Set the language for current thread - used on UI.
    Thread.CurrentThread.CurrentUICulture =
      CultureInfo.GetCultureInfo(cultureName);
    Thread.CurrentThread.CurrentCulture =
      CultureInfo.CreateSpecificCulture(cultureName);
  }

}
```

Torej, pri vsakem klicu metode na nadzornem razredu oziroma pri prikazu strani se kliče inicializacija jezika. Pri inicializaciji se najprej preveri, če je ta že inicializiran, torej če že obstaja ustrezen piškotek, v katerem je zapisano ime jezika. Če piškotek še ne obstaja, se preveri, če je trenutni uporabnik prijavljen v aplikacijo. Če je, se preko njegovega profila pridobi jezik, ki ga uporabnik lahko spremeni preko svojih nastavitev profila. Ta jezik se zapiše v piškotek. Nato pa se jezik iz piškotka zapiše v trenutno aktivno kulturo.

Razširljivost

Večina nadzornih razredov in pogledov je standardnih ASP.NET MVC, prikazovanje materiala pa je nekoliko drugačno, saj je narejeno tako, da je razširljivo.
Nadzorni razred niti ni poseben, je nadzorni razred, ki ponuja metode za pridobivanje seznama materialov, podrobnosti o enem materialu ter dodajanje, urejanje in brisanje materiala. Kot material se tretira vsak razred, ki deduje iz Content. Vsa posebnost je v tem, da se za delovanje nanaša na ContentModelBinder, o katerem je več omenjeno v poglavju Priprava okolja.

Pogled za prikaz in pogled za urejanje oziroma dodajanje pa sta nekoliko posebna, saj oba izpišeta nekaj skupnih informacij o materialu, podrobnosti pa prepustita drugim pogledom, ki so specifični za posametni tip pogleda. Tako npr. logika za pogled za prikaz podrobnosti izgleda tako:

```
...
<div class="sectionContent">
  <% Html.RenderPartial(Model.ViewName, Model.Content); %>
</div>
...
```

Pri tem Model.ViewName vsebuje ime pogleda, ki naj prikaže podrobnosti, napolni pa se preko metode v nadzornem razredu.

```
public class CatalogController: BaseController {

  ...

  public ActionResult View(int id) {
    var content = _contentService.Get(id);
    var viewInfo = ContentManager.GetViewInfo<ContentViewInfo>(content);
    var viewModel = new ContentViewModel {
      Content = content,
      Title = viewInfo.Title,
      ViewName = viewInfo.ViewName,
      Category = ContentManager.GetCategory(content)
    };
    return View(viewModel);
  }

}
```

ContentManager pa je razred, ki za nek tip materiala vrne ustrezno informacijo o pogledu, ki naj se uporabi. Pri tem je pogled specifičen glede na platformo, tako da je za splet to ContentViewInfo, implementiran v Web projektu, lahko pa bi implementiral tudi svoj view info za npr. winforms.

```
public abstract class ContentManager {

  ...

  public static T GetViewInfo<T>(Content content) {
    if (content == null)
      throw new ArgumentNullException("content");
    // Find provider that supports given content.
    var contentType = content.GetType();
    var manager = Instances.Values.FirstOrDefault(x =>
      x.ContentType == contentType);
    if (manager == null)
      throw new NotSupportedException(string.Format("Content type '{0}' is not
        supported.", content.GetType().FullName));
    return (T)manager.CreateViewInfo(content);
  }

}
```

Lastnost Instances vsebuje seznam vseh registriranih managerjev, registrirajo pa se preko nastavitvene datoteke.

```
<configuration>
  <configSections>
    <sectionGroup name="avCat" type="AVCat.Core.Configuration.AVCatSectionGroup,
      AVCat.Core">
      <section name="contentManagers"
        type="AVCat.Core.Configuration.ContentManagersSection, AVCat.Core"/>
    </sectionGroup>
  </sectionGroup>
...
  <avCat>
    <contentManagers>
      <add category="Movies"
        managerType="AVCat.Web.Models.ContentManagement.MovieProvider,
        AVCat.Web.Models" />
      <add category="Music"
        managerType="AVCat.Web.Models.ContentManagement.MusicProvider,
        AVCat.Web.Models" />
    </contentManagers>
  </avCat>
...
</configuration>
```

Vsaka implementacija ContentManager mora implementirati metodi za kreiranje nove instance materiala ter metodo za shranjevanje materiala. Poleg tega mora implementirati tudi lastnosti, ki določa, kakšen tip materiala je podprt.
Za pregled, kako so posamezne komponente povezane v celoto, pa svetujem pregled priložene izvorne kode. Pri tem bi rad še enkrat poudaril to, kar sem omenil že v poglavju 4.3.2 – aplikacija ni namenjena kot primer, kako naj se razvija celotna aplikacija, namenjena je prikazu uporabe različnih pristopov in vzorcev. Tako sem za potrebe tega diplomskega dela marsikateri problem rešil drugače, bolj kompleksno, kot bi ga drugače.

5. Sklepne ugotovitve

Aplikacije seveda lahko izdelamo na veliko različnih načinov. Vendar pa uporaba pravih tehnologij in pristopov izdelavo aplikacije lahko zelo olajša, nekateri pristopi pa imajo celo za posledico enostavno razširljivo ter robustno aplikacijo. Pri tem je seveda še enkrat potrebno poudariti, da so vsi pristopi in vzorci, tako tisti, ki sem jih omenil v tem delu, kot tudi ostali, še vseeno samo vzorci. Pravilna izbira pristopa k nekemu problemu je lahko razvijalcu v veliko pomoč, po drugi strani pa lahko naleti tudi na probleme, če se določenih pristopov drži brez ustreznega premisleka, zakaj tak pristop sploh uporabiti.

Pri tem bi opozoril še na dokaj pogosto napako razvijalcev – ti. ponovno odkrivanje tople vode. Namreč veliko razvijalcev, ko naleti na nek problem, ga hočejo rešiti sami, na svoj način. To je sicer pozitivno iz stališča, da se razvijalec kaj novega nauči, vendar če že obstajajo splošno uporabne rešitve, je v veliki večini primerov izdelava lastne rešitve lahko velika napaka. Konkretno v mojem primeru sem najprej hotel sam izdelati ekvivalent O/RM ogrodja, vendar sem pri tem naletel na precej težav (predvsem robni primeri), da sem se nato odločil za uporabo splošno priznanega O/RM orodja, NHibernate.

V določenih primerih neka tuja rešitev mogoče ne ponuja dovolj dobrih performanc ali česa drugega in včasih je potrebno rešitev res razviti sam, vendar sploh če je neko orodje v širši uporabi, je verjetnost za to, da ne bo zadoščalo konkretnim potrebam, precej majhna. Enako izkušnjo sem imel tudi na delovnem mestu. Razvili smo svoj ekvivalent O/RM orodja in uporabili tudi nekaj drugih pristopov, ki se tretirajo vse prej kot dobri. Sedaj je vzdrževanje te aplikacije zelo mukotrpno opravilo, zelo velikokrat ravno zaradi miselnosti »mi vemo najbolje«.

Vendarle pa so določeni vzorci, tukaj mislim predvsem na SOLID principe, tako univerzalni, da bi se jih morali vedno ali vsaj večinoma držati. Še pomembneje pa je seveda ohraniti trezno glavo in tudi vzorce uporabiti kjer je smiselno in ne za vsako ceno. Tudi nepravilna uporaba sicer dobrih vzorcev lahko prinese visoko ceno.

Seznam slik

Seznam tabel

Literatura in viri

[1] (2009) Deployment Patterns. Dostopno na: http://msdn.microsoft.com/en-us/library/ms998478.aspx

[2] M. Feathers, *Working Effectively with Legacy Code*.: Prentice Hall, 2004.

[3] M. Fowler. (2005) MF Bliki: FluentInterface. Dostopno na: http://martinfowler.com/bliki/FluentInterface.html

[4] M. Fowler. P of EAA: Repository. Dostopno na: http://martinfowler.com/eaaCatalog/repository.html

[5] J. Gehtland. (2004) Enterprise.NET Community: NHibernate. Dostopno na: http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernate

[6] Hibernate Tutorial, generator element. Dostopno na: http://www.allappforum.com/hibernate/hibernate_o_r_mapping_generator_element.htm

[7] (2009) How do you define software architecture? Dostopno na: http://www.codingthearchitecture.com/pages/define.html

[8] (2000) IEEE-Std-1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems. Dostopno na: http://www.enterprise-architecture.info/Images/Documents/IEEE%201471-2000.pdf

[9] J. Miller. (2009) The Unit Of Work Pattern And Persistence Ignorance. Dostopno na: http://msdn.microsoft.com/en-us/magazine/dd882510.aspx

[10] (2009) Model-View-Controller. Dostopno na: http://en.wikipedia.org/wiki/Model_view_controller

[11] L. Northrop. (2003) The Importance of Software Architecture. Dostopno na: http://csse.usc.edu/GSAW/gsaw2003/s13/northrop.pdf

[12] (2009) Object-relational mapping. Dostopno na: http://en.wikipedia.org/wiki/Object-relational_mapping

[13] (2009) Open/closed principle. Dostopno na: http://en.wikipedia.org/wiki/Open/closed_principle

[14] RoleProvider Class. Dostopno na: http://msdn.microsoft.com/en-us/library/system.web.security.roleprovider.aspx

[15] (2007) Software Architecture: What Is Software Architecture? Dostopno na: http://blog.softwarearchitecture.com/2007/05/what-is-software-architecture.html

[16] (2010) Solid (object-oriented design). Dostopno na: http://en.wikipedia.org/wiki/Solid_%28object-oriented_design%29

[17] (2002) The Philosophy of Extensible Software. Dostopno na: http://accu.org/index.php/journals/391

[18] (2008) The Repository Pattern. Dostopno na: http://blogs.hibernatingrhinos.com/nhibernate/archive/2008/10/08/the-repository-pattern.aspx

[19] (2008) What are MVP and MVC and what is the difference? Dostopno na: http://stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference

[20] (2008) Writing Better Code. Dostopno na: http://www.c-sharpcorner.com/UploadFile/rmcochran/extensibility02202008202814PM/extensibility.aspx