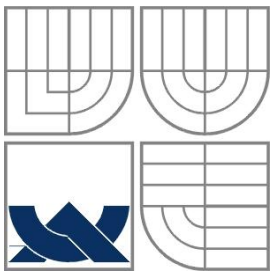
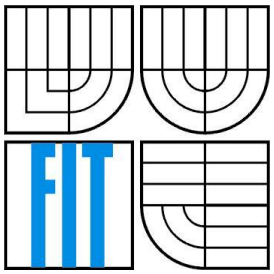


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENEROVÁNÍ INFORMAČNÍHO SYSTÉMU

INFORMATION SYSTEM GENERATING

DIMPLOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. PETR VOBORNÍK
AUTHOR

VEDOUCÍ PRÁCE Ing. LADISLAV RUTTKAY
SUPERVISOR

BRNO 2011

Abstrakt

Práce stanovuje požadavky pro implementaci generátoru informačních systémů. Zaobírá se nalezením společných prvků informačních systémů. Dále specifikuje požadavky na zápis konceptuálního modelu. Pokračuje popisem obecného uživatelského pohledu na systémy. Na základě zmíněných analýz jsou vybrány implementační technologie a navrhnuta architektura generovaných systémů. Generované systémy jsou navrženy tak, aby poskytovaly základní operace pro práci s položkami systému. Mimo to jsou diskutovány různé způsoby generování kódu. Je navržen metamodel pro uchování konceptuálního modelu systému a jsou diskutovány jeho možné zápisy. Nakonec jsou popsány problémy, které musí řešit generátor při generování kódu.

Abstract

The work sets out requirements for the implementation of information systems generator. It deals with finding common elements of information systems. Further it specifies requirements for registration of a conceptual model. Then it goes on to describe user's general point of view of systems. Implementation technologies are chosen and architecture of generated systems is designed based on mentioned analysis. Generated systems are designed to provide basic operations for working with items of the system. Moreover, different ways of generating code are discussed. Metamodel of conceptual model is designed and its possible types of representation are discussed. At the end problems which generator deals with are described.

Klíčová slova

Informační systém, generování kódu, .NET Framework, ASP.NET, MVC, web, konceptuální model, Razor, návrhové vzory

Keywords

Information system, code generation, .NET Framework, ASP.NET, MVC, web, conceptual model, Razor, design patterns

Citace

Voborník Petr: Generování informačního systému, diplomová práce, Brno, FIT VUT v Brně, 2011

Generování informačního systému

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Ladislava Ruttkaye.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bc. Petr Voborník
25. 5. 2011

Poděkování

Chtěl by poděkovat zvláště Ing. Ladislavu Ruttkayovi za odbornou pomoc při vedení této práce.

Obsah

1 Úvod

S rozvojem informačních technologií se zvyšují požadavky na informační podporu práce s podnikovými daty. Stále více firem vyžaduje pro zefektivnění své činnosti spravovat podniková data a částečně nebo úplně automatizovat práci s nimi.

Uvedené požadavky vedou k vývoji nový informačních systémů specifických pro doménu daného podniku. Vývoj informačních systémů je složitý a zdlouhavý proces, co se odráží v ceně. Investoři systémů si přejí, aby jejich cena byla co možná nejnižší. Dodavatelé proto hledají způsoby, jak cenu snížit – hledají vhodné prostředky, které usnadní proces vývoje. Značnou část práce, při vývoji informačních systémů, tvoří relativně stejná opakující se práce. Je to např. implementace formulářů uživatelských rozhraní nebo tvorba vrstvy pro přístup k datům, která má přesně danou strukturu a mění se jen počty a názvy položek. Je tedy stanoven postup, přesně dojít k požadovanému výsledku dle určitých vstupních informací. Tento fakt přímo nabádá takový proces automatizovat. V takovém případě výsledek automatizace představuje program, který vykonává práci programátora – tvoří zdrojový kód. Pro automatickou tvorbu zdrojového kódu se využívá pojem: generuje kód. Tedy program je generátorem zdrojového kódu.

První část práce se zaměřuje na stanovení podkladů pro návrh a implementaci generátoru kódu informačních systémů. Nejprve analyzuje společné prvky informačních systémů mezi různými doménami, tím specifikuje požadavky na zápis modelu domény podniku. Dále pak pokračuje identifikací základních operace s entitami domény. Kladou se požadavky na uživatelské rozhraní. Další kapitole jsou ze zmíněných požadavků porovnány a vybrány implementační technologie. Na to navazuje kapitola, která se zabývá návrhem architektury systému. Jsou v ní porovnány různé způsoby reprezentace modelu, datové vrstvy a další částí systému.

Druhá část práce se zabývá generováním. Nejprve jsou rozebrány různé způsoby generování kódu a je navrhnuta minimalistická architektura generátoru kódu. Ta je doplněna o schéma metamodelu pro vnitřní reprezentaci modelu generovaného systému. Na to navazuje rozbor způsobů zápisu modelu. Jeden je vybrán a navrhnut. Předposlední kapitola se zabývá tvorbou úloh pro generátor, pro generování informačních systému dle jejich navrhnuté architektury. Popisují se tam úskalí, které se vyskytly při jejich tvorbě.

V poslední části práce jsou zváženy možnosti rozšíření. Nakonec v závěru je diskutován výsledek práce.

Diplomová práce je přímým pokračováním semestrálního projektu. V upravené formě jsou z něj použity kapitoly obsažené v první části práce.

2 Specifikace cíle práce

Tato kapitola se zabývá stanovením cílů práce a definuje pojem informační systém, tak jak jej budeme chápat.

2.1 Informační systém

Informační systém je definován [1] jako:

> *Otevřený systém, jehož nosič používá konceptuální zdroje, konkrétně informace. Z toho vyplývá, že informační systém nenakládá s hmotnými zdroji, nýbrž zdroji nehmotnými. Nakládáním rozumíme provádění transformačních funkcí nad konceptuálními zdroji. Nečiní to však libovolně, nýbrž tak, že tyto informace modelují skutečné chování jiného fyzického systému (např. podniku). Informační systém tedy na nehmotné – virtuální úrovni homomorfně modeluje svůj fyzický vzor, pro jehož řízení je obvykle vytvářen. Vzhledem k tomu, že model nikdy nemůže postihnout veškeré chování a vlastnosti svého vzoru, je virtuální kopie pořizována vždy na takové úrovni abstrakce, která je pro příslušnou úroveň vhodná.*

Z pohledu práce budeme cílový systém chápat jako počítačový program, který poběží na určitém běhovém prostředí. Systém bude pracovat s daty získaných od uživatelů pomocí uživatelského rozhraní - klienta, případně bude komunikovat s jinými systémy. Získaná a upravená data bude ukládat do datového úložiště.

2.2 Generátor kódu

Hlavním cílem práce je vytvoření generátoru informačních systémů. Jako generátor kódu chápeme systém, jehož vstupem je modelovaná doména a výstupem zdrojové kódy cílového systému. Z aplikačního hlediska je to nástroj, jehož účelem je urychlení vývoje cílového systému.

2.3 Požadavky na fáze vývoje cílových informačních systémů

Při vývoji informačních systémů se využívají různé metodiky. Nebudeme se zde zaměřovat na žádnou konkrétní metodiku, ale specifikujeme fáze, které jsou v mnoha z nich zastoupeny, zanedbáváme přitom iterování různých fází.

Vývoj informačních systémů lze rozdělit do následujících fází:

1. Úvodní studie
2. Analýza
3. Návrh
4. Implementace
5. Zkušební provoz
6. Nasazení

Budeme předpokládat znalost podstaty jednotlivých fází a zaměříme se jen na jejich provedení, výstup, či popis z pohledu využití generátoru.

Úvodní studii a analýzu procesů, toků dat a případů užití provede analytik systému. Získané poznatky analytik zapíše do konceptuálního modelu.

Ve fázi návrhu se provádí návrh architektury informačního systému. Při použití generátoru systému je uživatel odkázán na architektury, které dokáže generátor generovat. Předpokládáme tedy, že generátor obsahuje takovou architekturu, která vyhovuje všem generovaným informačním systémům, nebo obsahuje více architektur a dle konceptuálního modelu dokáže rozhodnout, kterou využít. Případně umožňuje dodání specifikace požadované architektury. Tedy samotný návrh systému se při generování neprovádí a je vybrána již předem navrhnutá architektura systému.

Ve fázi implementace přichází na řadu generátor. Na základě konceptuálního modelu a předpřipravených architektur se provede generování kostry systému. Lze předpokládat, že specifikace konceptuálního modelu nebudou obsahovat všechny požadavky na cílový informační systém. Mohou to být speciální případy, které by bylo obtížné specifikovat. Proto musí být umožněno vygenerovaný systém ručně upravovat tak, aby bylo dosaženo cílových požadavků.

Fáze zkušebního provozu a nasazení mají stejnou podobu jako u klasického vývoje, jelikož generátor již byl využit.
[2]

Cílem je tedy:

- identifikovat společné prvky informačních systémů
- navrhnout způsob zápisu konceptuálního modelu či využít nebo modifikovat nějaký již existující
- navrhnout architekturu generovaných systému
- navrhnout nástroj – generátor, který by byl schopen generovat IS dle zapsaného konceptuálního modelu a dle navrhnuté architektury
- vybrat implementační a podpůrné technologie

3 Analýza a specifikace společných prvků informačních systémů

Aby bylo možno generovat informační systémy, musíme nejprve zjistit jejich společné vlastnosti. Při analýze informačních systémů se často postupuje tak, že se nejprve hledají případy užití. Tj. hledají se aktéři a operace, které mohou provádět. Dle nich modelujeme konceptuální schéma a procesy, které co nejlépe popisují cílovou doménu. Různé domény mohou obsahovat různé entity a procesy. Jinak řečeno: základní společnou vlastností IS je přítomnost entit a procesů. Dále se v této kapitole zabýváme analýzou systému z pohledu uživatele.

3.1 Entity

Entita reprezentuje v modelu reálný objekt modelovaného systému. Primární vlastností entity je, že má identitu. Identita zajišťuje, že entita nemůže být zaměněna s jinou např. i když se hodnoty jejich atributů shodují. Díky ní ji lze jednoznačně identifikovat během jejího životního cyklu. Respektive, díky identifikaci může mít nějaký životní cyklus. Pokud se na to pohlédne z opačného úhlu, tak entitou by měl být vždy takový objekt, který je nutné odlišit od ostatních kdykoliv během jeho života. Samotná identita je během života entity neměnná. Neměl by ji tedy představovat žádný atribut, který by se někdy mohl změnit. [3]

Entity by se daly rozdělit na nezávislé a závislé. Jak už sám název napovídá, nezávislé entity nejsou na žádné jiné závislé. Mohou existovat samostatně. Závislé entity vyžadují, aby byly svázány s jinou entitou. Tento druh vazby říká, že bez existence entity, na které jsou závislé, nemá smysl jejich vlastní existence.

3.1.1 Atributy

Důležitou součástí entit jsou jejich atributy, které modelují vlastnosti entit v reálném světě. Atribut je specifikován názvem a hodnotou. Hodnotou atributu může být jednoduchá hodnota, složená hodnota nebo odkaz na entitu. Dle typu hodnoty se dají atributy rozdělit na jednoduché, složené a vztahy. Za jednoduchou hodnotu se považuje údaj, u kterého není podstatná identita, ale pouze vlastní reprezentace – hodnota. Může to být např. název ulice či číslo domu. Složený atribut je struktura, která se skládá z více jednoduchý atributů a tvoří sémantický celek, ale stále, na rozdíl od entity, není důležitá jednoznačná identifikace. V určitém úhlu pohledu by mohly být složené hodnoty reprezentovány pomocí entit, které by obsahovaly jen jednoduché atributy nebo stejně reprezentované složené atributy. Takové způsoby se v určitých systémech vyskytují. Jejich nevýhodou je, že se tím stávají zbytečně složité – musí uchovávat identitu, tam kde by nemusely. [3] Samotný problém vypořádání se se složenými atributy se přenechá kapitole o generování doménové či datové vrstvy. Pokud je hodnotou atributu odkaz na entitu nebo kolekce odkazů na entity, představuje atribut část vztahu.

3.1.2 Vztahy mezi entitami

Mezi entitami mohou existovat vztahy. Vztahy lze rozdělit na binární a n-nární. N-nární vztah lze reprezentovat entitou. Taková entita se nazývá vazební. Obsahuje tři a více binárních vztahů.

Vazební entita nemusí vznikat jen pro n vazeb, ale může být použita i pro binární vztah. Je to vhodné např. když se chce vtah obohatit o určitý atribut. Každý konec nabývá určité kardinality (množství entit). Poté lze takto rozdělit vztahy [1]:

- 1:1 – entita z entitní množiny A se může vázat pouze na jednu entitu z entitní množiny B
- 1:N – entita z entitní množiny A se může vázat na jednu čí více entit z entitní množiny B
- M:N – více entit z entitní množiny A se může vázat na jednu čí více entit z entitní množiny B

3.1.2.1 Datové typy

Při definici jednoduchých atributů se musí specifikovat jeho typ. Je ho nutné vybrat z určité množiny, kde každý typ reprezentuje určitý druh dat. V systémech budeme uvažovat následující datové typy:

- celé číslo
- přibližné číslo s plovoucí řádovou čárkou
- přesné číslo s plovoucí řádovou čárkou
- znak
- znakový řetězec
- logická hodnota
- binární záznam
- datum/čas

Znakové řetězce, datum/čas či binární záznamy lze považovat za složené datové typy. Pro zjednodušení je budeme chápat jako jednoduché.

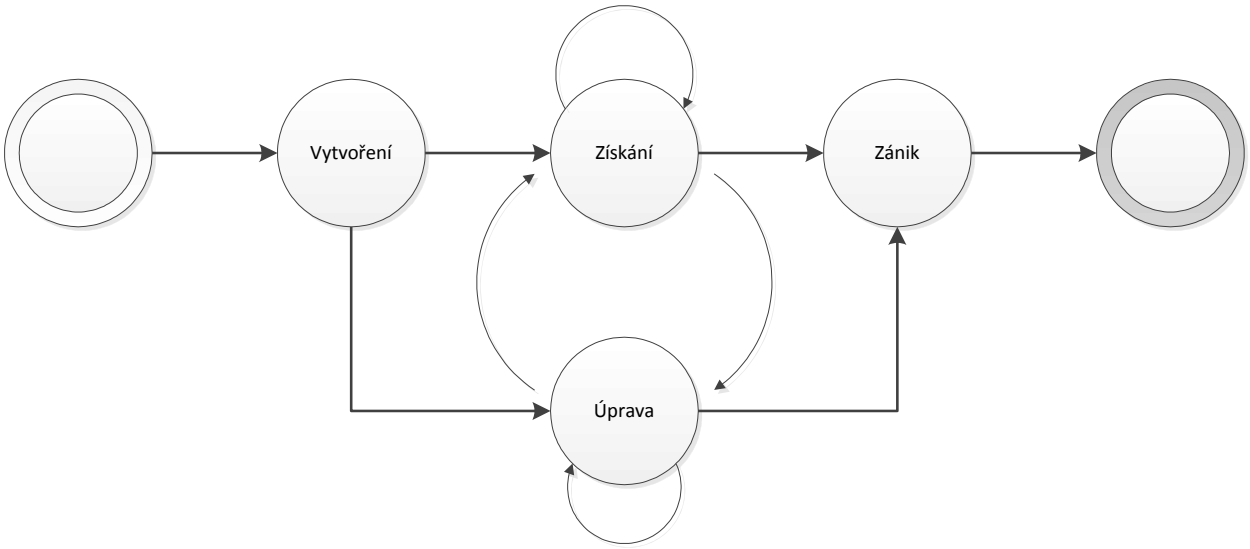
Při modelování entit domény lze narazit i na jiné datové typy, které by šlo považovat za jednoduché. Takové typy by se mohly modelovat jako složené, nebo by bylo nutno zavést nový jednoduchý typ. Z tohoto pohledu je vhodné, aby generátor umožňoval zavedení nového datového typu a specifikace práce s ním.

3.1.2.2 Integritní omezení

Pro správnou funkci systému je nutné, aby uložené údaje odpovídaly oboru hodnot vlastností domény. Použité datové typy můžou mít obor hodnot větší, než je obor hodnot domény. Proto je nutné u atributů specifikovat integritní omezení na obor hodnot. Běžně se setkáváme s omezeními, jako jsou: povinná položka, délka řetězce, nebo číselný interval. Ve většině systémů se navíc objevují omezení specifická pouze pro danou doménu např.: kladné číslo, jehož maximální hodnota může být počet měsíců od 1. 1. 2000 do dnes. Generátor tedy musí umožňovat specifikaci vlastních integritních omezení a práci s nimi.

3.2 Operace nad entitami

Existují 4 základní operace pro práci s entitami a to: vytvoření, získání/čtení, úprava a mazání/zánik, častou jsou označeny pod zkratkou CRUD (z anglického create, read/recieve, update, delete/destroy). Sekvencí těchto operací lze vyjádřit životní cyklus entity [3]. Popisuje ho obrázek 1.



Obrázek 1: Životní cyklus entity

Kombinací těchto základních operací lze vyjádřit všechny jiné, složitější operace. Identifikované procesy jsou tedy tvořeny posloupností těchto operací nad jednou či více entitami. Operace vytvoření a získání budou v systému vždy přítomny. Kdyby ne, bylo by možné entitní množinu do systému nezahrnout. Možnost úpravy a mazaní je závislá na dané doméně a mělo by ji být možno specifikovat v konceptuálním modelu.

Struktura složitějších procesů je doménově závislá. Jejich obecná specifikace v konceptuálním modelu by byla příliš složitá. Samotný zápis by mohl být složitější, než naprogramování programátorem. Proto se nepředpokládá, že by je měl být generátor schopen generovat.

Navzdory tomu je možné, že doména obsahuje mnoho procesů, které jsou si svojí strukturou podobné. Pokud jejich počet překročí určitou mez, stává se jejich generování žádoucí. Proto by měl generátor umožňovat rozšířit svou funkcionalitu o možnost specifikace generování dalších operací. Četnost operace je důležitá pro posouzení, zda se vyplatí naprogramovat rozšiřující modul pro generátor [4].

3.3 Uživatelský pohled

Z pohledu uživatele by měl systém poskytovat příjemné uživatelské rozhraní. Pod tímto označením si různí lidé představí různé věci, a proto by se měl výraz konkrétněji specifikovat. Pod uživatelským rozhraním se považuje vstupní bod systému, se kterým pracují lidé. Často se označuje pod pojmem klient. Typ aplikace určuje provedení klienta. Informační systémy se řádí do víceuživatelských systémů, se kterými může pracovat více uživatelů najednou. Je tedy zřejmé, že systém bude obsahovat více klientů a komunikační prvek, se kterým klienti komunikují, nazvěme jej server.

Uživatele většinou nezajímá, jakým způsobem klienti pracují a jak se dorozumívají se serverem. Zajímají je ale omezení a výhody z pohledu práce se systémem, které daný typ klienta obsahuje. Klienty lze dle místa provádění výpočtů a práci s lokálním diskem počítače rozdělit na tenké, tlusté a hybridní [5].

3.3.1 Tlustý klient

Tlustý klient provádí výpočty systému na lokálním zařízení, kde výsledky může i ukládat. Se serverem nemusí být v neustálém spojení. Stačí se synchronizovat jednou za čas. Výpočet na lokálním počítači umožňuje pracovat i s výpočetně složitými daty – např. multimediálními.

Možnost provádění výpočtu domény na lokálním počítači představuje nutnost nového získání klienta při úpravě systému nebo při přechodu na jiné zařízení. Dále dle typy zařízení, či jeho operačního systému je nutné implementovat různé verze klienta, nebo se omezit jen na určitý vyčet zařízení.

3.3.2 Tenký klient

Tenký klient slouží jen pro zobrazení výsledků operací a zadaní nových operací. Všechny operace se provádí na serveru. Všechna data jsou uložená na serveru. Provádění všech operací na serveru vede k tomu, že je nutné, aby byl klient se serverem vždy ve spojení. Komunikace a provádění v jednom bodě představuje nároky na přenosovou kapacitu a výpočetní výkon serveru. Tedy tenký klient nemůže pracovat s daty, které představují velkou zátěž na přenosovou kapacitu či s operacemi, které jsou výpočetně náročné. Oproti tlustému klientu změna v systému nevyžaduje stažení nové verze klienta, a pokud je klient dostatečně obecný, tak může být na zařízení již předem přítomen.

3.3.3 Hybridní klient

Hybridní klient je kombinací výše zmíněných a výhody či nevýhody jsou závislé na konkrétní implementaci.

3.3.4 Zhodnocení klientů

Jak je vidět, každý druh klienta má své výhody a nevýhody. Hlavní prací cílových informační systémy by měla být především úprava, uložení a zobrazení textových dat. Z tohoto pohledu vyhovuje jak tlustý, tak tenký klient.

Dále se neví na jakých různých zařízeních a v jakých lokalitách budou systémy využívány. Je tedy vhodné vybrat takový klient, který běží na většině rozšířených platforem (různé OS osobních počítačů, mobilní telefony, tablety…) a je schopen zobrazit cílová data. Implementace tlustých klientů pro různé platformy by mohla představovat značné úsilí vynaložené navíc. Vhodnější je tedy použít tenký klient, který existuje pro většinu platforem. Nejrozšířenějším takovým klientem je webový prohlížeč. Tudíž uživatelským rozhraním cílových systémů budou webové stránky.

3.4 Návrh uživatelského rozhraní

Z předešlých částí víme, že uživatelské rozhraní bude formou webových stránek. Jeho primárním cílem bude umožnit provádět základní (CRUD), či uživatelsky specifikované operace nad entitami domény. Musíme tedy specifikovat postup provedení daných operací z pohledu uživatele. Z nich odvodit požadované webové stránky navrhnou jejich obsah.

3.4.1 Vytvoření

Postup:

1. Zadaní příkazu vytvoření daného typu entity.
2. Zobrazení formuláře pro vyplnění údajů dané entity.
3. Zadání příkazu uložit.
4. Pokud zadané údaje vyhovují všem integritním omezením, provede se uložení. V opačném případě se provede návrat na bod 2.
5. Po uložení následuje přechod na jinou stránku, nebo reset formuláře pro možnost vytvoření nové entity stejného typu.

3.4.2 Zobrazení

Postup:

1. Nalezení požadované entity.
2. Zvolení příkazu zobrazit.
3. Zobrazení formuláře zobrazující entitu.

3.4.3 Úprava

Postup:

1. Nalezení požadované entity.
2. Zvolení příkazu upravit.
3. Zobrazení formuláře zobrazující entitu s možností editace hodnot.
4. Zadání příkazu uložit.
5. Pokud zadané údaje vyhovují všem integritním omezením., provede se uložení. V opačném případě se provede návrat na bod 3.
6. Po uložení se zobrazí hláška uloženo, nebo probíhá přechod na jinou stránku.

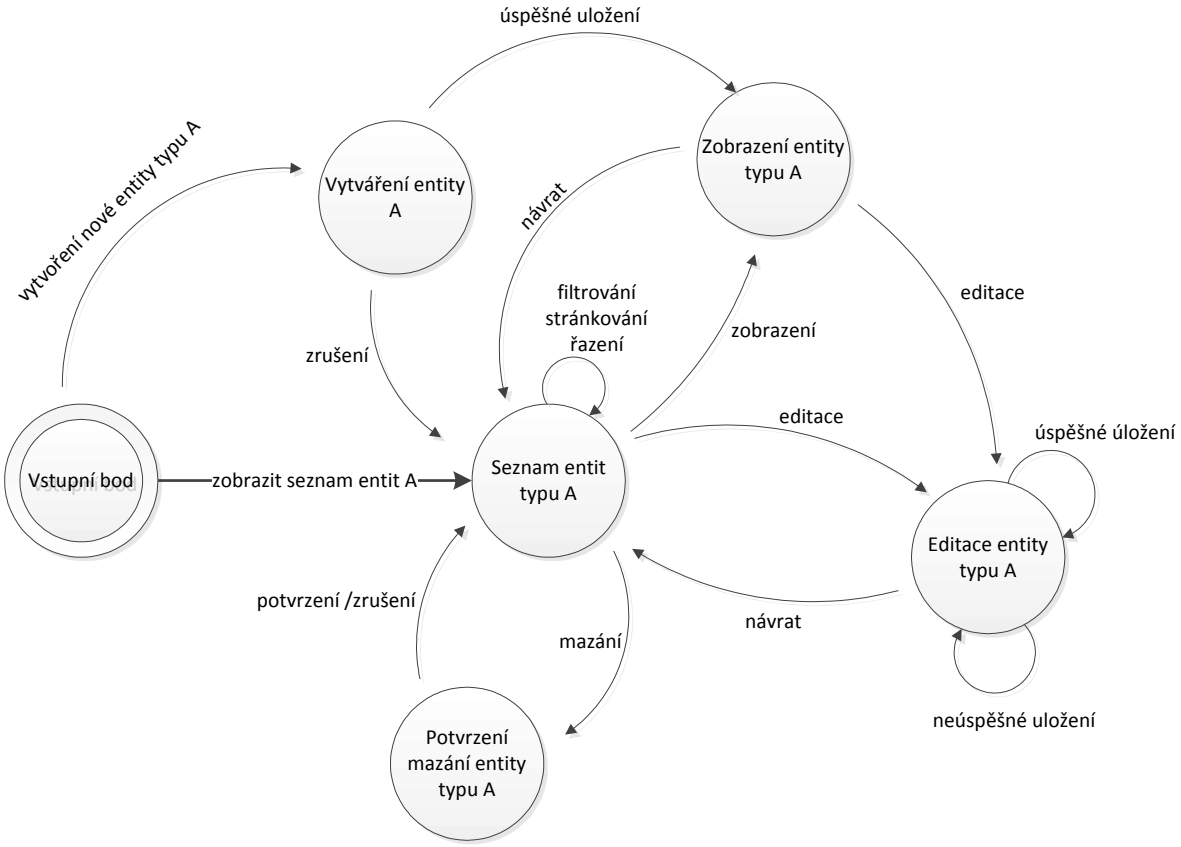
3.4.4 Mazání

Postup:

1. Nalezení entity.
2. Zvolení příkazu smazat.
3. Potvrzení smazání.
4. Provedení smazání.
5. Pokud nalezení entity představovalo zobrazení entity, musí se po smazání provést akce přechod na jinou stránku, protože entita již neexistuje a nejde tedy znovu zobrazit.

3.4.5 Struktura stránek

Každou operaci specifikuje její vlastní stránka, kde uživatel zadává detaily operace. Prerekvizitou pro zobrazení, editaci i mazání je nalezení entity. Systém musí obsahovat mechanismus nalezení entity. Tento mechanismus dobře zajistí stránka, která vypisuje seznam entit daného typu. Pro lepší uživatelskou přívětivost by měl seznam umožňovat vyhledávání, řazení a filtrování. Uživatelské operace, které pracují s určitou entitou, také vyžadují její nalezení. Tedy zadání provedení dané operace může probíhat stejně jako u zobrazení, editace či mazání. Možný diagram struktury stránek popisuje obrázek 2. V obrázku stavy představují zobrazenou stranu a přechody zvolený příkaz.



Obrázek 2: Struktura stránek pro jeden typ entity

3.4.6 Editace atributu a organizace formuláře

Podstatou vytváření a editace entity je editace jejích atributů případně závislých entit. Editací je myšleno zadání nebo úprava existující hodnoty. Složené atributy se mohou reprezentovat sekvencí jednoduchých atributů. Aby se mohl atribut editovat je nutné mít ve formuláři editor, který je svázaný s daným atributem. Pod pojmem editor si lze představit např. element typu input, případně jiný ovládací prvek vázaný na formulářový prvek HTML. Příkladem pokročilého editoru je jQuery Calendar pro zadaní data. Typ editoru je přímo závislý na datovém typu případně na bližší specifikaci (formátovací řetězec, subkategorie). Generátor tedy musí být schopen dle typu a bližší specifikace atributu určit vhodný editor a ten do formuláře vložit.

Množina editorů na stránce je z hlediska strojového plně dostačující. Pro uživatele by však byla matoucí – uživatel by nemusel vědět, který editor slouží k zadání kterého atributu. Je tedy vhodné formulář doplnit o údaje poskytující uživateli dostatek informací, díky kterým bude schopen formulář intuitivně vyplnit.

Dnešním standardem je u editoru uvádět:

- Popisek (label) – představuje textové označení jména atributu v jazyce uživatele
- Příznak povinné položky – bývá reprezentován znakem „*“ případně textem např. „povinná položka“.
- Místo pro zobrazení chyby – slouží k zobrazení textové hlášky reprezentující význam chyby, pokud uživatel zadá hodnotu, které nevyhovuje integritním omezením kladených na hodnotu atributu. Je zvykem text hlášky zdůraznit výraznou barvou textu, nebo pozadí, aby uživatel přehledně viděl, kde udělal chybu. Toto ověření by mělo probíhat

dvoufázově. Nejprve na klientu, po zadání hodnoty a před odesláním formuláře na server. Dále i na serveru, pro ověření, protože klientská ověření lze obejít.

Dále, pokud jsou k dispozici, je vhodné uvádět následující nadstandardní údaje:

- Upřesňující text – bývá reprezentován informační ikonkou, kde se u ní, po přejetí myší, zobrazí informační panel obsahující bližší informace o atributu domény. Cílem je, upřesnit uživateli, co daný atribut znamená, aby ho mohl vyplnit. Využívá se v případě, když atribut představuje vlastnost, která není v obecném povědomí lidí.
- Vodoznak – představuje text zobrazený v editoru (např. v podobě textového pole). Text je automaticky přepsán po zadání hodnoty do editoru. Text představuje upřesňující příkaz pro zadání hodnoty. Např. obsahuje formát textu reprezentující hodnotu.

Pokud editor očekává zadání hodnoty v určitém formátu, je vhodné, aby uživatelské rozhraní napomáhalo uživateli ji zadat např. pomocí grafického pomocníka, nebo masky, která zabrání vložení nežádoucích znaků. Všechny tyto údaje by měl generátor vyčíst z konceptuálního modelu a dle nich vygenerovat příslušný formulář.

Pro přehlednost je vhodné formulář patřičně organizovat. Např. aby byl pro každý atribut vyhrazen jeden řádek, kde počátky popisků, editorů, příznaků povinných položek atd. by byly horizontálně zarovnány tak, aby lícovaly s ostatními objekty v řádcích zbylých atributů. [6]

3.4.7 CRUD operace se závislými entitami

Definice závislých entit říká, že závislé entity nemohou existovat bez identifikující (základní) entity. Toto tvrzení lze vyložit tak, že nemá smysl vypisovat seznam entit toho typu nezávisle na identifikujících entitách. Tedy, seznam výpisu závislých entit by měl být součástí formuláře identifikující entity, nebo by měl být umožněn přechod na něj.

V případě že je výpis závislých entit součástí editačního formuláře identifikující entity. Mělo by být umožněno na tom formuláři provádět CRUD operace se závislými entitami. Pokud závislá entita neobsahuje příliš mnoho atributů, jejichž editory jsou prostorově rozměrné. Je možné závislé entity organizovat do tabulky a upravovat přímo. V případě, že entity mají mnoho atributů, nebo editory pro atributy zabírají více místa, než by se vešlo na řádek tabulky. Je vhodné do tabulky zobrazit jen souhrn entit a všechny atributy entity upravovat v jiném formuláři. Ten se může nacházet na jiné stránce. Případně lze využít editace v modálním okně. To vyžaduje zvýšené využití skriptů na straně klienta, ale vede k rychlejší práci v případě rychlé editace více závislých entit po sobě.

Editace závislých entit ve formuláře identifikující entity by se mělo chápat jako výchozí chování, ale také by se mělo počítat s ním, že ne vždy je to žádoucí, a proto by měl pro ně generátor umožnit i podporu CRUD operací jako u nezávislých entit.

3.4.8 Lokalizace

Generované systémy mohou být určené pro použití v různých zemích, kde lidé používají rozdílné jazyky. V tomto případě musí systémy podporovat změnu lokalizace – tj. změnu jazyka zobrazených textů, formátů dat, čísel apod. I v případě, kdy je systém určen pro použití jen v jednom kulturním prostředí, je vhodné systém vyvíjet stylem jako pro více kulturní prostředí. Napomáhá to k lepší udržovatelnosti kódu a k případnému budoucímu přechodu do více kulturního prostředí.

3.5 Bezpečnost

Jedním z hlavních požadavků kladených na informační systémy je bezpečnost. Zde se zaměříme pouze na část bezpečnosti webové aplikace. Techniky sociálního inženýrství, zkoumání postranních kanálů či infiltrace společnosti apod. budou opomenuty.

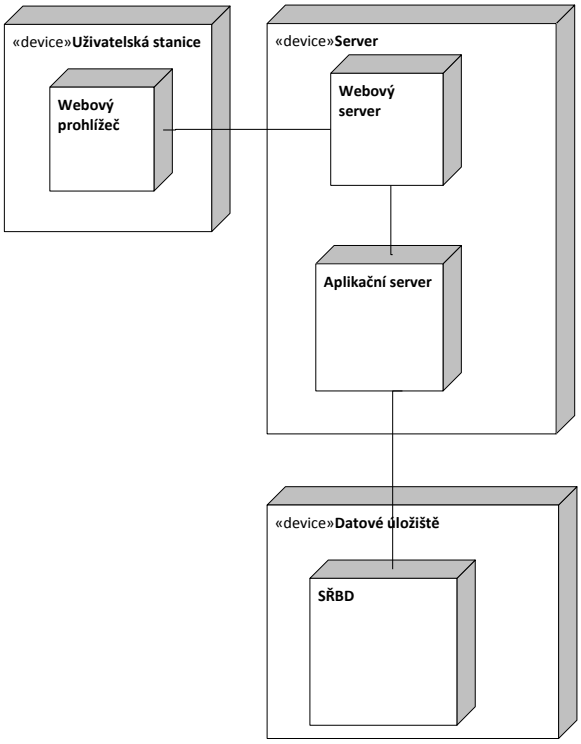
Aby se mohlo určit technické řešení, tak je nutné nejprve specifikovat bezpečnostní požadavky společností na budoucí informační systém. Předpokládejme, že systém bude mít dvě roviny. Veřejně přístupnou část a chráněnou část. V obou částech je nutné zajistit, aby uživatel nemohl provádět akce, na které nemá oprávnění. V případě veřejné části, uživatel nemusí prokazovat svou identitu a všechny data jsou přístupná pro kohokoliv. Hlavní problém, který se musí řešit ve veřejné části, je zajištění, aby uživatel nemohl využít systém ilegálním způsobem a provést tak akci, na kterou nemá oprávnění. Jsou to např. útoky typu cross-site scripting nebo sql injection.

U chráněné části je nutné zabezpečit všechny požadavky, které jsou u veřejné části. Dále jen určité osoby by měly mít přístup k informacím a k provedení operací v chráněné části. Je tedy nutné zajistit spolehlivý systém autentizace osob a autorizace provedení operací. Pokud osoba pracuje s chráněnou částí (tj. autorizovaná), je nutné zajistit, aby jiná osoba, která není autorizovaná, nemohla odposlouchávat komunikaci autorizované osoby, a tak získat chráněné informace. Toho lze zajistit šifrováním komunikačního kanálu (např. pomocí SSL – HTTPS).

Autorizace operace je často závislá na tom, zda uživatel disponuje určitým pověřením. To může být např. příslušnost do nějaké uživatelské skupiny. Někdy je pro autorizaci operace nutné provést výpočet, který zpracovává stavy i více objektů domény. Je vhodné, aby v cílových informačních systémech bylo možné specifikovat výše zmíněné metody autorizace. [7]

4 Implementační technologie

Tato kapitola se zabývá výběrem technologií, na kterých budou postaveny cílové informační systémy. Z předchozí kapitoly víme, že uživatel bude pro komunikaci se systémem využívat jako klientskou aplikaci webový prohlížeč. Webový prohlížeč zpravidla komunikuje s webovým serverem. Webový server zpracovává požadavek od prohlížeče. Vyhodnotí, o jaký požadavek se jedná a deleguje vyhodnocení požadavku aplikačnímu serveru, který je přiřazen pro ten typ požadavku. Aplikační server obsahuje vlastní logiku systému. Pro perzistenci objektů domény využívá aplikační server datové úložiště. Zmíněnou architekturu popisuje obrázek 3.



Obrázek 3: Diagram nasazení

4.1 Běhové prostředí

Zadání stanovuje, že by se systémy měly postavit na platformě. Net Framework od společnosti Microsoft. Protože IS budou v podobě webových aplikací využijeme technologii ASP.NET. ASP.NET, zjednodušeně řečeno, funguje jako aplikační server pro cílové systémy. Zpravidla běží jako ISAPI modul IIS (webový server).

ASP.NET má v této době dva oficiální frameworky pro tvorbu webových aplikací. A to ASP.NET Web Forms a ASP.NET MVC.

4.1.1 ASP.NET Web Forms

Framework ASP NET Web Forms byl vytvořen v roce 2002 a dodáván jakou součást .NET Framework. Jeho zaměřením bylo poskytnout stavové prostředí nad bezstavovém protokolu HTTP a umožnit podobný druh vývoje aplikací jako u klasických Windows aplikací. Stavovost se dosáhla ukládáním serializovaného stavu prvků stránky ve formě hidden pole – tzv. ViewState, případně využitím session state – data jsou ukládána na serveru a požadavky jsou odlišeny jedinečným

identifikátorem (cookie, nebo součást url). Podobnosti vývoje s Windows aplikacemi bylo docíleno především použitím event modelu spolu s hierarchií ovládacích prvků na stránce, které jsou protějškem ovládacích prvků ve Windows (button, label, drop down list).

Výhody:

- Snadný přechod od vývoje Windows aplikací
- ViewState – uchovávání stavu, event model
- Podpora vizuálního návrhu stránky ve Visual Studiu
- Velký počet knihoven ovládacích prvků

Nevýhody:

- ViewState – jeho velikost, přenáší se při každém požadavku na server
- Menší ovlivnitelnost generovaného HTML kódu
- Falešný pocit oddělení zodpovědností (SoC – separation of concerns) – míchání prezentace a logiky aplikace v code behind
- Nesnadný unit testing

4.1.2 ASP.NET MVC

První verze frameworku ASP.NET MVC byla vytvořena v roce 2009. V současné době je ve vývoji jeho třetí verze. Oproti Web Forms se ASP.NET MVC nesnaží abstrahovat webovou aplikaci jakou Windows aplikaci. Místo bojování s bezstavovostí http protokolu se s ní snaží spolupracovat. Využívá architektonický vzor MVC (model view controller). Dosahuje tím většího oddělení zodpovědností a snadnějších unit testů. Samotný framework je navržen tak, aby se jeho jednotlivé součásti daly nahradit za jiné, které lépe vyhovují dané aplikaci.

Výhody:

- Lepší kontrola nad generovaným HTML kódem
- Čistší HTML kód
- Lepší oddělení zodpovědností
- Snadnější unit testing
- Nemá ViewState
- Snadnější integrace s frameworky jako je jQuery

Nevýhody:

- Těžší přechod od vývoje Windows aplikací (nemá event model)
- Nemá ViewState
- Menší počet knihoven ovládacích prvků

[8] [9] [10]

4.1.3 Zhodnocení běhových prostředí

Zopakujme, že cílem je generovat systémy, které budou poskytovat moderní uživatelské rozhraní a budou snadno udržovatelné. V obou dvou frameworcích se dokáží implementovat podobně funkční aplikace. Moderní aplikace používají pro lepší uživatelský zážitek pro některé pomocné funkce technologii AJAX. Využití AJAX ve web forms aplikací přináší mnoho nástrah (velikost view state, délka trvání partial requestů, problematické znovuvytvoření stromu ovládacích prvků na straně serveru, při modifikace na straně klienta). Zkušenosti s vývojem Windows aplikací jsou nepodstatné (jiná situace než při vzniku Web Forms). Stavovost (ViewState) bývá výhodou i nevýhodou.

Výhod ASP.NET Web Forms nevyužijeme a v případě využití AJAX nebo jednotkového testování nás framework limituje. Při využití ASP.NET MVC získáme navíc možnost lepší kontroly nad generovaným HTML kódem, což vede k možnosti tvorby optimalizovanějších aplikací. Dále lépe odděluje zodpovědnosti prvků systému, což vede k lepší udržovatelnosti. Pro implementaci je tedy vhodnější ASP.NET MVC.

4.2 Perzistence dat

Podnikové informační systémy potřebují pracovat s perzistentními daty. Proto je nutné zvolit způsob perzistence dat. V současné době se pro informační systémy nejčastěji používají relační databáze s případnými post relačními rozšířeními. Někdy se též využívají, objektové nebo key/value databáze. Z pohledu implementovaných systémů nás zajímá, zda je možné do databáze uložit datové typy v systému používané. Typy specifikované v oddílu Datové typy dokáže ukládat většina databází. Tedy výběr databáze záleží na zákaznikových preferencích, zavedených uživatelských typech a na případném použitém řešení pro mapování dat mezi doménovou vrstvou a databází. Např. v případě relačních databází jde o ORM (object relational mapping) nástroj. Uvažujme, že využijeme nejrozšířenější řešení, tedy využijeme relační databázi. Je vhodné použít i ORM nástroj, protože psaní vlastního ORM řešení by bylo příliš složité a existující nástroje jsou již dostatečně vyvinuté a vhodné k obecnému řešení. Nejrozšířenější ORM nástroje pro .NET platformu jsou nHibernate a Entity Framework. Z dlouhodobého hlediska je vhodné, aby doménová vrstva nebyla závislá na použitém ORM nástroji. ORM nástroj tedy musí podporovat perzistenci POCO (plain old clr object) objektů. Tomuto požadavku vyhovují oba systémy. Nhibernate je starší a vyspělejší produkt, proto by se k němu mohlo v tomto případě přiklonit.

4.3 Systémy pro generování zdrojových kódů

V současné době je na trhu dostupných několik řešení pro generování zdrojových kódů. Níže je uveden seznam nejčastějších se zaměřením na .NET.

Rad Software NextGeneration

- Zdroj metadat: databáze
- Generování n-vrstvé aplikace
- Šablonově zaměřený
- Editovatelné šablony, rozšiřitelný o pluginy
- Komerční

MyGeneration

- Zdroj metadat: databáze, volitelný plugin (MyMeta Plugin)
- Metadata reflektují strukturu databáze
- Šablonově zaměřený
- Rozsáhlá databáze šablon
- Podpora 12 databází
- Freeware

Smart Code Generator (Asp.Net)

- Zdroj metadat: databáze
- Šablonově zaměřený, šablony v podobě .ascx control
- Podpora mssql, mysql, oracle
- Freeware

CodeSmith Generator

- Zdroj metadat: databáze, xml, xsd
- Rozšiřitelná meta^{2}data
- Šablonově zaměřený, šablony se syntaxí podobnou ASP.NET
- Možnost spouštění z příkazové řádky
- Integrace do Visual Studia
- Komerční

Text Template Transformation Toolkit (T4)

- Zdroj metadat: funkcionalita .NET Framework
- Možnost spouštění z příkazové řádky
- Šablonově zaměřený, šablony se syntaxí podobnou ASP.NET
- Integrace do Visual Studiu
- Součástí Visual Studia

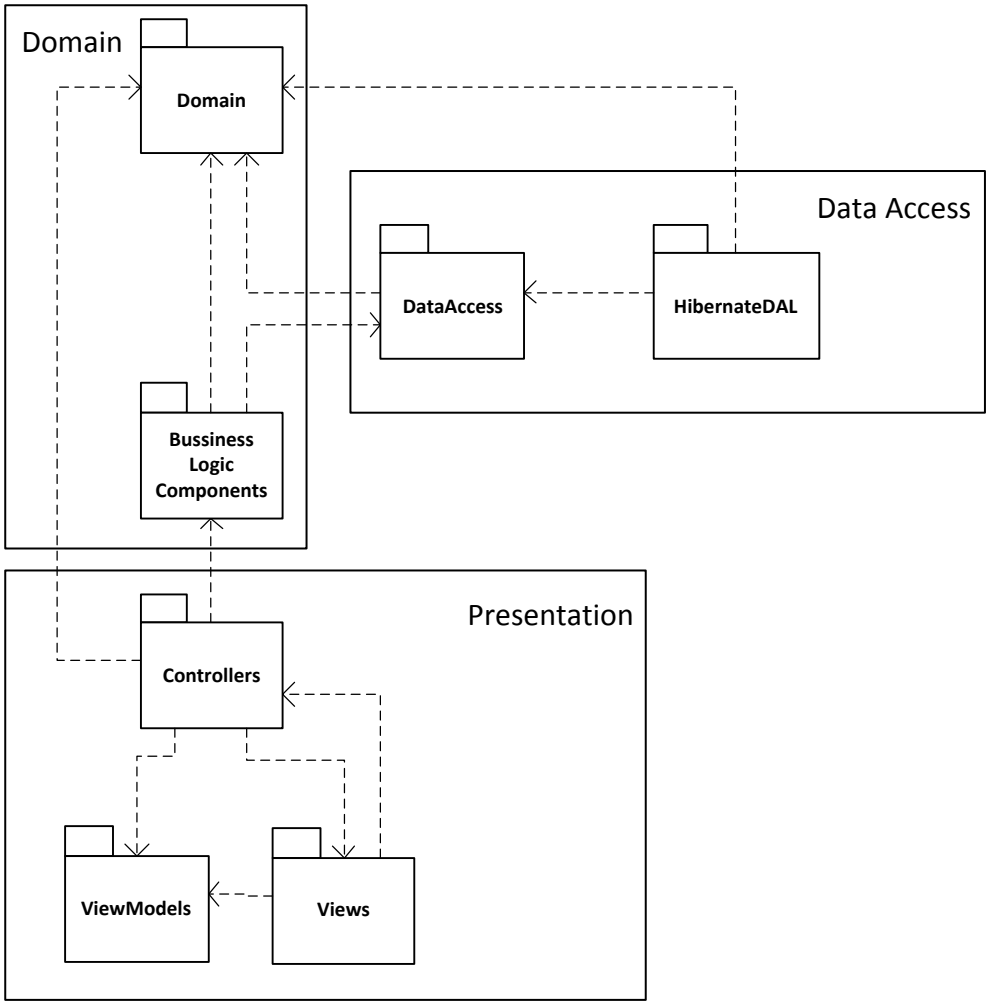
Požadavkům definovaným v kapitole Analýza a specifikace společných prvků informačních systémů vyhovují CodeSmith Generator a T4. Ostatní nevyhovují především z nemožnosti specifikovat vlastní meta^{2}data. T4 díky možnosti využít veškeré funkcionality .NET Framework nabízí oproti CodeSmith mnohem širší možnosti specifikace a práci s vlastními metadaty a meta^{2}daty. Dále integrace do vlastního nástroje je u CodeSmith licenčně závislá. Problémem T4 je, že využívá běhové prostředí Visual Studia. To nepřináší problém, pokud se systémy generují jen z něj. Ale kdyby se vyskytl požadavek spouštět generátor i mimo Visual studio, tak by se T4 nemusel dát použít s ohledem na nutnost přítomnosti instalace Visual Studia. Implementace vlastního generátor za využití volně dostupných komponent tyto problémy odstraňuje.

5 Návrh generovaných informačních systémů

Tato kapitola se zabývá návrhem generovaných informačních systémů dle požadavků specifikovaných v kapitole Analýza a specifikace společných prvků informačních systémů. Návrh představuje doporučení, jaké soubory s jakým obsahem by měl generátor generovat.

5.1 Architektura

Při návrhu architektury se musí především rozhodnout následující problémy: reprezentace modelu domény, perzistence domény, způsob komunikace s uživatelem.



Obrázek 4: Závislosti komponent na sobě

Obrázek 4 ukazuje závislosti jednotlivých komponent návrhu na sobě. Komponenty jsou zařazeny do logických balíků, vyjadřující, zda se jedná o prvky domény, části pro přístup k datům, nebo o uživatelské rozhraní. V implementaci často balíky odpovídají projektům v řešení systému. Výstupem projektu je např. knihovna, konzolová aplikace, nebo webová aplikace. V obrázku 4 je vidět, že balíky

domény a přístupu k datům jsou na sobě vzájemně závislé. V implementaci je nutné je buď sloučit do jednoho projektu, nebo upravit tak, aby na sobě nebyly závislé. Jejich rozdělení by zavedlo zbytečně moc projektů. Udržovatelnost systému by to moc neovlivnilo. Při ponechání v jednom projektu se musí dát pozor, aby se nezaváděly nové, zbytečné vazby. Obě varianty nebrání rozšíření systému např. o jiný způsob persistence. Většinu logiky v datové vrstvě budou obsluhovat knihovny třetích stran (zmíněno dále). Důsledkem je, že námi implementovaná část logiky datové vrstvy nebude příliš rozsáhlá. Díky tomu lze využít možnosti sloučení projektů do jednoho. Výsledné řešení informačního systému se bude skládat ze dvou projektů: z projektu business logiky (zahrnující část přístup k datům) a projektu webové aplikace představující uživatelské rozhraní. Toto rozdělení umožňuje v budoucnu vytvořit jiné uživatelské rozhraní, nebo část business logiky využít pro jiné účely.

5.2 Doména

Pro návrh implementace reprezentace domény systému lze využít návrhové vzory: Transaction Script, Domain Model, Table Module. [11]

Transaction Script

Organizuje doménovou logiku do procedur, kde každá procedura zpracovává jednu žádost od prezentační vrstvy. Výhodou je jednoduchost. Je vhodný pro systémy, které neobsahují složitou doménovou logiku. V případě nárůstu logiky a počtu procedur může nastávat problém s duplikováním kódu.

Table Module

Jedna instance table module zpracovává všechnu doménovou logiku pro všechny řádky databázové tabulky nebo pohledu. Reflektuje strukturu databáze: jedna objekt odpovídá jedné databázové tabulce pohledu či dotazu. Výhodou tohoto přístupu je snadná spolupráce s databází. Tento přístup není příliš vhodný v případě komplexní doménové logiky, protože se nemůže využít výhod přímých vazeb mezi instancemi nebo polymorfismu.

Domain model

Představuje objektový model domény, který představuje jak chování, tak data. Objektově orientovaný doménový model často připomíná strukturu databáze, na rozdíl od ní může využívat složité atributy, komplexní asociace či dědičnost. Je vhodný pro systémy, které obsahují složitou doménovou logiku. Tyto výhody jsou vykoupeny složitým návrhem a spoluprací s databází.

Zhodnocení

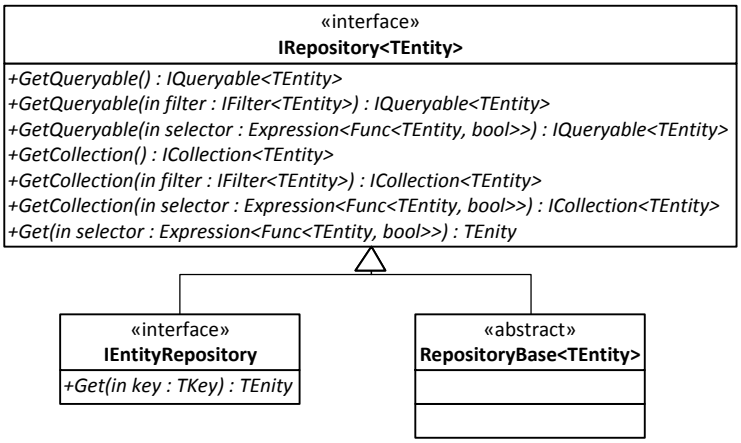
Hlavním požadavkem, je aby systémy umožňovaly CRUD operace. Toho lze dosáhnout využitím všech výše uvedených principů. Dodatečným, ale důležitým, požadavkem je, aby bylo možno doprogramovat vlastní logiku do systému ručně. Nelze předem vědět, jak složité logiky budou budoucí systémy využívat. Mělo by se tedy předpokládat i složitá logika. Tomu vyhovuje domain model. Nevýhody doménového modelu lze vykoupit využitím generování kódu a ORM nástroje, tedy pro kontext práce je velmi vhodný.

5.3 Datová vrstva

Účelem datové vrstvy je provádět perzistenci, rušení objektů domény do databáze. Při využití domain model, jako vzoru pro návrh doménové vrstvy. Lze využít pro perzistenci vzory Active record, Data mapper či jiné. Rozdíl mezi těmito vzory je vytváření zodpovědnosti, kdo má na starost perzistenci. Active record dává zodpovědnost samotnému objektu domény. Objekt domény je tedy většinou vybaven metodami Insert, Save, Delete a statickou Load. Oproti tomu data mapper obstarává veškerou práci s databází a protože využívá vzoru Mapper, nechává objekty domény bez vazby na databázi – tedy jsou ve stavu „Persistent ingnorance“. Toto chování je výhodné z hlediska testování a udržovatelnosti.

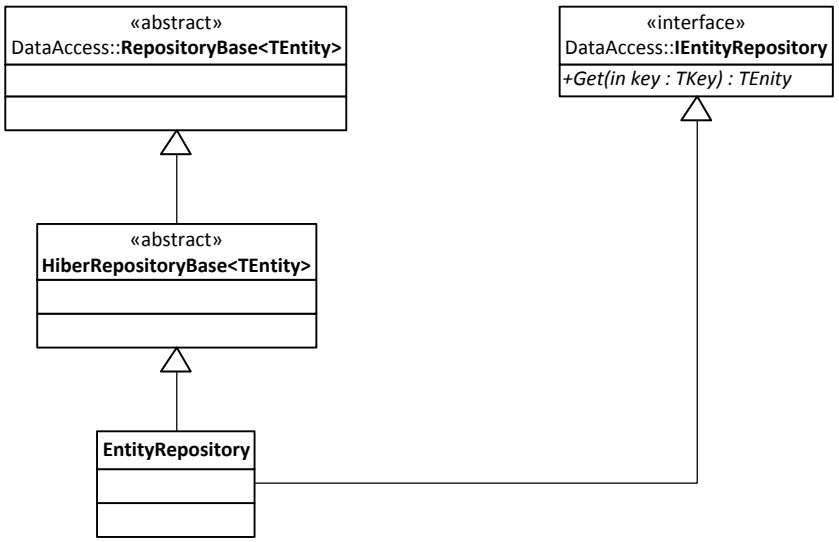
Při využití komplexní logiky a data mapperu, je výhodné využít další abstraktní vrstvy, kde se soustřeďuje kód pro dotazování. Pro tyto účely slouží návrhový vzor repository (repositář). Repositář slouží jako prostředník mezi doménovou logikou a data mapperem. Poskytuje jednotné rozhraní pro dotazování, nezávisle na použitém datovém zdroji. Díky tomu programátor nemusí přemýšlet v konkrétním dotazovacím jazyce.

V systému bude existovat pro každou nezávislou entitu repositář. Demonstruje to obrázek 5. Rozhraní `IRepository<TEntity>` představuje generické rozhraní, které musí každý repositář pro daný typ entity implementovat. Pro každý typ nezávislé entity bude existovat jeho specializace, v diagramu je vyjádřena rozhraním `IEntityRepository`. Specializované rozhraní by mělo obsahovat metody specifické pro daný typ entity. Abstraktní třída `RepositoryBase<TEntity>` slouží jako bázová třída pro konkrétní implementace repositářů. Její využití je však volitelné. Tedy doménová logika bude pracovat pouze s objekty implementující rozhraní repositářů, nebude závislá na konkrétní implementaci.



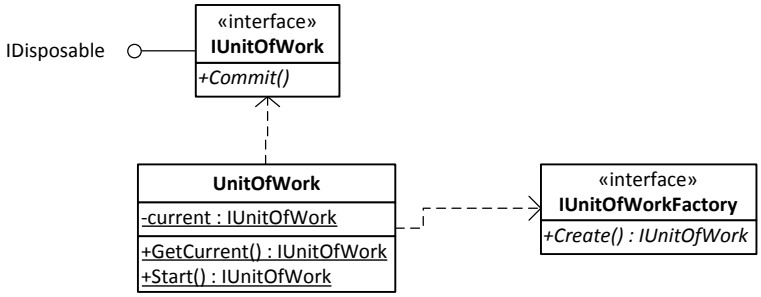
Obrázek 5: Obecný repository

Konkrétní implementaci repositářů demonstruje obrázek 6. Obrázek demonstruje vytváření repositářů pro nHibernate. Abstraktní třída `HibeRepositoryBase<TEntity>` zajišťuje společné chování repositářů využívající nHibernate. Z ní pak konkrétní repositáře dědí a implementují rozhraní `IEntityRepository` popsané výše.



Obrázek 6: Příklad konkrétní implementace repository

Odstíněním kontrolní vrstvy (část Controllers) od datového zdroje docílíme využitím návrhového vzoru Unit Of Work. Unit of Work slouží k uchovávání přehledu o změnách stavu objektu vůči datovému zdroji a poskytuje možnost provedené změny uložit. Většina ORM nástrojů tuto funkcionalitu implementuje např. ObjectContext v Entity Framework nebo Session v nHibernate. Schéma vzoru zobrazuje obrázek 7. Pro využití v systému stačí jen implementovat vlastní třídu implementující rozhraní `IUnitOfWork`, což většinou bude wrapper kolem Unit Of Work daného ORM nástroje a tovární třídu, která bude vytvářet její instance.



Obrázek 7: Schéma Unit of Work

5.4 Dependency injection

Dependency injection (DI) je speciální případ principu Inversion of Control. Zajišťuje, že při tvorbě instance objektu, jsou mu závislé komponenty dodané třetí stranou. Tedy třetí strana rozhoduje jaký přesný typ komponenty (určitá implementace rozhraní, nebo potomek třídy) bude vytvářený objekt využívat. Tímto se zavádí volnější vazba mezi komponentami systému. Zmíněnou třetí stranu představuje tzv. dependancy injection container (kontejner). Tímto způsobem může kontejner zajistit transparentní sestavení stromu komponent. Kontejner je nutné nakonfigurovat, aby věděl, jaké konkrétní typy komponent má dodávat. Konfiguraci řeší různé implementace kontejnerů různě [12]. V aplikaci se využije kontejner nInject. Jeho výhodou je, že se dá snadno konfigurovat ze zdrojového kódu místo často využívaného XML. To umožní typovou kontrolu, tedy poskytuje částečnou kontrolu správnosti nastavení již při kompilaci. [13]

5.5 Řadič

Řadič (Controller) spadá do aplikační vrstvy – někdy je řazen do prezentační. Z pohledu aplikace představuje vstupní bod – požadavky na webový server jsou pomocí frameworku ASP.NET MVC transformovány na volání metod řadičů. Úkolem řadičů je tedy zpracovávání požadavků, dle kterých volá provedení operací doménové logiky. Výsledky operací zpracovává do modelu vhodný pro pohledy tzv. view modelu. Nakonec vybírá pohled. Pohled a view model předává systému pro zpracování pohledů (view engine). Ten vytvoří výsledek ve formě html dokumentu. Kromě html dokumentů může mít řadič i jiné druhy výsledků. Jsou to např. JSON dokumenty či binární soubory.

Generované systémy budou obsahovat jeden řadič pro každý typ nezávislé entity. Řadič musí dědit ze třídy `Controller` (ASP.NET MVC framework). V systému nahradíme standardní ASP.NET MVC Controller Factory vlastní implementací, která nám umožní využít DI kontejner. [10]. Pro každou operaci na entitě bude řadič obsahovat jednu metodu pro metodu Get HTTP protokolu, a případně dodatečnou POST metodu, pokud se očekává odpověď od formuláře (např. u metody create, update, delete).

5.6 View Model

View Model slouží jako nosič dat, které jsou dodávaná pohledu. Ke každému pohledu bude existovat view model. Pro každý editor v pohledu by měl view model obsahovat vlastnost obohacenou o příslušné atributy (definované v konceptuálním modelu), díky kterým se v pohledu bude moci rozhodnou jakým způsobem vykreslit editory, jaké použít formátování pro vykreslení hodnot či případné validátory. Díky přítomnosti validačních atributů lze provádět client-side validaci formuláře.

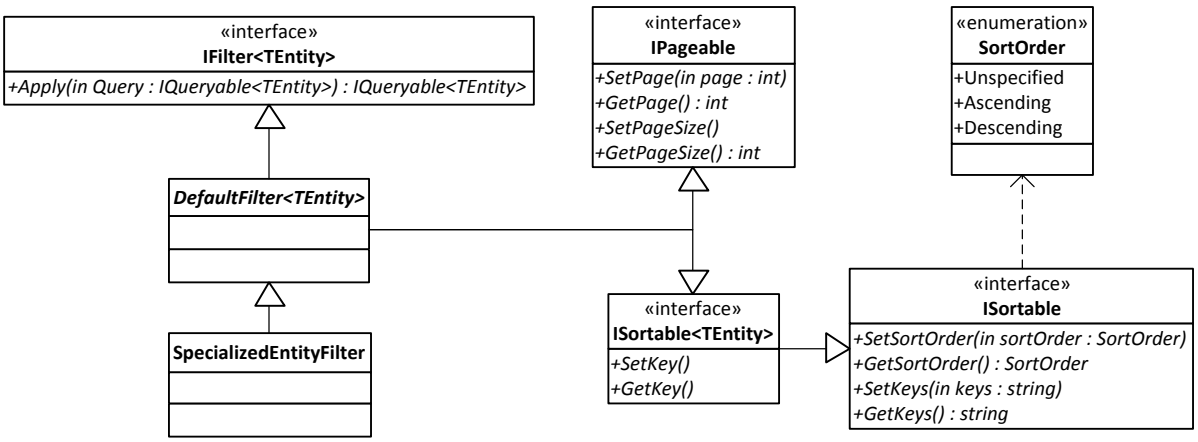
5.7 Pohled

Pohled (View) představuje uživatelský výstup aplikace – webovou stránku. Pracuje s daty (view modelem) získanými od řadiče. Pohled by neměl obsahovat žádnou doménovou logiku. Porušování tohoto pravidla vede ke špatné udržovatelnosti aplikace. Tedy pohled přestavuje šablonu, do které se dosadí hodnoty view modelu.

Aplikace bude obsahovat proces s uživatelským vstupem/výstupem alespoň jeden pohled. Dále musí obsahovat částečné pohledy pro dílčí prvky – např. řádek pro editaci závislé entity.

5.8 Vyhledávání

Jedním z požadavků definovaných v analýze bylo, aby systém umožňoval vyhledávání, řazení a stránkování. K tomu byly vytvořeny třídy s rozhraním `IFilter<TEntity>`. Jejich účelem je modifikovat dotaz reprezentovaný objektem implementující `IQueryable<T>` dle jejich stavu. Třída `DefaultFilter<T>` slouží jako bázová třída pro konkrétní implementace filtrů. Měla by obsahovat implementaci řazení a stránkování. Konkrétní implementace filtrů by měly obsahovat podporu pro filtrování (vyhledávání).



Obrázek 8: Subsystém pro filtrování, řazení, stránkování

Z uživatelského hlediska by měl pohled obsahovat editor pro všechny pole filtru. Rozhraní těchto editorů bude defaultně skryto a zobrazeno jako modální okno pomocí JavaScriptu. Při řazení a stránkování by se měl zachovávat předchozí výsledek filtrování. Toho lze dosáhnout uchováním si předchozího obsahu filtru v session.

5.9 Lokalizace

Každá textová hodnota, která není uložena v databázi, musí být lokalizovatelná. V .Net Framework lze využít zabudovaného systému. Všechny texty aplikace budou umístěny v tzv. .resx souborech. Resx soubor je xml soubor, který obsahuje zdroje v podobě klíč-hodnota-komentář. Při kompilaci lze využít nástroje, který dle těchto souborů připraví třídu obsahující vlastnosti reflektující klíče v .resx souboru. Díky tomu můžeme v systému přistupovat k řetězcům přímo a o lokalizaci se stará `ResourceManager`, který je součástí vygenerované třídy. Je vhodné texty dělit do více .resx souborů, dle jejich využití. Např. jeden soubor na operace s určitou entitou.

5.10 Procesy

V této podkapitole se zaměříme podrobněji na jednotlivé procesy – zejména CRUD a jak budou reprezentovány v aplikaci.

Proces představuje uzavřenou posloupnost operací doménové logiky. Tedy veškeré změny v systému se provádějí přes procesy. Proces je v systému reprezentován třídou, která se skládá ze vstupu, průběhu a výstupu. Někdo by mohl namítnout, že pro určité jednoduché operace by bylo jednodušší použít např. jen statickou metodu v třídě obsahující podobné operace. Případně pro prosté získání entity dle jejího klíče by se mohla přímo volat metoda repozitáře. Takové zjednodušení mohou urychlit vývoj – ušetří se pár řádků zdrojového kódu, popř. se nemusí vytvářet další třídy. V případě kdy voláme repozitáře přímo z řadiče dochází k zbytečným závislostem na nižší architektonické vrstvě. Volání statické metody je určité zjednodušení, často však vede k tomu, že i složitější procesy se začnou implementovat ve stejné třídě, poté začnou sdílet určité privátní metody a nakonec to může vést k tomu, že programátor ztrácí přehled o tom, které procesy která metoda ovlivňuje. Obecně zavedení těchto nejednotností zhoršuje udržitelnost kódu. Proto se těmto zdánlivým zjednodušením chceme vyhnout. Tedy pokud mají určité procesy sdílet část doménové logiky, měla by tato část být implementována jako zvláštní proces.

5.10.1 CRUD

CRUD operace představují základní procesy v systému. Pro každou entitu je vytvořen proces Create, Read, Update, Delete a List.

Create

Vstupem procesu je nově vytvořená entita naplněná daty. Jeho úlohou je entitu perzistovat.

Read

Vstupem procesu je klíč entity. Jeho úlohou je získat entitu s daným klíčem z repozitáře. Výstupem je nalezená entita.

Update

Úlohou procesu update je perzistovat změny provedené v existující entitě.

Delete

Úlohou procesu delete je smazat entitu z úložiště.

List

Proces List by se dal označit za jiný případ procesu Read. Jeho vstupem je objekt filtru pro daný typ entity. Výstupem je kolekce entit, která vyhovuje omezením obsažených ve filtru.

5.10.2 Moduly

Procesy svázané s určitou entitou jsou organizovány do modulů. Modul slouží jako zjednodušující prvek pro získání procesu.

5.10.3 Doménově specifické procesy

Doménově specifické procesy vytvářejí podstatu informačního systému. Jejich obsah není generován generátorem, ale musí ho napsat programátor ručně. Složitější procesy by měly být navrženy tak, aby se skládaly, když je to vhodné, z již existujících jednodušších. Zamezí se tak zbytečnému opakování kódu a aplikace obsahuje jednotné dílčí chování.

6 Analýza a návrh generátoru

Tato kapitola se zabývá návrhem generátoru. Generátor je program, jehož účelem je vytvářet soubory s určitým obsahem – zpravidla zdrojové kódy dle dodaných informací. V případě generování informačních systémů je vstupem model systému a výstupem soubory představující informační systém.

6.1 Vstup

Generátor tedy pracuje s modelem systému, dle něj generuje patřičné soubory. Aby mohl model načíst a aby mu rozuměl, musí být definován model modelu systému – metamodel. Interně je model reprezentován jako instance tříd metamodelu. Vstupem generátoru by mohla být hierarchie takových objektů. Model nelze vytvářet přímo. Můžeme zvolit cestu, že generátoru přiřadíme uživatelské rozhraní, které nám umožní definovat model systému a tak vytvářet požadovanou paměťovou strukturu. Jinou možností je definovat určitý způsob uložení modelu a jeho izomorfní strukturu k metamodelu. Např. definovat jazyk, kterým by se takový model dal zapsat v textovém souboru. Zajímavou volbou je vytvořit databázovou strukturu izomorfní k metamodelu a k tomu uživatelské rozhraní, které by umožnilo model vytvářet, upravovat a ukládat. Generátor by pak potřeboval mít schopnost načíst model, který je uložen takovým způsobem. Tedy mít adaptér pro načtení modelu určitého typu.

6.2 Způsoby generování výstupu

Tato podkapitola se zabývá způsoby, kterými generátor mlže vytvářet svůj výstup. Obecně je generátor jen transformační funkce modelu na kolekci souborů. Tato transformace se skládá z dílčích úloh, které lze klasifikovat do několika typů: generování pomocí šablony, přímé vytváření výstupu, či kopírování existujících souborů [4].

Přímé vytváření výstupu je nejobecnější metodou. Generátor zpracovává model a výstup formuje přesně tak, jak daná úloha potřebuje. Do souboru zapisuje dle potřeby, nebo vše najednou na konci. Tento způsob má největší vyjadřovací sílu – co se potřebuje, to se vygeneruje. Nevýhodou je, že sestavování výstupu může být značně zdlouhavé – ve smyslu psaní kódu.

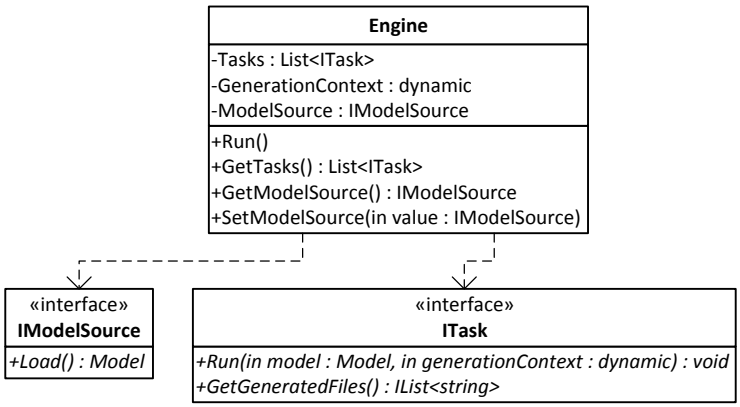
Pro úkoly, kde se provádí jen menší množství rozhodnutí, a pak se vypočítané hodnoty zapíší na určité místo v jinak statickém souboru, je často vhodnější využít šablonového nástroje. Šablona je textový soubor, který obsahuje statický text, ve kterém jsou vyznačeny části s určitou logikou. To může být jednoduché dosazení hodnoty, nebo i složitější operace jako je procházení kolekce a generování výstupu pro každý člen, či volání jiné šablony pro vygenerování dílčí části. Tento přístup je zvláště dobrý využít, kdy již máme příklad požadovaného výstupu. V takovém případě se v něm nahradí jen hodnoty specifické pro danou část systému (např. entitu). Celkově to vede k výraznému zvýšení rychlosti vývoje logiky generování dané úlohy.

Poslední zmíněnou možností je kopírování existujících souborů. Někdy při generování systému se nepotřebuje dosazovat do určitých potřebných souborů hodnoty specifické pro daný systém. Může se jednat např. o obrázky či kaskádové styly. V takovém případě je vhodné mít tyty soubory připravené a do nově generovaného řešení se jen zkopírují.

6.3 Návrh generátoru

Při návrhu generátoru se musí předpokládat, že požadavky na generované systémy se budou časem měnit. Také se musí předpokládat, že v budoucnu se může chtít načítat modely různými způsoby. Z těchto důvodů by se mělo jádro generátoru navrhnout tak, aby bylo pokud možno co nejméně závislé na způsobu zapsání modelu a na generovaném obsahu. Tento požadavek implikuje, že generátor musí být rozšiřitelný o nové způsoby načítání modelu a o specifikaci obsahu pro generování. Tohoto cíle se dosáhne tak, že se navrhne minimalistický generátor, který sám o sobě nic neumí a veškerá jeho funkcionalita je mu dodána při inicializaci.

Jádro generátoru bude tvořit třída Engine. Při inicializaci se objektu třídy `Engine` nastaví objekt implementující `IModelSource` a seznam objektů implementující `ITask`. To je znázorněné na diagramu tříd, který ukazuje obrázek 9. Objekt implementující rozhraní `IModelSource` zajišťuje načtení modelu. Objekty implementující rozhraní `ITask` představují úlohy, které se mají provést – vlastní logiku generování. Začátek generování je inicializován tak, že se zavolá metoda `Run` objektu třídy `Engine`. Metoda `Run` vyvolá načtení modelu informačního systému voláním metody `Load` u objektu `IModelSource`. Poté se sekvenčně volá metoda `Run` všech objektů `ITask`. Při volání se jim předává model systému. Tím dojde k provedení všech požadovaných úloh – např. vygenerování informačního systému. Díky rozhraní `IModelSource` můžeme vytvářet objekty, které jsou schopné načítat model informačního systému z různých míst a různých formátů. V návrhu se dále počítá s tím, že by někdy mohlo dojít k situaci, kdy různé úlohy by si chtěly mezi sebou předávat data. Např. při provedení první úlohy by se vytvořila jistá informace, kterou by chtěla následující úloha použít. K tomuto účelu byl `Engine` rozšířen o vlastnost `GenerationContext`. Ta je typu `dynamic` (konkrétně `ExpandoObject`), což umožňuje ji použít podobně jako slovník. Tento objekt je předáván každému volání úlohy. Tedy úloha ho může využít – pro čtení či zápis.



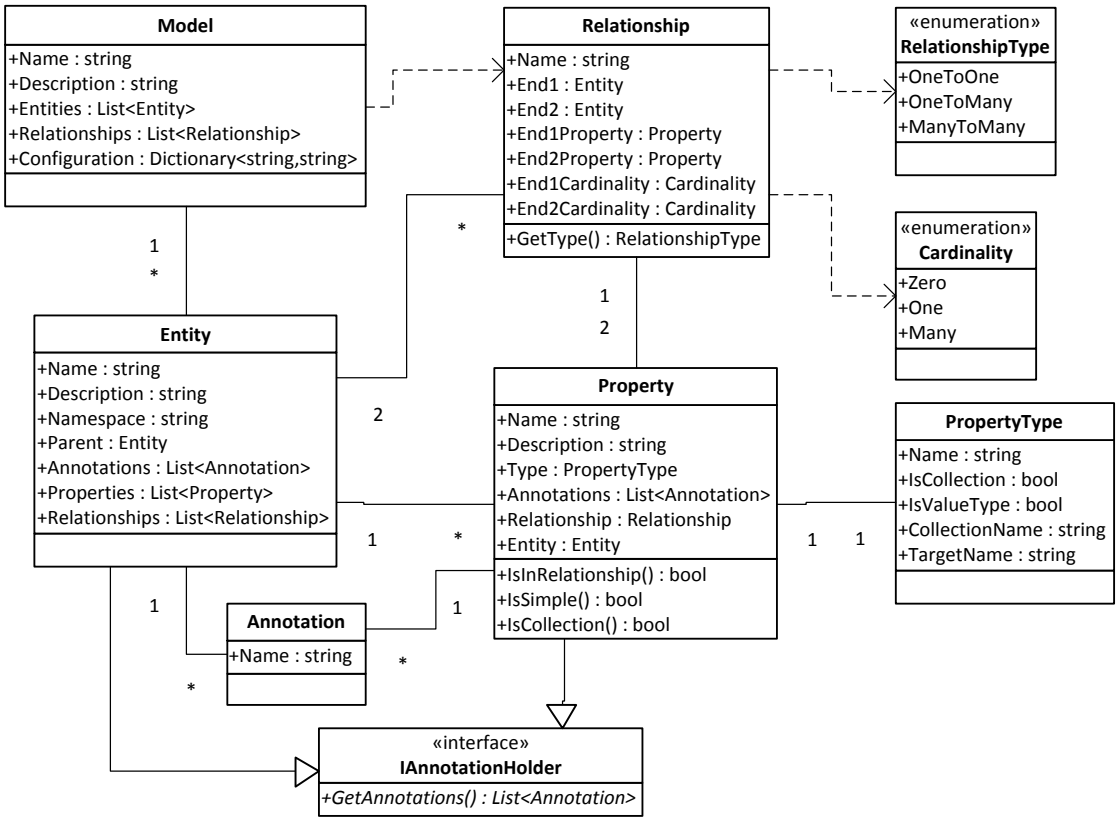
Obrázek 9: Architektura jádra generátoru

6.4 Metamodel

V kapitole 3 byly specifikovány požadavky na zápis konceptuálního schématu. Mělo by být umožněno reprezentovat všechny specifikované požadavky formou modelu. Proto musíme definovat metamodel informačního systému. Metamodel specifikuje strukturu modelu. Jinak řečeno specifikuje vyjadřovací schopnost, kterou může model nabývat. Schéma metamodelu je zobrazeno na obrázku 10.

Základem metamodelu je třída `Model`, jehož instance reprezentuje celý model systému. Slouží jako vstupní bod pro inspekci zbytku modelu. Obsahuje tři klíčové vlastnosti

- seznam entit – `Entities`
- seznam vztahů – `Relationships`
- slovník konfiguračních řetězců – `Configuration`



Obrázek 10: Schéma metamodelu

Entity

Základním stavebním prvkem modelu je entita. Každou entitu reprezentuje jeden objekt třídy `Entity`. Entita je identifikovaná jménem (`Name`). Atributy entity tvoří kolekce objektů typu `Property`. Všechny vztahy entity s jinými entitami jsou uloženy v kolekci `Relationships`. Pro specifikaci dodatečných informací o entitě slouží kolekce anotací – `Annotations`. Položka `parent` představuje entitu, na které je daná entita závislá. Nepřímo tedy definuje, že se jedná o závislou entitu.

Vztahy

Každý vztah mezi dvěma entitami je reprezentován objektem třídy `Relationship`. Jsou vněm zaznamenány informace o obou koncích vztahu. Každý konec je tvořen kardinalitou, entitou a vlastností entity, která daný vztah tvoří. Ve vztahu není nijak definován směr, proto jsou oba konce stejně významné.

Vlastnosti

Atributy entity jsou reprezentovány vlastnostmi – objekty třídy `Property`. Primárními vlastnostmi vlastnosti jsou její jméno, objekt typu vlastnosti a vztah. Dále jsou vlastnosti, podobně jako entity,

doplněny kolekcí anotací. Poslední významnou vlastností je odkaz na entitu, která danou vlastnost obsahuje.

Typy vlastností

Každá vlastnost má typ, který je reprezentován objektem třídy `PropertyType`. Typ je reprezentován jménem. Pokud se jedná o kolekci, tak dodatečně nastavuje typ kolekce a jméno typu, která kolekci nese. Poslední parametrem je údaj zda se jedná o hodnotový nebo referenční typ – to může být užitečné při rozhodování jak s vlastností daného typu nakládat.

Anotace

Anotace slouží jako základní prostředek pro specifikování dodatečných informací do modelu. Dalo by se to přirovnat např. ke stereotypům v UML. Na rozdíl od UML, kde stereotyp je reprezentován jen svým jménem jsou anotace určené i k nesení dalších, upřesňujících dat. Dosáhnout by se toho mělo zděděním od třídy Annotation. Ve zděděné třídě se dodají vlastnosti pro uchování informací. To vede k zvýšeným požadavkům na objekty načítající model z určitého zdroje. Při načítání modelu se musí vytvořit instance daného atributu, tedy načítající objekt musí daný typ znát. Což vede k tomu, že by objekty pro načítání modelu měly pracovat buď s jistou, danou množinou anotací, nebo obsahovat mechanismus pro rozšíření svých schopností.

Anotace nám dávají do rukou velmi silný nástroj, jak definovat informace pro řízení generování, nastavování vztahů, určování vzhledu formulářů, lokalizaci ap.

Dodatečná konfigurace

Objekt typu Model je vybaven slovníkem pro uložení dodatečných textových informací. Tato funkcionalita je zvláště vhodná pro definici jména projektů, jmenných prostorů, či jiných položek nutných pro vygenerování informačního systému.

Rozšíření metamodelu

Metamodel lze rozšířit tak, že se zdědí z určitých jeho tříd. Poté ale musí s tímto počítat i objekt pro načítání modelu. Důvodem pro rozšíření metamodelu může být například specifikace částí systémů, či procesů, které stávající metamodel nedokáže pojmout. Stávající úlohy stále budou schopné pracovat s nerozšířeným modelem i po jeho rozšíření, protože základní rozhraní zůstane nezměněno.

6.5 Zápis a načítání modelu

Při vývoji informačního systému se model může zapisovat různými způsoby. Výběr způsobu zápisu je závislý především na metodice vývoje systému. Ideální by bylo, kdyby analytik systému mohl model zapisovat ve srozumitelném tvaru, už během konzultace, nebo hned po ní. K tomu by potřeboval intuitivní nástroj, který by ho nezdržoval v práci – ideálně program s grafickým uživatelským rozhraním. Takový program by měl nabízet postupnou definici modelu v pořadí a přesnosti definice jeho částí dle procesu vývoje. Jednou z možností, kterou lze použít jsou různé UML nástroje. Tyto nástroje jsou výborné pro prezentaci principů či konceptů – využívají k tomu celou řadu diagramů. Problém v použitelnosti nastává, když chceme vytvořené diagramy využít pro vytvoření vlastního systému pomocí generování kódu. Soubory s digramy nemůžeme využít, protože zpravidla používají proprietární formát souborů, které by bylo téměř nemožné zpracovat, nebo by se nevyplatil vývoj speciálního komponenty pro jejich načtení. Rozumnou volbou je export do známého formátu. Pro tyto účely byl vyvinut formát XMI – XML Metadata Interchange. Tento formát byl vyvinut skupinou OMG – Object Management Group jako formát souborů pro přenos metadat. Umožňuje přenos UML

modelů. Jevil by se tedy jako vhodný kandidát a v mnoha případech by byl dostatečným řešením. Avšak při zpracovávání UML diagramů může nastat problém s rozpoznáním sémantiky informací. Informace z diagramů jako je stavový diagram, sekvenční digram, diagram nasazení jsou v kontextu této práce zbytečné. Vyhodnocení jejich sémantiky by byl komplikovaný, ne-li nemožný úkol a užitek by byl téměř nulový, protože máme architekturu informačního systému již pevně danou. Informace, které nás zajímají, jsou obsaženy především v diagramech statické struktury – jako je digram tříd. Pro definici struktury tříd je UML výborný prostředek. Problém nastává, když třídu chceme chápat jako entitu, s ní spjaté operace a případnou reprezentaci v grafickém prostředí. Takové dodatečné informace je v UML problematické definovat. Největší překážkou tvoří právě UML nástroje. Definice těchto údajů je zpravidla zdlouhavá, protože nepatří mezi často použitou funkcionalitu. U některých nástrojů nemusí být vůbec možná. Nabízí se možnost využít UML nástroj jen pro základní definici modelu a zbytek dodefinovat ručně přímo v XMI souboru. Tato varianta se nejeví jako vhodná, protože vyžaduje dobrou znalost XMI formátu a navíc úprava rozsáhlého modelu by mohla být zdlouhavá a nepřehledná.

Vhodnější variantou, se jeví vytvořit program pro specifikaci modelu přímo na míru požadavkům vývojového procesu. Mohla by to být webová, či nativní „okenní“ aplikace. Uživatelské rozhraní by bylo vytvořeno tak, aby podporovalo přímo proces analýzy. Zvláštní důraz by měl být kladen na uživatelskou přívětivost definice anotací. Toho by mohlo být docíleno organizací anotací do kategorií dle jejich významu, případně vytvoření sady šablon anotací pro nejběžnější účely. Aplikace by musela ukládat model do určitého úložiště. Vhodným úložištěm může být relační databáze, jejíž databázové schéma by odpovídalo schématu metamodelu. Při použití relační databáze, jako úložiště, je pak velmi snadné a rychlé naprogramovat komponentu (`IModelSource`) pro načítání tohoto modelu do generátoru.

Nevýhodou předchozí varianty je její časová náročnost na vývoj. Hlavní přínos by byl až v praxi. V kontextu této práce je možné využít implementačně jednoduššího způsobu zápisu modelu, než je webové rozhraní a zároveň intuitivnějšího než je použití XMI. Při uvažování textové specifikace je z uživatelského hlediska nejvhodnější jazyk vytvořený přímo na míru. Vyžadovalo by to specifikaci jeho syntaxe a vytvoření parseru pro možnost dalšího zpracovávání.

Jinou možností je využít již existující jazyk, který by umožňoval definovat všechny potřebné části modelu. První kandidát je implementační jazyk generátoru – C#. Samotným využitím jiného jazyka pro zápis nezaniká nutnost psaní parseru. Je tedy nutné vyhnout se zkoumání zápisu modelu jako textového souboru a najít jinou možnost. Nabízí se dané zdrojové soubory s definicí modelu zkompilovat a vytvořenou knihovnu prozkoumat pomocí reflektivních nástrojů .Net Frameworku. Třídy ve jmenném prostoru `System.Reflection` umožňují získat všechny třídy v určité knihovně, zkoumat jejich vlastnosti, atributy či metody. Tato funkcionalita je plně postačující pro získání všech informací kterém by se mohly vyžadovat. Výhodou tohoto přístupu je, že pokud použijeme k modelu Visual Studio, které již slouží pro vývoj generátoru, získáme navíc zvýraznění syntaxe, kontrolu syntaxe a doplňování výrazů při psaní.

6.5.1 C# jako jazyk pro zápis modelu

Při použití C#, nebo jiného jazyka s podporou .Net Framework, by se mohl model definovat přímo pomocí vytváření objektové hierarchie metamodelu. Pak by pro získání modelu stačilo ve zdroji modelu (`IModelSource`) volat metodu, která takto vytvořený model navrátí. Takový přístup dokáže maximálně využít vyjadřovací schopnost metamodelu. Tento přístup by se dal označit spíše za imperativní – model je vytvořen po vykonání příkazu. A tedy samotná vhodnost pro použití záleží spíše na subjektivním pocitu uživatele. Někteří uživatelé by radši ocenili více deklarativní přístup – např. pomocí využití jazykových konstruktů a následné introspekci při načítaní modelu. Pro takový

případ se musí najít prostředky jazyka C#, které odpovídají částem metamodelu. Zbytek podkapitoly analyzují takový způsob zápisu.

Základem jazyka C# jsou třídy. Třídy mohou obsahovat položky, vlastnosti, metody. Každá vlastnost, položka, třída, metoda může být doplněna o atributy.

Zápis entity

Sémanticky entitě odpovídá třída nebo strukturu. Třída se jeví jako vhodnější, protože může obsahovat metody, které by šlo případně využít při rozšíření metamodelu. Tedy každá třída by odpovídala entitě. Takové prohlášení může vést k potenciálním problémům při rozšíření metamodelu. Třídy by nešly použít na nic jiného. Musí se tedy vytvořit označení, které by deklarovalo, které třídy jsou entity. Takovým označení je atribut `Entity`. Potom jméno třídy označuje jméno entity. Položky `Description` a `Namespace` se definují v atributu `Entity`. Definici, zda se jedná o závislou entitu, představuje výskyt atributu `DependantEntity`, který má jako parametr třídu, představující entitu, na které je entita závislá.

Vlastnosti

Vlastnosti entity odpovídají položkám, nebo vlastnostem třídy. Mohly by se zvolit obě varianty, ale dle shody jmen je vhodnější, že vlastnostem bude odpovídat vlastnost. Podobně jako u tříd bude každá vlastnost označena atributem `Property`. Tím je zajištěna možnost využít vlastnosti v budoucnu pro jiné účely.

Typy vlastností

V jazyce C# se typ vlastnosti zapisuje v definici vlastnosti před jejím jménem. Tohoto se dá využít. Musí se jen zajistit, aby pro všechny typy podporované generátorem existoval ekvivalentní typ, který by se v definici použil. Ekvivalencí je myšleno, že daný typ má shodné jméno a zda se jedná o hodnotový typ, nebo o kolekci.

Vztahy

Pro definici vztahu se nabízí několik možností. Mohla by se vytvořit vlastnost, kde jméno by představovalo název vztahu a ostatní položky (názvy entit, vlastností, kardinality) by byly definovány jako parametry atributu `Relatioship`. Taková definice by musela být někde umístěna. Nabízí se možnost umístit ji do některé třídy entity, které jsou součástí vztahu. To by mohlo vést k interpretaci, že jedna z entit má větší význam, než druhá, tedy potenciálně nevhodný záměr. Lepší možností by tedy bylo definici vztahu umístit mimo obě třídy (entity) např. do speciální třídy, která by sloužila jako místo pro definici všech vztahů.

Jinou možností je využít faktu, že vztah se týká dvou vlastností určitých entit. Vztah můžeme definovat tak, že k definici obou vlastností vtahu v entitě přiřadíme atribut `Relationship`, jehož parametr `Name` bude obsahovat jméno vztahu. Tedy entity a vlastnosti jsou získány samotným umístěním atributů `Relationship`. Jméno vztahu vytváří párování obou míst definice. Kardinalita je buď odvozena od datového typu vlastnosti. Pokud je typem kolekce, nemůže být kardinalitou `One` ani `Zero`. Pokud typem vlastnosti není kolekce, je kardinalita `One` nebo `Zero`. Implicitní nastavení záleží na způsobu načtení modelu, nebo pomocí atributu vlastnosti `Required`. Výhodou této možnosti definice vztahu je, že při prohlížení definice entity je vidět, které vlastnosti jsou součástí vztahů a tedy i všechny vztahy, které entita obsahuje. Zároveň je zápis kratší. Nevýhodou je nutnost specifikovat stejné jméno vztahu na dvou místech, což představuje potenciální místo pro zdroj chyb.

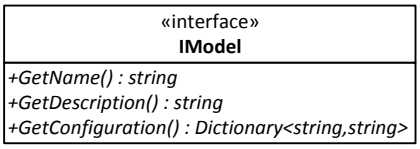
V implementaci byla využita druhá možnost.

Anotace

Atributy v C# mají stejný sémantický význam jako anotace – přidávají pomocné, či dodatečné informace k určitému prvku. Jsou tedy ideálním kandidátem pro jejich reprezentaci. Protože se v definici modelu bude chtít využít různé typy anotací, tak musí ke každé takové anotaci existovat příslušný atribut. Aby se anotace odlišili od atributů definující jinou sémantiku modelu (`Entity`, `Property` apod.) musí atribut anotace dědit z `AnnotationAttribute`, což je abstraktní třída obsahující abstraktní metodu `GetAnnotation`. Každá implementace atributu anotace musí tuto metodu předefinovat. Metoda musí vracet instanci anotace s vlastnostmi inicializovanými dle vlastností atributu. Anotace se entitě/vlastnosti přiřadí tak, že se přiřadí příslušné třídě/vlastnosti reprezentující danou entitu/vlastnost.

Model a dodatečná konfigurace

Instance třídy model se v načteném modelu bude vyskytovat vždy jednou. Jde tedy o jedinečný případ užití, pro který se nejeví žádná část jazyka C# jako vhodná. Není vhodné vytvářet nelogický způsob zápisu. Vedlo by to ke špatné čitelnost modelu, případně špatné zapamatovatelnosti způsobu definice. Pro definici jedinečné struktury modelu je vhodná metoda přímého vytváření instancí, zmíněná na začátku kapitoly. Definujme rozhraní `IModel` (Obrázek 11), které bude obsahovat metody pro získání vlastností modelu, které ještě nebyly zmíněny. Při načítaní modelu poté stačí vyhledat třídu implementující toto rozhraní, vytvořit její instanci a získat tak požadovaná data – název modelu a dodatečnou konfiguraci.



Obrázek 11: Rozhraní `IModel`

Příklad definice modelu

Následuje ukázka definice modelu. Jedná se o část fakturačního systému. Pro přehlednost jsou uvedeny jen některé vlastnosti a entity.

Následující příklad představuje entitu Invoice. Entita obsahuje vlastnosti Id, Number, Contact, InvoiceItems. U některých vlastností jsou definovány různé atributy např. Required, představující anotaci se stejným jménem a říkající, že položka je povinná. Další zajímavostí je vlastnost InvoiceItems, která je vybavena atributem Relationship, který říká, že je součástí vztahu s názvem InvoiceInvoiceItem. Protože datovým typem vlastnosti je `IList<InvoiceItem>`, tedy kolekce, tak kardinalita této strany vztahu je Many. Vlastnost Contact má jako datový typ Company, což je jiná entita modelu. Tím je vytvořen implicitní nepojmenovaný vztah.

```
[Entity]
[Display("Faktura")]
public class Invoice
{
      [Property, PrimaryKey, Required]
      public int Id { get; set; }

      [Property, Required, NavigationProperty, Label, Display("Číslo")]
      [Length(20)]
```

```
        public string Number { get; set; }

        [Property, Required, Display("Kontakt")]
        public Company Contact { get; set; }

        [Property, Relationship("InvoiceInvoiceItem"), EditorGrid, Display("Položky
faktury")]
        public IList<InvoiceItem> InvoiceItems { get; set; }
}
```

Následující blok kódu představuje entitu InvoiceItem (položku faktury). V kódu je ukázána definice druhé části vztahu InvoiceInvoiceItem, který byl uveden v předchozím příkladu.

```
[Entity, DependantEntity(typeof(Invoice))]
[Display("Položka faktury")]
public class InvoiceItem
{
        [Property, PrimaryKey, Required]
        public int Id { get; set; }

        [Property, Relationship("InvoiceInvoiceItem"), Identifying,
Display("Faktura")]
        public Invoice Invoice { get; set; }

        [Property, Required, Length(20), Label, NavigationProperty, EditInGrid,
Display("Jméno")]
        public string Name { get; set; }

        [Property, EditInGrid, Display("Číslo"), Length(20)]
        public string Number { get; set; }

        [Property, Required, EditInGrid, Display("Jednotková cena")]
        public decimal ItemPrice {get; set;}

        [Property, Required, EditInGrid, Display("Množství")]
        public decimal Quantity { get; set; }
}
```

Poslední příklad ukazuje definici modelu a dodatečné konfigurace.

```
public class Model: IModel
{
        public string Namespace { get { return "Vopet.Sample"; } }
        public string Description { get { return "Model description"; } }
        public string Name { get { return "Vopet.Sample"; } }

        public IDictionary<string, string> GetConfiguration()
        {
```

```
        Dictionary<string, string> d = new Dictionary<string, string>();

        d[ConfigurationKeys.ModelProject] = "Vopet.Sample.Model";
        d[ConfigurationKeys.ModelNamespace] = "Vopet.Sample.Model";
        d[ConfigurationKeys.ModelGUID] = "670a7c19-fa89-417e-926b-95185e38bce8";

        d[ConfigurationKeys.WebProject] = "Vopet.Sample.Web";
        d[ConfigurationKeys.WebNamespace] = "Vopet.Sample.Web";
        d[ConfigurationKeys.WebGUID] = "568db0ea-c708-4e33-b2e9-09b3a5aac356";

        d[ConfigurationKeys.Database] = "VopetSample";
        d[ConfigurationKeys.ConnectionString] = "Data
Source=rfg.vopet.net;User=sample;Password=sample;Initial
Catalog=vopet.sample;MultipleActiveResultSets=True";
        d[ConfigurationKeys.Company] = "Petr Voborník";

        return d;
        }
}
```

6.6 Implementace generátoru

Na začátku kapitoly bylo definováno jádro generátoru. Aplikace představující generátor musí obsahovat inicializaci jádra a jeho volání. Inicializace se skládá z nastavení zdroje modelu a definování úloh. Vše ostatní jsou jen věci navíc. To zda generátor bude mít příjemné grafické rozhraní, zda bude konfigurovatelný pomocí konfiguračních nástrojů, je záležitostí pracovního procesu programátora. V této práci je implementován minimalistický generátor, který nemá žádné rozhraní a jeho konfigurace probíhá změnou programu. Tento přístup je výhodný pro vývoj a testování kódu jednotlivých úkolů.

7 Generování informačního systému

Tato kapitola se zabývá problémy, které se musí řešit při generování systému, jehož architektura byla popsána v kapitole 5, pomocí generátoru navrhnutém v kapitole 6.

Při generování informačního systému se musí vygenerovat každý soubor, který bude součástí řešení. Jak bylo zmíněno dříve, za generování může být považováno vytvoření dle šablony, zkopírování, nebo obecné generování. Obecné generování je závislé případ od případu, ale pro generování dle šablony a pro kopírování lze vytvořit základní logiku, na které by se dále stavělo. Dle návrhu minimalistického generátoru generování probíhá přes tzv. úlohy. Je tedy vhodné vytvořit bázové třídy pro generování dle šablony a kopírování. V příloze 2 jsou uvedeny ukázky výstupů, které úlohy generátoru vytváří.

7.1 Generování dle šablony

Generování dle šablony je díky jeho snadné použitelnosti nejrozšířenější způsob vytváření obsahu v implementovaném řešení. Aby se mohlo generovat dle šablony, tak se musí použít nějaký šablonový systém. Vytváření vlastního systému by stálo zbytečně moc úsilí, vhodnější je použit nějaký existující. V kapitole 4.3 byl zmíněn šablonový systém T4, který byl vyvinut speciálně pro generování zdrojových kódu a jiných textových dokumentů. Tento systém by byl pro účely této práce ideální, ale bohužel ho není vhodné využít z dříve zmíněných důvodů. Při hledání vhodné šablonového nástroje se využije faktu, že mnoho systému pro tvorbu webových stránek je vlastně šablonovým nástrojem. Stačí tedy vybrat nějaký vhodný, implementovaný pro .Net framework. Pro přehled se můžou některé vyjmenovat: NHaml, NDjango, NVelocity, Spark, Web Forms, Brail, StringTemplate.Net, Razor. Výše vyjmenované byly zběžně zkoumány a nakonec byl vybrán Razor. Samotný Razor je spíše jen technologie pro parsování šablony a její překlad, ale díky open source projektu RazorEngine se z něho stává jednoduše použitelný šablonový systém. Jeho výhodou, oproti ostatním je, že může v sobě obsahovat příkazy jazyka C# a tedy plně využít možnosti .Net framework či jiných knihoven. Tuto vlastnost má i Web Forms, ale na rozdíl od něj není Razor vázán na přítomnost ASP.NET runtime a lze ho tedy využít.

V řešení byly implementovány třídy, které RazorEngine do něj integrují. Dále byl vytvořen obecný systém (sada rozhraní) pro generování dle šablony a bázová třída pro úlohy postavené na šablonách. Bázová třída usnadňuje generování tak, že ji stačí předat cestu k cílovému souboru (ten který má vytvořit), cestu k šabloně a objekt obsahující data pro šablonu. Bázová třída se pak postará o zajištění samotného generování a uložení souboru na disk. Díky tomu se logika úkolů může soustředit jen na vlastní transformaci modelu, výběr šablon a výběr cílových souborů. Protože Razor šablony mohou obsahovat příkazy v jazyce C#, tak se programátor úkolů může rozhodnout, zda transformaci modelu provede v úkolu a v šabloně data jen zobrazí, nebo zda transformace je provedena v šabloně, či je z ní volána. Všechny přístupy mají své výhody a nevýhody – převážně v udržovatelnosti a rychlosti vývoje. Důležité je, že programátor má možnost výběru a může tak vybrat vhodný způsob pro daný úkol.

7.2 Generování kopírováním

Podobně jako pro generování pomocí šablony, byla i pro generování pomocí kopírování vytvořena bázová třída úlohy. Implementace je poměrně primitivní. Umožňuje specifikovat cestu k existujícímu souboru či složce a ty pak zkopíruje na cílovou pozici.

7.3 Model

Při generování modelu se musí vygenerovat třídy reprezentující entity, filtry, sql skript pro vytvoření databáze, soubory definující mapování entit na databázové tabulky pro nhibernate, procesy, moduly a repozitáře. Všechny uvedené soubory se generují pomocí šablon.

Entity

Pro každou entitu v modelu je vytvořena její reprezentace jako třída. Vytvořené šablony počítají s tím, že každá entita bude mít jako klíčovou vlastnost (primary key v databázi) vlastnost se jménem Id. Předpoklad vychází z požadavku, který klade bázová třída pro entity. Tako třída v sobě, již mimo jiné, obsahuje vlastnost Id, takže ji nemá smysl předefinovat.

SQL skript pro vytvoření schématu databáze

Generátor obsahuje šablonu pro vytvoření schématu SQL Serveru. Pro jiný typ databáze by se tato šablona musela upravit. V šabloně je zahrnut kód pro generování tabulky pro každou entitu včetně primárního klíče a jeho indexu. Pokud je mezi entitami vztah M:N, tak pro každý takový vztah je vygenerován kód pro definici vztahové tabulky. Dle vztahů jsou v tabulkách generovány definice cizí klíčů včetně indexace.

Mapování pro nhibernate

Mapování navazuje na vygenerované třídy a jejich vlastnosti a na strukturu databáze definované v generování sql skriptu. Generování mapování vlastností je závislé na tom, zda jsou vlastnosti jednoduché, nebo zda jsou součástí vztahu, případně jakého typu. Při generování vztahů 1:M a M:N je nutné dbát na použitý typ kolekce dle typu vlastnosti entity.

Procesy, Moduly

Pro každý typ procesu a modul existuje šablona. Procesy i moduly jsou vygenerovány pro všechny entity.

Repozitáře

Pro každou entitu je vygenerován soubor třídy repozitáře a soubor jeho rozhraní. Veškerá logika repozitářů je obsažena v jejich bázových třídách. Šablony tedy neobsahují téměř žádnou logiku.

Pomocné třídy

Je zřejmé, že šablony musí generovat jména pro vlastnosti entit, cizí klíče v databázi a jiné. Tvorba těchto jmen může vyžadovat určitou logiku. Zároveň různé šablony by měly používat stejná jména pro stejné objekty. Je tedy velmi vhodné tuto logiku umístit do pomocné třídy, a když ji některá šablona potřebuje, tak ji využije. Tím se zajistí lepší udržovatelnost kódu šablon a předejde se tak zbytečnému opakování kódu.

7.4 Webové rozhraní

Řadiče

Řadiče slouží jako vstupní bod webového rozhraní, proto by měl popis generování GUI začínat u nich. Řadiče by se daly rozdělit na dvě skupiny: řadiče pro entity a ostatní. Mezi ostatní se řadí především ty, co zajišťují společnou resp. obecnou funkcionalitu systému, jako je navigace, přihlášení, či

zobrazení výchozí stránky. Řadiče entit slouží pro obsluhu procesů entit a zobrazení jejich vstupu a výstupu.

Řadiče se generují jen pro nezávislé entity. Generování řadič pro závisle entity (entity s anotací DependantEntity) nemá smysly, protože samotné nemají žádné zobrazení a operace s nimi přísluší entitám, na kterých jsou závislé. To klade větší požadavky na generování řadičů pro nezávislé entity. Musí se zjistit, zda na entitě nejsou závislé jiné entity. Pokud ano musí se pro ně inicializovat příslušné moduly pro poskytování procesu, upravit metodu pro uložení změn, tak aby se promítly i změny v závislých entitách.

View Modely

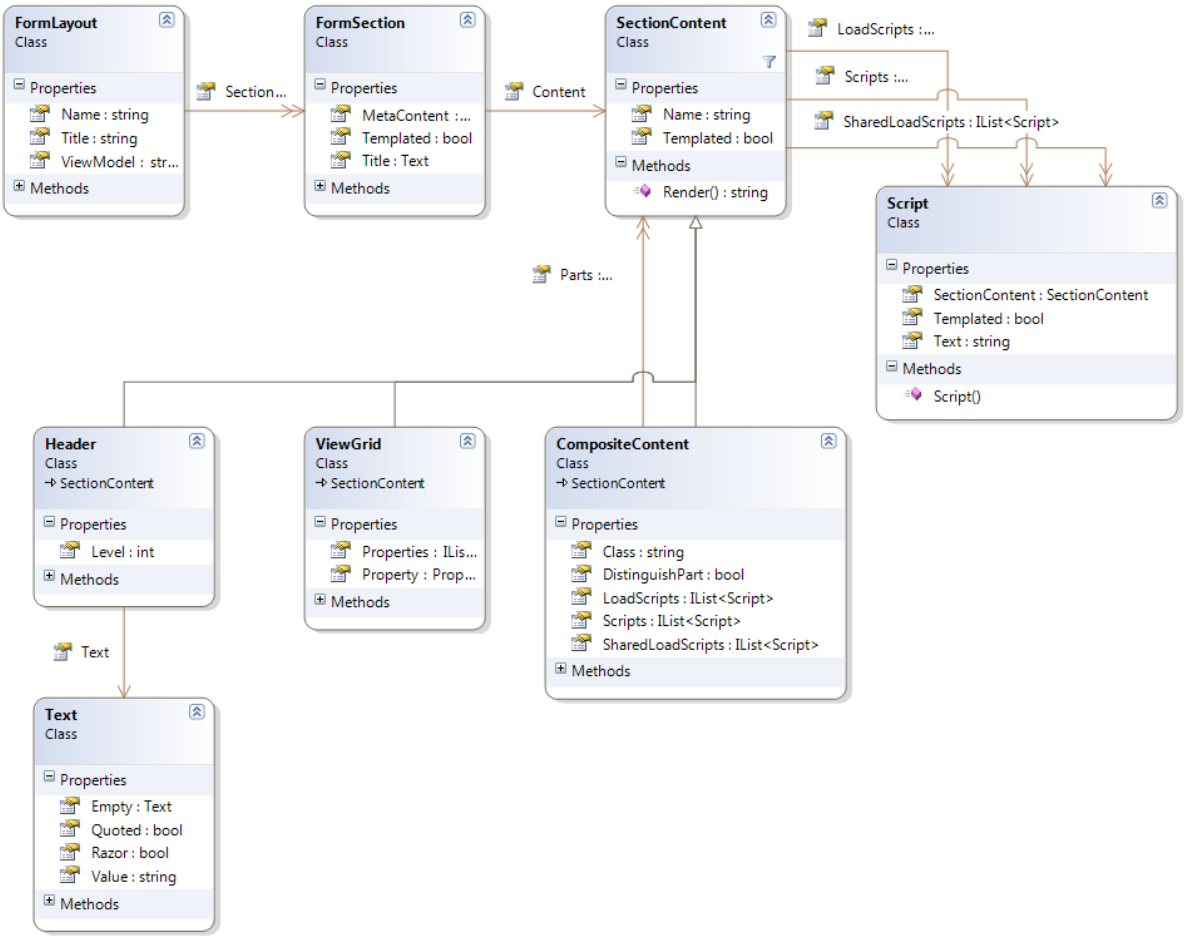
View modely jsou generované v závislosti na typu entity. Pro závislé entity se generuje jen model editace na řádku. Pro nezávislé entity se generují model pro zobrazení seznamu, pro vybrání prvku, pro editaci a pro vytvoření entity. Pokud entita je určená pro zobrazení v tabulce (forma navigace, značená anotací ViewGrid) je pro entitu ještě vygenerován model zobrazení na řádku. Při generování view modelů se musí počítat s jejich využitím, tak aby model obsahoval všechny požadované vlastnosti. Součástí vlastností view modelu jsou i atributy. V závislosti na použití modelu se generují různé atributy – většinou závislé na anotacích vlastností. Z toho důvodu byly vytvořené pomocné třídy, které se starají o mapování anotací dle použití a generování atributů dle těch anotací.

Pohledy

Generované pohledy odpovídají výstupům řadičů a dalo by se říci, že i view modelům (teoreticky by měl ke každému pohledu příslušet zvláštní view model). Tedy jako u předchozích, vygenerované pohledy závisí na tom, zda se jedná o závislou nebo nezávislou entitu.

Vzhled pohledů může být značně ovlivněn anotacemi. Proto byl pro pohledy vytvořen jednoduchý metamodel. Instance tříd metamodelu tvoří model pohledu – layout. Předání modelu pohledu šabloně pro generování pohledů dojde k vytvoření vlastního pohledu. Tento, na první pohled složitý systém, umožňuje vytvářet různé pohledy bez editace šablon, díky tomu tvoří základ pro případné rozšíření generátoru, které by umožňovalo návrh vlastních layoutů – pohledů.

Třídní diagram metamodelu pro pohledy je zobrazen na obrázku 12. Základ modelu tvoří třída reprezentující layout formuláře. Ta může obsahovat několik sekcí. Sekce mohou být s titulkem nebo bez, to ovlivňuje, jak budou vygenerovány – s titulkem tvoří viditelnou část formuláře s jasným nadpisem, bez titulku jsou spíše obecné, vhodné např. pro skryté části. Každá sekce může mít obsah. Různé typy obsahů se vytvářejí tak, že se zdědí z bázové třídy obsahu sekce. Při generování je důležité, zda je obsah tvořen šablonou, nebo obsahuje vlastní logiku pro vykreslení (vytvoření řetězce, tvořící jeho obsah). Pokud je obsah vykreslován šablonou, tak je šablona vybrána dle jména uvedeném ve vlastnosti Name. Každý obsah může obsahovat několik JavaScriptů. Ty jsou děleny do třech seznamů. Skripty, které se vykonají hned, skripty, které se vykonají po načtení stránky a skripty, které se vykonají po načtení stránky, ale jsou do pohledu vloženy jen jednou pro daný typ obsahu. Vykreslování skriptů může být, stejně jako u obsahů, šablonou nebo přímo. O vykreslení skriptů do stránky se stará šablona vykreslení layoutu – projde všechny obsahy všech sekcí. Díky tomu lze skripty vkládat do hlavičky stránky. V metamodelu jsou připraveny různé typy obsahů: nadpis, zobrazení v tabulce, editace v tabulce, více sloupcový layout editace či zobrazení vlastností, tlačítka a jiné. Důležitým typem obsahu je složený obsah. Tento obsah může obsahovat více obsahů. Díky tomu lze vytvářet složitější zobrazení. Aby fungovalo vykreslování skriptů, poskytuje vždy dynamicky vytvořené seznamy skriptů, které jsou tvořeny seznamy ze skriptů obsahů, které obsahuje.



Obrázek 12: Zjednodušený třídní diagram metamodelu pohledů

Resources

Resources představují soubory, které obsahují seznam položek klíč-hodnota. Položky představují lokalizované (přeložené) řetězce určené k zobrazení uživateli v jeho jazyce. Stávající implementace umožňuje generovat lokalizaci jen pro jeden jazyk a to češtinu. Lokalizované řetězce se získají z anotací Display či Resource, přiřazené k vlastnosti nebo entitě. Pro každou entitu je vygenerován jeden lokalizační soubor .resx a soubor v jazyce C# který tvoří třídu pro přístup k řetězcům. Tento zdrojový soubor by byl standardně vygenerován utilitou ve Visual Studiu. Bohužel je nutné ho generovat, protože kdyby se tak nestalo, musel by uživatel generátoru po vygenerování systému ve Visual Studiu ručně inicializovat znovu vygenerování tohoto souboru, což představuje uživatelskou nepříjemnost. Kromě resource souborů pro entity, generátor generuje ještě několik implicitních resource souborů, které obsahují řetězce využité v různých částech systému. Pro snazší vytváření modelu pro generátor byla vytvořena pomocná třída, která ho zajistí.

Inicializace a konfigurace

Webový projekt obsahuje několik souborů, které slouží pro konfiguraci aplikace. Ve většině těchto souborů je nutné upravit jen několik informací. Často se jedná jen o jmenné prostory, řetězec k připojení k databázi apod. Nejsložitějším z nich je nastavení komponenty Automapper, která zajišťuje mapování modelu na view modely, tak jak jsou využity v řadičích.

Statický obsah

Soubory nutné pro běh webového rozhraní, které jsou nezávislé na modelu, jsou umístěné ve složce, jejíž umístění je definováno v konfiguraci generátoru. Soubory této složky jsou zkopírovány do výsledného řešení. Těmito soubory jsou např. obrázky, css styly, použité knihovny nebo JavaScript skripty. Je pro to využita bázová třída úkolu pro generování kopírováním.

Soubory řešení a projektů

Aby se mohly vygenerované soubory využít ve Visual Studiu, tak musí být součástí projektu anebo řešení. Lze je vytvořit ručně a soubory do nich přidat, nebo je lze vygenerovat. Generování je pohodlnější cesta, která umožní po vygenerování systému rovnou otevřít řešení, zkompilovat projekty a spustit je. Pro tvorbu těchto souborů je nutné znát jména těch projektů a mít k dispozici jedinečný identifikátor ve formě GUID. Tyto informace se předají generátoru pomocí dodatečné konfigurace. Soubory projektů musí obsahovat záznam o všech souborech, které jsou členem projektu. Tyto soubory by šlo získat tak, že by každá úloha specifikovala, jaké soubory vygenerovala a úloha pro generování projektových soborů by je zpracovala. To by ale vyžadovalo, aby se generoval vždy celý systém najednou, a to nemusí být vždy vhodné. Někdy je potřeba znovu generovat jen určitou část systému a generování ostatních by jen zdržovalo. Proto je generování projektových souborů pojato jako nezávislý úkol a všechny záznamy jsou definovány v šabloně. To bohužel vede k nutnosti upravovat šablonu projektů vždy, když se do projektů přidávají nové generované soubory.

8 Možnosti rozšíření

V generátoru byl implementován základní metamodel pro definici layoutů pohledů. To vede k myšlence rozšířit metamodel informačního systému o možnost definice modelu layoutů. Společně s rozšířením metamodelu o možnost definice nových procesů, či úpravy existujících, by bylo možno ještě více ovlivňovat výsledný vzhled a chování vygenerovaného systému.

Při zmínění definice nových procesů může někoho napadnout využití notací jako je BPMN - Business Process Modeling Notation či BPEL - Business Process Execution Language. Použití těchto systémů by představovalo obtížný implementační úkol, především z pohledu propojení se specifikací modelu. Pokud by se to podařilo, mohla by to být funkcionalita, kterou by ocenili především analytici informačních systémů.

V kapitole 6.5 byly diskutovány různé způsoby zápisu a načítání modelu. Jako velmi vhodná, ale implementačně náročná varianta byla zmíněna definice modelu pomocí webového uživatelského rozhraní. Vezměme v potaz, že takové rozhraní je vlastně informační systém, jehož modelem je metamodel. Pokud se specifikuje model odpovídající metamodelu systému a vygeneruje se z něj systém, získá se solidní základ pro zmíněné uživatelské rozhraní. Toto rozhraní by se mohlo rozšířit o rozšíření zmíněné na začátku kapitoly: definice layoutů, procesů či případné použití BPMN nebo BPEL notace. Jiným rozšířením může být vytvoření úlohy pro zkompilování vygenerovaných souborů a jejich případné nahrání na produkční server.

Kombinací všech rozšíření by se mohl vytvořit systém, kde by si uživatel definoval informační systém včetně všech procesů a vzhledu, tento informační systém by se pak vygeneroval, zkompiloval a nasadil. Byl by tedy rovnou použitelný. Taková podoba generátoru trochu odbíhá od původní myšlenky projektu: ulehčit jen práci programátorovi. Určitě ale stojí za úvahu a pravděpodobně by se našlo i jeho použití.

9 Závěr

V práci byla nejprve provedena analýza společných vlastností cílových systémů. Především byly popsány vlastnosti entit a příslušných operací. Na to navázal rozbor systému z hlediska uživatele. Tím byly stanoveny požadavky na strukturu uživatelského rozhraní. V průběhu analýz byly stanovovány požadavky na zápis konceptuálního schématu modelu generovaných systémů.

Dle požadavků uvedených analýz byly vybrány implementační technologie a porovnány existující systémy pro generování kódu s ohledem na možnost jejich případného využití. Z uvedených požadavků byla navrhnuta architektura cílových systémů.

Další část práce se zabývala návrhem generátoru. Byl v ní navrhnut minimalistický generátor. Pro reprezentaci konceptuálního modelu v paměti bylo navrženo schéma metamodelu. Značná část kapitoly se zabývala různými možnostmi zápisu a načítání konceptuálního schématu modelu. Pro práci nejvhodnější byl vybrán způsob zápisu pomocí jazyka C#. Poté se popsaly způsoby, jak v něm zapsat všechny části modelu.

Předposlední kapitola se věnovala vlastnímu generováním informačního systému. Byly v ní popsány úskalí, které se musely při implementaci řešit. Součástí kapitoly je návrh metamodelu pro zápis modelu rozvržení vzhledu webových formulářů.

V poslední kapitole byly popsány různé možnosti rozšíření práce. Některé z nich umožňují projektu získat jiné použití, než byl původní cíl práce.

Výstupem práce je návrh a implementace minimalistické generátoru. Ten získává funkcionalitu rozšířením o tzv. úlohy. Výhodou návrhu generátoru a jeho metamodelu oproti jiným řešením dostupných na trhu je, že je snadno rozšiřitelný o specifikaci dodatečných informací k částem modelu. To umožňuje vytvářet šablony, které generují základ systémů přesně dle představ programátora. Minimalizuje to tedy jeho následnou práci.

V práci je implementována řada úloh a šablon, které dohromady umožňují vygenerovat systém zajišťující CRUD operace. Jejich tvorba byla časově nejnáročnější částí práce. Významným kladným bodem generátoru je, že generuje kód systému, který je čistý a dobře čitelný – vypadá téměř tak, jako by ho psal programátor sám. Generovaný kód odpovídá navrhnuté architektuře, která byla sestrojena pomocí vhodných návrhových vzorů. Programátor tak získává do ruky základ systému, který je velmi dobře udržovatelný a rozšiřitelný.

10 Literatura

[1] **Hruška, T., Křivka, Z.:** *Informační systémy (IIS, PIS) Studijní opora.* Brno, FIT VUT Brno, 2008.

[2] **Maciaszek, L. A., Liong, B. L.:** *Practical software engineering: a case study approach.* Boston, Addison-Wesley, 2005. 825s. ISBN 0-321-20465-4.

[3] **Evans, E.:** *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston, Addison-Wesley Professional, 2003. 560s. ISBN 9780321125217.

[4] **Herrington, J.:** *Code Generation in Action.* Greenwich, Manning Publications, 2003. 368s. ISBN-13 978-1930110977.

[5] **Microsoft Patterns & Practices Team:** *Microsoft Application Architecture Guide (Patterns & Practices).* Grove City, Microsoft Press, 2009. 560s. ISBN 978-0735627109.

[6] **Nielsen, J., Valík, L.:** *Web design.* Vyd. 1. Praha, SoftPress, 2002. 382s. ISBN 80-864-9727-5.

[7] **Microsoft:** *Vytváříme zabezpečené aplikace v Microsoft ASP.NET.* Vyd. 1. Brno, Computer Press, 2004. 542s. ISBN 80-251-0466-4.

[8] **Esposito, D.:** Web Forms vs. ASP.NET MVC. *.NET Architectonics.* [Online] 11. 04. 2009. [Citace: 23. 12. 2010.] URL <http://weblogs.asp.net/despos/archive/2009/04/11/web-forms-vs-asp-net-mvc.aspx>.

[9] **Gadodia, V.:** Choosing Between WebForms and MVC. *Habitually Good.* [Online] 16. 12. 2008. [Citace: 23. 12. 2010.] URL <http://blog.gadodia.net/choosing-between-webforms-and-mvc/>.

[10] **Sanderson, S.:** *Pro ASP.NET MVC 2 Framework.* New York, Apress, 2010. ISBN 978-1-4302-2887-5.

[11] **Fowler, M.:** *Patterns of Enterprise Application Architecture.* Boston, Addison Wesley, 2002. 560s. ISBN 0-321-12742-0.

[12] **Fowler, M.:** Inversion of Control Containers and the Dependency Injection pattern. *martinfowler.com.* [Online] 23. 01. 2004. [Citace: 25. 12. 2010.] URL <http://martinfowler.com/articles/injection.html>.

[13] Overview. *nInject.* [Online] Nate Kohari, open source community, 25. 12. 2010. [Citace: 25. 12. 2010.] URL <http://ninject.org>.

[14] **Dentler, J.:** *NHibernate 3.0 Cookbook.* Birmingham, Packt Publishing, 2010. ISBN 978-1-84951-304-3.

[15] **Larman, C.:** *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* 3rd ed. New Jersey, Prentice Hall, 2005. 703s. ISBN 01-314-8906-2.

Seznam příloh

- DVD se zdrojovými kódy, textem práce a manuálem pro použití generátoru
- Příloha 1 – definice ukázkového modelu
- Příloha 2 – Ukázky částí vygenerovaného systému

Příloha 1 – definice ukázkového modelu

P1.1 Entita faktura – Invoice.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Generator.LibraryModelSource;
using Generator.LibraryModelSource.AnnotationAttributes;
using Generator.Helpers;

namespace SampleModel
{
     [Entity]
     [Display("Faktura")]
     [TwoColumn]
     [Resource(LocalizationHelper.MenuTitle, "Faktury")]
     [Resource(LocalizationHelper.MenuDescription, "Seznam faktur")]
     [Resource(LocalizationHelper.ListTitle, "Seznam faktur")]
     [Resource(LocalizationHelper.CreateTitle, "Nová faktura")]
     [Resource(LocalizationHelper.UpdateTitle, "Editace faktury")]
     [Resource(LocalizationHelper.ReadTitle, "Faktura")]
     [Resource(LocalizationHelper.DeleteTitle, "Smazaní faktury")]
     [Resource(LocalizationHelper.InsertCommand, "Vytvořit fakturu")]
     [Resource(LocalizationHelper.UpdateCommand, "Uložit")]
     [Resource(LocalizationHelper.DeleteCommand, "Smazat")]
     [Resource(LocalizationHelper.DeleteConfirmation, "Opravdu si přejete smazat f
akturu?")]
     public class Invoice
     {
          [Property, PrimaryKey, Required]
          public int Id { get; set; }

          [Property, Required, NavigationProperty, Label, Display("Číslo")]
          [Length(20)]
          public string Number { get; set; }

          [Property, Required, Display("Kontakt")]
          public Company Contact { get; set; }


[Property, Relationship("InvoiceInvoiceItem"), EditorGrid, Display("Položky faktur
y")]
          [Resource(LocalizationHelper.EGAdd, "Přidat")]
          [Resource(LocalizationHelper.EGDelete, "Smazat")]

[Resource(LocalizationHelper.EGDeleteConfirm, "Opravdu si přejete smazat položku f
aktury?")]
          [Resource(LocalizationHelper.EGEdit, "Upravit")]
          public IList<InvoiceItem> InvoiceItems { get; set; }
     }
}
```

P1.2 Entita položka faktury – InvoiceItem.cs

```
using Generator.LibraryModelSource;
using Generator.LibraryModelSource.AnnotationAttributes;

namespace SampleModel
{
     [Entity, DependantEntity(typeof(Invoice))]
     [Display("Položka faktury")]
     public class InvoiceItem
     {
          [Property, PrimaryKey, Required]
          public int Id { get; set; }


     [Property, Relationship("InvoiceInvoiceItem"), Identifying, Display("Faktura"
)]
          public Invoice Invoice { get; set; }


     [Property, Required, Length(20), Label, NavigationProperty, EditInGrid, Displ
ay("Jméno")]
          public string Name { get; set; }

          [Property, EditInGrid, Display("Číslo"), Length(20)]
          public string Number { get; set; }

          [Property, Required, EditInGrid, Display("Jednotková cena")]
          public decimal ItemPrice {get; set;}

          [Property, Required, EditInGrid, Display("Množství")]
          public decimal Quantity { get; set; }
     }
}
```

Příloha 2 – Ukázky částí vygenerovaného systému

P2.1 Třída entity Invoice – Ivoice.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Iesi.Collections.Generic;
using Vopet.Framework;

namespace Vopet.Sample.Model.Model
{
	public partial class Invoice: Entity
	{
		#region Fields

		private String _number;
		private Company _contact;
		private IList<InvoiceItem> _invoiceItems;

		#endregion

		#region Properties
		public virtual String Number
		{
			get{ return _number; }
			set{ _number = value; }
		}
		public virtual Company Contact
		{
			get{ return _contact; }
			set{ _contact = value; }
		}
		public virtual IList<InvoiceItem> InvoiceItems
		{
			get{ return _invoiceItems; }
			set{ _invoiceItems = value; }
		}
		#endregion

		#region Constructor

		public Invoice()
		{
			_invoiceItems = new List<InvoiceItem>();
		}
		#endregion
	}
}
```

P2.2 nHibernate mapování pro třídu Invoice – Invoice.hbm.xml

```
<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"

       namespace="Vopet.Sample.Model.Model" assembly="Vopet.Sample.Model">
     <class name="Invoice" table="InvoiceSet">

     <id name="Id" column="Id" type="Int32" unsaved-value="-1">
          <generator class="native" />
     </id>

     <property name="Number" type="String" length="20" not-null="true" />
        <many-to-one name="Contact" class="Company" column="ContactId" />
        <bag name="InvoiceItems" inverse="true">
          <key column="InvoiceId"/>
          <one-to-many class="InvoiceItem"/>
        </bag>

     </class>
</hibernate-mapping>
```

P2.3 Třída filtru pro fakturu – InvoiceFilter.cs

```
namespace Vopet.Sample.Model.Filters
{
     public partial class InvoiceFilter: DefaultFilter<Invoice>
     {
          #region Properties

          public String Number { get; set;}

          #endregion

          #region Apply

          public override IQueryable<Invoice> Apply(IQueryable<Invoice> query)
          {
               if (Number != null)
               {
                    query = query.Where(e => e.Number.Contains(Number));
               }
               query = base.Apply(query);
               return query;
          }

          #endregion
     }
}
```

P2.4 Repozitář pro Fakturu – InvoiceRepository.cs

```
using Vopet.Framework;
using Vopet.Sample.Model.Model;

namespace Vopet.Sample.Model.Repositories
{
	public partial class InvoiceRepository: HibernateRepositoryBase<Invoice>, IIn
voiceRepository
	{
	}
}
```

P2.4 Rozhraní repozitáře pro Fakturu – IInvoiceRepository.cs

```
using Vopet.Framework;
using Vopet.Sample.Model.Model;

namespace Vopet.Sample.Model.Repositories
{
	public partial interface IInvoiceRepository: IRepository<Invoice>
	{
	}
}
```

P2.5 Create Proces pro fakturu – CreateProcess.cs

```
using Vopet.Framework;
using Vopet.Sample.Model.Model;
using Vopet.Sample.Model.Repositories;

namespace Vopet.Sample.Model.Processes.Invoice
{
	public class CreateProcess: IProcess
	{
		protected IInvoiceRepository _invoiceRepository;
		public Model.Invoice Invoice { get; set; }

		public CreateProcess(IInvoiceRepository invoiceRepository)
		{
			_invoiceRepository = invoiceRepository;
		}

		public void Run()
		{
			_invoiceRepository.Add(Invoice);
		}
	}
}
```

P2.6 Get Proces pro fakturu – GetProcess.cs

```
using Vopet.Framework;
using Vopet.Sample.Model.Model;
using Vopet.Sample.Model.Repositories;

namespace Vopet.Sample.Model.Processes.Invoice
{
	public class GetProcess: IProcess
	{
		protected IInvoiceRepository _invoiceRepository;
		public int Id { get; set; }

		public GetProcess(IInvoiceRepository invoiceRepository)
		{
			_invoiceRepository = invoiceRepository;
		}

		public Model.Invoice Run()
		{
			return _invoiceRepository.Get(Id);
		}

		public Model.Invoice Load()
		{
			return _invoiceRepository.Load(Id);
		}

		public Model.Invoice Get()
		{
			return Run();
		}
	}
}
```

P2.7 List Proces pro fakturu – ListProcess.cs

```
using Vopet.Framework;
using Vopet.Sample.Model.Filters;
using Vopet.Sample.Model.Repositories;

namespace Vopet.Sample.Model.Processes.Invoice
{
	public class ListProcess: IProcess
	{
		protected IInvoiceRepository _invoiceRepository;
		public InvoiceFilter Filter { get; set; }

		public ListProcess(IInvoiceRepository invoiceRepository)
		{
			_invoiceRepository = invoiceRepository;
		}

		public IList<Model.Invoice> Run()
		{
			if(Filter == null)
				return _invoiceRepository.GetCollection();
			return _invoiceRepository.GetCollection(Filter);
		}
	}
}
```

P2.8 Delete Proces pro fakturu – ListProcess.cs

```
using Vopet.Framework;
using Vopet.Sample.Model.Model;
using Vopet.Sample.Model.Repositories;

namespace Vopet.Sample.Model.Processes.Invoice
{
        public class DeleteProcess: IProcess
        {
                protected IInvoiceRepository _invoiceRepository;
                public Model.Invoice Invoice { get; set; }

                public DeleteProcess(IInvoiceRepository invoiceRepository)
                {
                        _invoiceRepository = invoiceRepository;
                }

                public void Run()
                {
                        _invoiceRepository.Remove(Invoice);
                }
        }
}
```

P2.9 Modul pro faktury – InvoiceModule.cs

```
using Vopet.Framework;
using Vopet.Sample.Model.Model;
using Vopet.Sample.Model.Repositories;
using Vopet.Sample.Model.Processes.Invoice;

namespace Vopet.Sample.Model.Modules
{
     public class InvoiceModule
     {
          protected IInvoiceRepository _invoiceRepository;

          public InvoiceModule(IInvoiceRepository invoiceRepository)
          {
               _invoiceRepository = invoiceRepository;
          }
          public CreateProcess GetCreateProcess()
          {
               return new CreateProcess(_invoiceRepository);
          }
          public GetProcess GetGetProcess()
          {
               return new GetProcess(_invoiceRepository);
          }
          public ListProcess GetListProcess()
          {
               return new ListProcess(_invoiceRepository);
          }
          public DeleteProcess GetDeleteProcess()
          {
               return new DeleteProcess(_invoiceRepository);
          }
     }
}
```

P2.10 Controller pro faktury – InvoiceController.cs

Z důvodu délky řádku a velkého stupně odsazení původního souboru je výpis v určitých místech nepřehledně formátován.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Vopet.Sample.Model.Model;
using Vopet.Sample.Model.Modules;
using Vopet.Sample.Model.Filters;
using Vopet.Sample.Web.ViewModels;
using Vopet.Sample.Web.ViewModels.Invoice;
using AutoMapper;
using Vopet.Framework;

namespace Vopet.Sample.Web.Controllers
{
     public class InvoiceController : Controller
     {

          private InvoiceModule _processes;
          private CompanyModule _companyModule;
          private InvoiceItemModule _invoiceItemModule;


          public InvoiceController(InvoiceModule invoiceModule
     ,CompanyModule companyModule
               ,InvoiceItemModule invoiceItemModule
)
          {
               _processes = invoiceModule;
               _companyModule = companyModule;
               _invoiceItemModule = invoiceItemModule;
          }

          //
          // GET: /Invoice/

          public ActionResult Index()
          {
               return RedirectToAction("List");
          }

          public ActionResult List(InvoiceFilter filter)
          {
               var process = _processes.GetListProcess(); //get process
               process.Filter = filter; //set process conditions
               var invoiceList = process.Run(); //run process

     var invoiceListModel = Mapper.Map<IList<Invoice>, IList<InvoiceListModelItem>
>(invoiceList); //map result to view model
               var model = new ListModel { Items = invoiceListModel };
               return View(model); //show view
          }
```

```
        public ActionResult Get(int id)
        {
            var process = _processes.GetGetProcess(); //get process
            process.Id = id; //set process conditions
            var invoice = process.Run(); //run process

    var model = Mapper.Map<Invoice, InvoiceEditModel>(invoice); //map result to v
iew model
            return View(model); //show view
        }

        public ActionResult Edit(int id)
        {
            var process = _processes.GetGetProcess(); //get process
            process.Id = id; //set process conditions
            var invoice = process.Run(); //run process

    var model = Mapper.Map<Invoice, InvoiceEditModel>(invoice); //map result to v
iew model
            return View(model); //show view
        }

        [HttpPost]
        public ActionResult Edit(InvoiceEditModel model)
        {
            if (!ModelState.IsValid)
                return View(model);

            var process = _processes.GetGetProcess(); //get process
            process.Id = model.Id; //set process conditions
            var invoice = process.Run(); //run process
            Mapper.Map<InvoiceEditModel,Invoice>(model, invoice);
    //update entity with data form user

            var companyProcess = _companyModule.GetGetProcess();
            companyProcess.Id = model.ContactId;
            invoice.Contact = companyProcess.Load();


            {

    model.InvoiceItems = model.InvoiceItems ?? new List<ViewModels.InvoiceItem.In
voiceItemRowEditModel>();
                 var addProcess = _invoiceItemModule.GetCreateProcess();
                 var deleteProcess = _invoiceItemModule.GetDeleteProcess();
                foreach (var item in model.InvoiceItems)
                {
                    //add new
                    if (item.Id <= 0 || item.Id == null)
                    {

    var invoiceItem = Mapper.Map<ViewModels.InvoiceItem.InvoiceItemRowEditModel,
InvoiceItem>(item);
                        invoiceItem.Invoice = invoice;
                        addProcess.InvoiceItem = invoiceItem;
                        addProcess.Run();
                    }
                    else
```

```
                        {

     InvoiceItem invoiceItem = invoice.InvoiceItems.Where(e => e.Id == item.Id).Fi
rstOrDefault();
                              if (invoiceItem != null)
                              {
                                 //update existing

     Mapper.Map<ViewModels.InvoiceItem.InvoiceItemRowEditModel, InvoiceItem>(item,
 invoiceItem);
                              }
                              //else - something is wrong
                        }
                  }


     var deleted = invoice.InvoiceItems.Where(e => model.InvoiceItems.FirstOrDefau
lt(k => k.Id == e.Id) == null).ToList();
                  foreach (var invoiceItem  in deleted)
                  {
                        deleteProcess.InvoiceItem = invoiceItem ;
                        deleteProcess.Run();
                  }
            }

            UnitOfWork.Current.Transaction.Commit(); //save

            return View(model); //show view
      }

      public ActionResult Add()
      {
            return View();
      }

      [HttpPost]
      public ActionResult Add(InvoiceCreateModel model)
      {
            if (!ModelState.IsValid)
                  return View(model);

            var invoice = Mapper.Map<InvoiceCreateModel, Invoice>(model);
            var companyProcess = _companyModule.GetGetProcess();
            companyProcess.Id = model.ContactId;
            invoice.Contact = companyProcess.Load();

            var process = _processes.GetCreateProcess();
            process.Invoice = invoice;
            process.Run();
            UnitOfWork.Current.Transaction.Commit(); //save
            return RedirectToAction("Edit", new { id = invoice.Id} );
      }


      [HttpPost]
      public ActionResult Delete(int id)
      {
            //get invoice
```

```
                var process = _processes.GetGetProcess();
                process.Id = id;
                var invoice = process.Run();

                //delete invoice
                var deleteProcess = _processes.GetDeleteProcess();
                deleteProcess.Invoice = invoice;
                deleteProcess.Run();

                UnitOfWork.Current.Transaction.Commit(); //save

                return RedirectToAction("Index");
            }
            public JsonResult SelectList(string term)
            {
                var process = _processes.GetListProcess();

      process.Filter = new InvoiceFilter { Number = term, SortKeys= "Number" };
                var list = process.Run();
                var result = Mapper.Map<IList<Invoice>,  ListItem[]>(list);
                return Json(result, JsonRequestBehavior.AllowGet);
            }

            public PartialViewResult AddInvoiceItem(int position)
            {

      ViewData.TemplateInfo.HtmlFieldPrefix = String.Format("InvoiceItems[{0}]", po
sition);

      var model = new ViewModels.InvoiceItem.InvoiceItemRowEditModel { Id = -1 };
                return PartialView("EditorTemplates/InvoiceItemRowEditModel");
            }

        }
}
```

P2.11 View Model pro editaci faktury – InvoiceEditModel.cs

```
using System.ComponentModel.DataAnnotations;
using Vopet.Sample.Web.Resources;


namespace Vopet.Sample.Web.ViewModels.Invoice
{
     public class InvoiceEditModel
     {
          #region Properties
[Required]
          public Int32 Id { get; set; }

[Display(Name = "NumberDisplayName", Description = "NumberDescription", Prompt = "
NumberPrompt", ShortName = "NumberShortName", ResourceType = typeof(InvoiceStrings
))]
[Required]
          public String Number { get; set; }

[Display(Name = "ContactDisplayName", Description = "ContactDescription", Prompt =
 "ContactPrompt", ShortName = "ContactShortName", ResourceType = typeof(InvoiceStr
ings))]
[Required]
          public Int32 ContactId { get; set; }
[Display(Name = "ContactDisplayName", Description = "ContactDescription", Prompt =
 "ContactPrompt", ShortName = "ContactShortName", ResourceType = typeof(InvoiceStr
ings))]
[Required]
          public String ContactName { get; set; }


     public IList<InvoiceItem.InvoiceItemRowEditModel> InvoiceItems { get; set; }


          #endregion
     }
}
```

P2.12 Pohled pro editaci faktury – Views/Invoice/Edit.cshtml

```
@model Vopet.Sample.Web.ViewModels.Invoice.InvoiceEditModel
@using Vopet.Framework.Web
@{
        ViewBag.Title = "";
}

@section head
{
<script type="text/javascript">

function UpdateAddInvoiceItemLink()
{
        $("#InvoiceItemsBody").show(0);
        var buttonName = "addInvoiceItemButton";
        var href = $("#" + buttonName).attr("href");
        var parts = href.split("=");
        parts[1] = Number(parts[1]) + 1;
        $("#" + buttonName).attr("href", parts.join("="));
}
$(function ()
{
$("#addInvoiceItemButton").click(function ()
{
        $.ajax({
                url: this.href,
                type: "POST",
                cache: false,

        success: function (html) { $("#InvoiceItemsBody").append(html); UpdateAddI
nvoiceItemLink(); }
        });
        return false;
});

$('tr td a.deleteRow').live('click', function()
{
        $(this).parent('td').parent('tr').remove();
        return false;
});      });

</script>
}

@using (Html.BeginForm()) {

<div class="section">
        <h2>@(InvoiceStrings.UpdateTitle)</h2>

<div class="twoColumn">

<div class="column">
        @Html.HiddenFor(e => e.Id)

<div class="property">
        <div class="editor-label">
                @Html.LabelFor(e => e.Number)
        </div>
```

```
        <div class="editor-field">
                @Html.EditorFor(e => e.Number)
                @Html.ValidationMessageFor(e => e.Number)
        </div>
</div>
</div>

<div class="column">

<div class="property">
        <div class="editor-label">
                @Html.LabelFor(e => e.ContactName)
        </div>
        <div class="editor-field">
@Html.AutoCompleteFor(m => m.ContactId, m => m.ContactName, "/Company/SelectList")
                @Html.ValidationMessageFor(e => e.ContactName)
        </div>
</div>

</div>
</div>
</div>

<div class="section">
        <h2>@(InvoiceStrings.InvoiceItemsDisplayName)</h2>

<table class="editorGrid">
<thead>
<tr>
         <th>@(InvoiceItemStrings.NameShortName)</th>
         <th>@(InvoiceItemStrings.NumberShortName)</th>
         <th>@(InvoiceItemStrings.ItemPriceShortName)</th>
         <th>@(InvoiceItemStrings.QuantityShortName)</th>
        <th></th>
</tr>
</thead>
<tbody id="InvoiceItemsBody">
@{ var count = Model.InvoiceItems != null ? Model.InvoiceItems.Count : 0;}
@if(count >0)
{
        @Html.EditorFor(e => e.InvoiceItems);
}
</tbody>
</table>

<div class="submitArea">
        @Html.ActionLink(InvoiceStrings.InvoiceItemsGridAdd, "AddInvoiceItem", new
 { position = count }, new { @class = "button", id = "addInvoiceItemButton" })
</div>
</div>

<div class="submitArea">
        <input type="submit" class="button" value="@(InvoiceStrings.UpdateCommand)
" />
</div>
}
```