VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

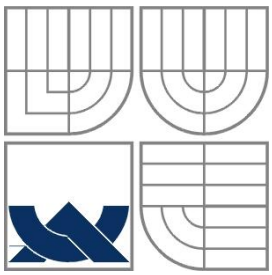
FRAMEWORK PRO DYNAMICKÉ VYTVÁŘENÍ INFORMAČNÍHO SYSTÉMU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

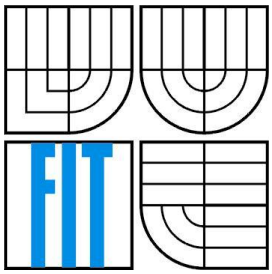
AUTOR PRÁCE
AUTHOR

Bc. PAVEL DZIADZIO

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

FRAMEWORK PRO DYNAMICKÉ VYTVÁŘENÍ INFORMAČNÍHO SYSTÉMU

FRAMEWORK FOR DYNAMIC INFORMATION SYSTEM CREATING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL DZIADZIO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LADISLAV RUTTKAY

BRNO 2012

Abstrakt

Tato práce se zabývá analýzou a definicí požadavků na framework podporující snadný a rychlý vývoj podnikových informačních systémů. Hlavním přínosem frameworku je zrychlení vývoje, snížení ceny a zkvalitnění produktu. Práce porovnává a zhodnocuje existující dostupné nástroje. Výsledkem je podrobný návrh a realizace vlastního flexibilního řešení, které splňuje všechny stanovené požadavky a odstraňuje nevýhody dostupných řešení. V závěru práce jsou uvedeny možnosti a směry dalšího vývoje.

Abstract

This thesis analyzes and defines requirements of framework, which helps with quick and effortless development of business information systems. The main goal of the framework is development acceleration, price reduction and overall improvement of product quality. The thesis also compares and evaluates existing tools. The result is detailed design and implementation of own flexible solution that fulfills all defined requirements and removes disadvantages of existing solutions. In the scope for further studies framework development possibilities and directions are listed.

Klíčová slova

Framework pro informační systém, ERP, ORM, .NET Framework, požadavky na framework pro IS, podnikový informační systém, LightSwitch, eXpressApp Framework, Cuberry, MM .NET Application Framework, entita, generované formuláře, WCF RIA Services, virtuální režim DataGridView

Keywords

Framework for information system, ERP, ORM, .NET Framework, requirements for IS framework, business information system, LightSwitch, eXpressApp Framework, Cuberry, MM .NET Application Framework, entity, generated forms, WCF RIA Services, DataGridView virtual mode

Citace

Dziadzio Pavel: Framework pro dynamické vytváření informačního systému, diplomová práce, Brno, FIT VUT v Brně, 2012

Framework pro dynamické vytváření 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. Pavel Dziadzio
7. května 2012

Poděkování

Na tomto místě bych rád poděkoval vedoucímu této diplomové práce, panu Ing. Ladislavu Ruttkayovi, za odborné vedení. Dále pak rodině a přítelkyni za podporu při psaní této práce a v neposlední řadě lidem kolem projektu OT2011 za příležitost k získání cenných zkušeností využitých i při tvorbě této práce.

© Pavel Dziadzio, 2012
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod

Informační systémy dnes nacházejí uplatnění téměř ve všech odvětvích podnikání. Zavedení kvalitního informačního systému přináší podniku řadu výhod v podobě větší efektivity podnikových procesů. Vývoj kvalitního a cenově dostupného informačního systému (IS) ale není snadná záležitost. Jeho tvůrci musí řešit množství problémů specifických pro tento typ aplikací. Tyto problémy se u většiny IS stále opakují, proto se nabízí možnost použití nástroje (frameworku), který je bude řešit, a programátoři se tak mohou zaměřit jen na požadavky konkrétního systému. Stejný nástroj navíc bude možné použít i jako základ pro další IS. Specifikem oblasti informačních systémů je rovněž požadavek na časté úpravy a doplňování systému, což u špatně navržených aplikací může být problém. Proto by měl použitý nástroj zároveň určovat architekturu výsledné aplikace s důrazem na možnost a snadnou proveditelnost budoucích úprav.

Cílem této práce tedy bude formulovat požadavky na framework sloužící jako základ pro tvorbu podnikových IS a tento framework implementovat. Jeho hlavním přínosem by mělo být podstatné zrychlení vývoje IS, což se také projeví snížením nákladů na jeho výrobu. Zároveň tím bude zajištěna určitá minimální úroveň kvality výsledného produktu. Tato problematika je blíže probrána v kapitole 2.

V kapitole 3 se následně zabývám studií, porovnáním a zhodnocením již existujících frameworků a obdobných nástrojů.

Následující, 4. kapitola obsahuje popis architektury vlastního řešení frameworku, který bude splňovat všechny dříve stanovené požadavky a odstraňovat nedostatky existujících řešení. Dále se tato kapitola zabývá detailním rozborem a návrhem klíčových částí frameworku včetně uživatelského rozhraní a návrhu prototypového řešení využití ve vícevrstvé SW architektuře. Kapitola 5 uvádí některé zajímavosti a zvolené koncepty při implementaci navrženého řešení.

Závěrem této práce je uvedeno zhodnocení dosažených výsledků a nástin možností a oblastí dalšího vývoje vytvořeného frameworku.

2 Stanovení požadavků

V této kapitole bude blíže určena oblast zájmu pro účely této práce. Dále budou analyzovány, stanoveny a přesně formulovány požadavky, které musí zmiňovaný framework splňovat, aby přinesl očekáváný užitek.

2.1 Motivace a oblast využití

V současné době potřebují téměř všechny společnosti i drobní podnikatelé evidovat množství informací o svých zaměstnancích, zákaznících, dodavatelích, prodejích, apod. Tyto potřeby pokrývá nespočet tzv. *ERP systémů*. Enterprise Resource Planning (ERP) systémy jsou softwarové nástroje používané k řízení podnikových dat. Pomáhají podnikům v oblasti dodavatelského řetězce, příjmu materiálu, skladového hospodářství, přijímání objednávek od zákazníků, plánování výroby, expedice zboží, účetnictví, řízení lidských zdrojů a dalších podnikových funkcích [1]. Tyto oblasti včetně jejich označení také znázorňuje obrázek 2.1.

Obrázek 2.1: Oblasti zájmu ERP systémů (převzato z [2])

ERP systémy dodává v různých podobách mnoho výrobců SW. Jedná se jak o tzv. balíkové systémy, kdy výrobce dodává totožný systém pro určitou skupinu zákazníků, tak i o systémy vyráběné nebo upravované na zakázku podle potřeb daného podniku. Čím dál více se také dostává do popředí outsourcing v IT a SaaS[1] [5]. Díky svému rozšíření a počtu konkurenčních produktů jsou tyto systémy dobře finančně dostupné.

Zároveň si také mnoho převážně menších a středních podniků začíná uvědomovat, že by své podnikové procesy mohly podstatně vylepšit a zefektivnit pomocí vhodného SW nástroje. Většinou se pak jedná o velmi úzce specializované procesy, které jsou zcela unikátní pro daný podnik nebo malou skupinu podniků (např. evidence servisní činnosti a oprav na výrobních strojích, podrobná evidence odpracovaných hodin na stanovených úkolech apod.). Takové moduly však výrobci ERP systémů do svých produktů nezařazují. Podniku pak nezbývá jiná možnost, než si nechat současný ERP systém upravit na zakázku nebo rovnou objednat specializovaný IS (informační systém). Výroba takového zakázkového IS je ale nákladná záležitost a mnoho menších podniků

[1] SaaS (Software as a Service) – jedná se o poskytování softwaru formou outsourcingu. Podnik nevlastní SW ani HW, na kterém SW běží. Obojí si pouze pronajímá.

na jejich pořízení nemá dostatečné finanční prostředky. Důsledkem toho pak často vznikají sice levné ale nekvalitní a špatně navržené nástroje, které ve výsledku podniku nepřinesou téměř žádný užitek.

Hlavním cílem této práce je poskytnout programátorům nástroj (framework), který jim podstatně zjednoduší a urychlí práci. Musí tedy řešit pokud možno veškeré rutinní práce spojené s programováním převážně zakázkových, úzce specializovaných informačních systémů. Jedná se typicky o nízkoúrovňové operace související s přístupem k datovému úložišti, implementace základních operací pro vytvoření nového záznamu, změnu a odstranění. Tyto operace zpravidla řeší vrstva aplikace označovaná jako *ORM* (Object-Relational Mapping). Další oblastí, v níž je možné práci významně zjednodušit je uživatelské rozhraní aplikace. Jedná se především o tzv. *data binding* – provázání grafických uživatelských komponent s prvky datových objektů aplikace.

Framework ale musí mimo tyto rutinní záležitosti řešit i další problémy, se kterými se musí programátoři vypořádat téměř v každém IS. Jsou to operace typu správa a autentizace uživatelů systému, nastavování a kontrola jejich oprávnění k určitým operacím, řešení konkurenční editace záznamů více uživateli najednou, ukládání některých dat do různých mezipamětí, filtrování a řazení dat, atd. Obzvláště v těchto oblastech se pak mohou nejčastěji projevit chyby způsobené nevhodným návrhem architektury aplikace nebo nedostatečným testováním. Navíc se tyto problémy často projeví až v době, kdy je již systém nasazen v ostrém provozu a tak se jejich odstranění stává mnohem nákladnější.

Přínos frameworku by měl být znatelný ve 2 možných scénářích:

- Snížení nákladů a urychlení vývoje v případě objednávky informačního systému u externího dodavatele.

- Zkvalitnění výsledného produktu, popřípadě vůbec umožnění realizace v případě tvorby informačního systému IT oddělením v rámci podniku.

2.2 Určení cílové technologie

Pro další potřeby této práce je nutné stanovit dostupné technologie, které budou dále předmětem zájmu. Vzhledem ke značné převaze platformy Microsoft v podnikovém prostředí se omezíme na technologie pro platformu Microsoft Windows.

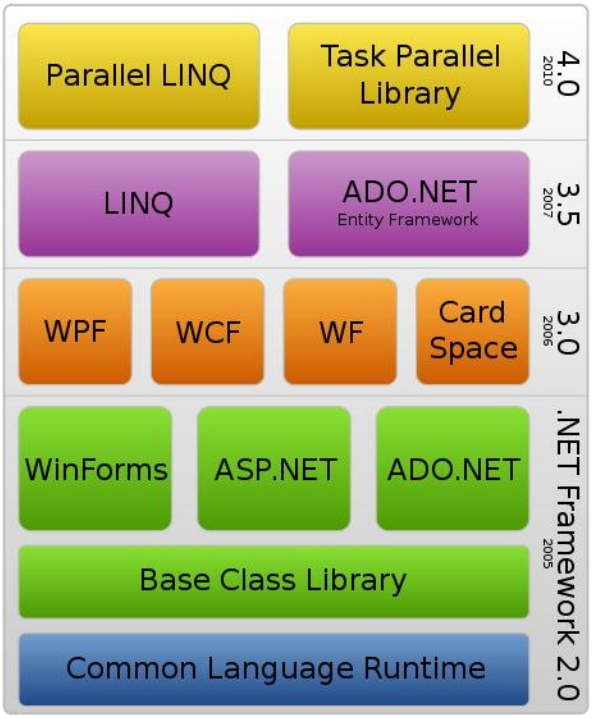
Požadovaná životnost většiny informačních systémů je poměrně dlouhá. Zároveň také často bývá vyžadována průběžná aktualizace a vylepšování nebo rozšiřování systému. Z toho důvodu je vhodné na počátku zvolit moderní, dobře dostupnou platformu pro vývoj (programovací jazyk, vývojové prostředí a další použité knihovny. Tou je bezesporu *.NET Framework* a *programovací jazyk C#.*

2.2.1 .NET Framework

Jedná se o aplikační framework vyvinutý firmou Microsoft primárně pro jejich operační systémy rodiny Windows. Počínaje Windows Vista a Windows Server 2003 je přímou součástí operačního systému.

Základní stavebním kamenem je *CLR* (*Common Language Runtime*), což je virtuální běhový stroj, který implementuje specifikaci CLI[2], poskytuje aplikacím GC[3] pro správu paměti, zpracování výjimek, bezpečnostní vrstvu pro přístup ke zdrojovému kódu, bohatý systém typů a další [3]. CLR za běhu průběžně překládá zdrojový IL (intermediate language) kód způsobem *Just In Time* (*JIT*). Nejedná se ale o klasický interpret. CLR překládá IL kód do strojového kódu předtím, než jej provede. Tento překlad neprobíhá opakovaně. Na rozdíl od interpretů je tak snížení výkonu způsobené překladem pouze jednorázové. Při opětovném vykonávání jednou přeloženého kódu má již CLR tento kód k dispozici a může jej rovnou začít vykonávat. Jedná se tedy o vytváření spravovaných (managed) aplikací.

Díky IL kódu není CLR vázáno na konkrétní programovací jazyk. Platforma .NET nabízí hned několik programovacích jazyků, které se překládají do IL kódu. Je možné sdílet moduly napsané v různých jazycích, protože součástí standardů jsou i *metadata* obsažena přímo uvnitř spravovaného modulu. Metadata kompletně popisují obsah *sestavení* (assembly). Odpadá tak nutnost společně s binárními moduly distribuovat i hlavičkové soubory známé např. z jazyka C/C++. Nejvíce rozšířený a nejznámější je jazyk *Visual C# .NET* (dále C#), který je svou syntaxí podobný jazyku C++ nebo Java. Jedná se o silně typový, objektově orientovaný jazyk. Mimo C# existují i další – Visual Basic .NET nebo Visual F# .NET. Pro účely této práce se ale zaměřím pouze na využití jazyka C#.



Obrázek 2.2: Součásti různých verzí .NET Frameworku. Zdroj: http://commons.wikimedia.org/wiki/File:DotNet.svg

Součástí .NET Frameworku je již od první verze knihovna základních tříd poskytující přístup k datovým zdrojům (kolekce, ADO.NET), síťovou komunikaci, numerické algoritmy, podporu pro uživatelské rozhraní (ASP.NET, Windows Forms), a další. V dalších verzích (3.0 a výš) pak přibývaly další součástí, které stavějí na základech předchozích verzí.

[2] CLI (Common Language Infrastructure) je popsáno v mezinárodním standardu ECMA-335 [4] a definuje kromě samotného IL jazyka také např. formát souborů Portable Executable (PE) pro binární moduly obsahující IL kód.

[3] GC (Garbage Collector) – automatická správa paměti, odpadá tak manuální alokace a dealokace paměti.

V současné době je aktuální verze 4.0. K vývoji aplikací pro prostředí .NET Framework poskytuje Microsoft vývojové prostředí Visual Studio (aktuálně ve verzi 2010).

2.2.2 Datové úložiště

Informační systémy běžně produkují a spravují velké množství dat. Jejich součástí tedy musí být nějaké úložiště dat. Nejběžnějším úložištěm je některá z relačních databází. U zamýšlených IS pro specifické účely to však nemusí platit vždy a jako úložiště může postačit např. XML soubor. Žádné striktní omezení na konkrétní relační databázi nebo jiný typ úložiště není vhodné. Většina běžně dodávaných ERP systémů podporuje alespoň několik relačních databází, případně i jiné typy úložišť. Nejčastěji se jedná o produkty Microsoft SQL Server, Oracle Database nebo PostgreSQL. Podpora alespoň některého z nich bude podmínka pro další uvedené nástroje.

2.3 Požadavky na framework

V této kapitole budou uvedeny stanovené požadavky na hledaný framework. Pro přehlednost jsou rozděleny do skupin podle oblasti, k níž se vztahují. U každého požadavku je rovněž uvedeno krátké odůvodnění.

Obecné vlastnosti a cíle

Tyto požadavky vyplývají z předchozího textu v kapitolách 1 a 2, proto uvádím pro úplnost pouze jejich výčet.

- Podstatně urychlit a tím i snížit náklady na vývoj především zakázkových IS pomáhajících ve specifických podnikových procesech. Využití může ale najít i při vývoji obecných podnikových IS (např. ERP systémů).
- Největší užitek přinese programátorům IS, protože nebudou muset řešit rutinní operace, které nijak nesouvisí s logikou vytvářeného IS.
- Automatická implementace základních CRUD (create, read, update, delete) operací.
- Musí zvládat pracovat s poměrně velkým množstvím dat stejně jako běžné ERP systémy (řádově statisíce záznamů v modulu).
- Definování architektury celé aplikace a vhodných postupů pro návrh.
- Modulární, vícevrstvá architektura.
- Vysoká přizpůsobitelnost svázaná s předchozím požadavkem. Celý framework musí umožnit doplnění vlastních rozšíření pro řešení speciálních požadavků. Proto by měl být vysoce modulární, aby bylo možné jednotlivé moduly v případě potřeby nahradit vlastní implementací.

Přístup k datovým zdrojům

- Framework musí podporovat různé zdroje dat. Jak již bylo zmíněno v kapitole 2.2.2, většina IS podporuje více zdrojů dat. Zpravidla se jedná o různé relační databáze. Vzhledem k účelu frameworku je ale nutné, aby byla podpora datových zdrojů širší. Například i zmíněné XML soubory nebo dočasné čistě paměťové úložiště. Přitom pro vyšší vrstvy frameworku by mělo být samotné úložiště dat abstrahováno tak, aby bylo možné se všemi zdroji dat pracovat jednotně a v případě výměny fyzického úložiště nebylo nutné provádět žádné rozsáhlé změny ve vyšších vrstvách.

- Pro pravděpodobně nejvíce využívaný zdroj dat, relační databáze, by ale mělo být umožněno spouštění vlastních SQL příkazů pro specifické účely. Dále framework musí podporovat pokročilejší prvky relační databáze, jako je volání uložených procedur nebo databázových pohledů. Důvodem je možnost umístění části logiky aplikace přímo do databáze.

- Musí podporovat atomické provádění operací měnící data – transakce.

- Mělo by být umožněno na základě datového modelu aplikace vytvořit a inicializovat zdroj dat. Např. v případě relační databáze je tím myšleno vytvoření databáze s odpovídajícím schématem.

Datový model aplikace

Datovým modelem aplikace je myšlena definice datových objektů použitých v aplikaci a vazby mezi nimi. Prostřednictvím těchto objektů se pak přistupuje k datovému úložišti a provádí se v něm změny. Datový model bývá často reprezentován formou různých konfiguračních souborů. Součástí těchto definic je seznam jednotlivých objektů, jejich atributů a vazeb mezi jednotlivými objekty. Z konfiguračního souboru je pak zpravidla přímo generován kód tříd reprezentující dané objekty. Propracovaný systém datových modelů má například *ADO.NET Entity Framework* [6]. Ten využívá 2 modelů – konceptuální, se kterým pracuje aplikace, a datový, který koresponduje se strukturou dat v úložišti. Mezi těmito 2 modely pak definuje vzájemné mapování. Definici modelů lze provést ručně v grafickém editoru nebo importem z již existující databáze. Obdobný přístup používá množství dalších ORM.

Alternativním přístupem může být definice modelu přímo ve zdrojovém kódu. Objekty se definují jako třídy, atributy objektů jako atributy třídy. Tento přístup navíc umožňuje snadné doplnění další logiky přímo do definované třídy. Problém je ale s grafickým znázorněním a editací takového modelu. Uvedený přístup využívá např. MM .NET Application Framework (kapitola 3.5).

Požadavky na datový model:

- Možnost definovat objekty a jejich atributy, které mohou být různého typu a mohou mít předdefinovaná omezení (povinné pole, maximální délky, pole pouze pro čtení, apod.). Dále možnost definice vazeb mezi datovými objekty. To vše pokud možno jednoduchým deklarativním způsobem s nutností psaní žádného nebo minima kódu.

- Možnost definice vazby typu master-detail. Toto by mělo také zahrnovat podporu pro fyzické ukládání změn v podřízených modulech až při uložení nadřízeného modulu. Takové chování je typické např. pro editaci faktury. Položky faktury lze libovolně měnit, vkládat, odstraňovat, aby se mohla interaktivně přepočítávat celková cena faktury. Veškeré změny v položkách se ale fyzicky uloží až při uložení celé nadřízené faktury. Tím je umožněno v případě omylu veškeré změny v položkách faktury zrušit a vrátit se k původní uložené verzi.

- Může být vyžadována současná práce s objekty z různých datových zdrojů. Vazby tedy mohou být definovány i napříč různými zdroji dat.

Aplikační logika

V oblasti aplikační logiky musí framework především vést programátora k tomu, aby umístil veškerou aplikační logiku do jednoho centralizovaného místa k tomu určenému. Tím velmi přispěje k čistotě architektury aplikace a předejde poměrně časté chybě, kdy jsou části aplikační logiky umístěny například přímo v kódu obsluhy události stisknutí tlačítka.

- Centralizace veškeré aplikační logiky do jednoho místa (vrstvy) aplikace. Framework by měl k tomuto účelu poskytnout vhodné nástroje a události. Rovněž by ve stejném místě měl být umístěn doplňující kód speciálních operací, které jsou následně volány z uživatelského rozhraní aplikace. Hlavním důvodem tohoto požadavku je možnost opětovného využití celé aplikační logiky v jiné aplikaci nebo v případě změny uživatelského rozhraní.

- Framework musí řešit některé základní opakující se činnosti. Jedná se především o řešení konkurentní editace záznamů v úložišti dat. Tento problém lze snadno přehlídnout během implementace i při prvotním testování, protože danou funkčnost testuje zpravidla pouze jeden uživatel. V reálných situacích ale může nastat případ, kdy jeden záznam chtějí v jednom okamžiku upravovat dva a více uživatelů. Systém musí umožnit editovat záznam pouze jednomu z nich. Dalším požadavkem je výchozí implementace správy uživatelů, jejich autentizace a kontroly uživatelských oprávnění k definovaným operacím.

Uživatelské rozhraní

Uživatelské rozhraní aplikace má sloužit pouze jako vizuální obálka aplikační logiky a datových objektů zprostředkující komunikaci s uživatelem systému. V případě klasického přístupu je však mnohdy k jeho tvorbě zapotřebí více úsilí, než je nutné věnovat logice aplikace. Uživatelské rozhraní je tak součást frameworku, která vyžaduje největší úroveň zautomatizování. Od toho se odvíjejí i následující požadavky.

- Uživatelské rozhraní by mělo být poskytováno zcela automaticky na základě datového modelu aplikace. To zahrnuje generování zobrazovacích a editačních formulářů, implementaci rozhraní pro vkládání, editaci a odstraňování záznamů.

- Předchozí požadavek ale nevylučuje ponechání možnosti vlastního návrhu formuláře. Automaticky generované formuláře totiž mohou být v některých případech nepřehledné a tak je vhodné programátorovi v tomto ohledu ponechat volnost.

- V případě ručního návrhu formuláře musí framework i tuto činnost zjednodušit. Především se jedná o snadnou definici datových vazeb mezi komponentami formuláře a atributy datových objektů. Součástí by tedy měly být i jednoduše použitelné komponenty, pomocí nichž lze vizuálně navrhnout vlastní formuláře. Systém komponent musí ale umožňovat použití dalších doplňujících komponent.

- Musí definovat základní šablonu uživatelského rozhraní (např. hlavní okno aplikace, metody různých systémových hlášení), dále pak mechanismus pro specifikaci modulů systému, jejich formulářů, povolených akcí a dalších vlastností. *Moduly systému* lze chápat jako logické celky IS složené ze skupiny datových objektů a formulářů. Příkladem takových modulů můžou být zákazníci, objednávky nebo zboží.

- Užitečná je rovněž možnost volby typu uživatelského rozhraní. Mělo by být možné zvolit si z několika druhů uživatelského rozhraní. Často je požadováno, aby k jednomu IS existovalo klasické grafické uživatelské rozhraní v podobě spustitelného souboru. Rovněž ale může existovat např. webové rozhraní pro vzdálenou práci se systémem. V současné době se rozšiřuje i podpora mobilních zařízení a aplikací pro ně. Framework, který nabídne více možností pro uživatelské rozhraní, včetně snadného přechodu na jiný typ rozhraní, tak bude určitě žádanější. Samozřejmostí je pak opětovné využití všech nižších vrstev aplikace pro různá uživatelská rozhraní.

3 Dostupné nástroje a frameworky

Tato kapitola obsahuje seznam a stručný popis některých dostupných frameworků, nástrojů a podobných produktů, které jsou určeny pro snazší vytváření IS. Zaměřuje se především na zhodnocení nástroje z pohledu požadavků stanovených v kapitole 2, hledá výhody a nevýhody jednotlivých přístupů.

3.1 Implementace „od nuly“

Tento postup nevyužívá žádný nástroj nebo framework pro zjednodušení tvorby podnikových IS. Často se také lze setkat s anglickým označením „from scratch“. I když tento postup žádný nástroj nevyužívá, zaslouží si také pozornost minimálně z důvodu srovnání s ostatními.

Stručné představení eXpressApp Framework v dokumentaci k tomuto produktu [7] výstižně popisuje výhody a nevýhody takového přístupu. K nevýhodám patří zejména tyto:

- Vytvoření i nejjednodušší aplikace sloužící k zobrazování a změně údajů zabere spoustu času. Vývojáři takové aplikace musí analyzovat a implementovat každou její součást.

- Velké množství úsilí a prostředků stojí i testování aplikace. Vzhledem k tomu, že v každém software jsou nějaké chyby a množství napsané kódu je v tomto případě obrovské, mnohonásobně se také zvětšuje počet chyb v aplikaci. Tento stav je možné omezit opětovným použitím dříve vyvinutých a otestovaných komponent.

- Je těžké udržovat a rozšiřovat takovou aplikaci. Komponenty aplikace většinou nebývají připraveny na větší změny, je mnohem těžší se ve zdrojovém kódu aplikace vyznat, drobná změna (např. datového typu nějakého datového pole) znamená úpravy ve více místech kódu a všech formulářích zobrazujících toto pole. Takové úpravy jsou pak dalším zdrojem nových chyb.

Tento přístup má však i své výhody. Mezi ně patří:

- Každá nejmenší součást je (nebo může být) pod přímou kontrolou vývojářů. Díky tomu, že součást navrhli a vytvořili, také důkladně chápou její účel a funkci a snadno ji upraví.

- Vývojáři mohou vysoce optimalizovat součástí aplikace pro její specifické použití. V případě univerzálních frameworků a nástrojů to není možné.

- Aplikace se nemusí držet pravidel nějakého externího nástroje a může si řešit veškeré operace specifickým způsobem.

Z výše uvedeného vyplývá, že implementace „od nuly“ je vhodná pouze v případech velmi specifických požadavků nebo velmi vysokých nároků na optimalizaci aplikace. Metoda je ovšem nejpracnější a nejnákladnější.

3.2 Microsoft Visual Studio LightSwitch

LightSwitch je nová edice vývojového prostředí Visual Studio, která zahrnuje speciální typy Visual C# a Visual Basic .NET projektů a nové průvodce a designéry [8]. Umožňuje snadno vytvářet tzv. data-centric aplikace[4] formou jednoduchých grafických editorů datového modelu a obrazovek aplikace. Tento nástroj staví na řadě již známých technologií .NET Frameworku (např. LINQ [3], ADO.NET Entity Framework [6], Silverlight [9], WCF RIA Services [20]) a využívá je k vytvoření abstraktní vrstvy optimalizované pro správu dat.

Výrobce:	Microsoft
Domovská stránka:	http://www.microsoft.com/visualstudio/en-us/lightswitch
Dostupnost:	Komerční nástroj, dostupné různé edice, 30-ti denní trial verze zdarma.
Výhody:	• Vytvoření aplikace bez napsání jediného řádku kódu • Přímá podpora cloudového řešení (Windows Azure) a 3 vrstvé architektury (aplikační server) • Bezproblémový, odladěný nástroj • Dostupnost rozšíření poskytovaných třetími stranami
Nevýhody:	• Omezená možnost přizpůsobení formulářů • Složitý způsob vytváření vlastních rozšíření nástroje • Chybí přímá podpora pro práci se vztahy master-detail • Nepřehledná práce s obrazovkami a doplňováním kódu do různých událostí • Pouze jeden typ uživatelského rozhraní – Silverlight

Celkově se jedná o propracovaný a užitečný nástroj, který je ale primárně určen pro vývoj jednodušších aplikací zaměřujících se převážně na základní práci s daty. Pro vývoj aplikací se složitější business logikou se příliš nehodí. Uplatnění pravděpodobně nejčastěji najde v podnikových IT odděleních.

3.3 eXpressApp Framework

Společnost Developer Express Inc. už dlouhou dobu vyvíjí velkou sadu různých komponent pro všechny technologie platformy .NET. Součástí plného balíku obsahujícího kolem 350 komponent je i eXpressApp Framework (XAF). Na rozdíl od předchozího se jedná opravdu o framework, jehož základem je ověřené ORM s názvem *eXpress Persistent Objects* (XPO) [10]. Další stěžejní součástí je *Application Model*, který obsahuje všechny potřebné informace pro automatické generování uživatelského rozhraní. Application model je automaticky generován ze zdrojového kódu objektů business vrstvy a díky integraci s Visual Studiem ho lze upravovat v grafickém editoru. Více podrobností v [7].

[4] Aplikace primárně zaměřené na vytváření a správu velkého množství dat.

Výrobce:	Developer Express Inc.
Domovská stránka:	http://www.devexpress.com/Products/NET/Application_Framework/
Dostupnost:	Komerční nástroj, 30-ti denní trial verze zdarma.
Výhody:	• Propracovaný framework zabírající všechny oblasti požadavků definované v kapitole 2 • Přímá podpora 2 typů uživatelských rozhraní vystavěných nad společnou vrstvou logiky aplikace (Windows Forms a ASP.NET) • Připravené vzory obecných datových objektů • Součástí je i podpora tiskových sestav a základních datových analytických funkcí • Podpora všech běžně dostupných relačních databází
Nevýhody:	• Vysoká cena a nutnost pořídit celý balík s některými nepotřebnými komponentami • Omezené možnosti přizpůsobení formulářů • Chybí přímá podpora pro práci se vztahy master-detail • Nepodporuje moderní uživatelská rozhraní (WPF, Silverlight) • Nižší výkon pravděpodobně způsobený celkovou univerzálností • Neumožňuje používat zároveň více různých datových zdrojů

XAF je velice propracovaný nástroj, který splňuje většinu nejpodstatnějších požadavků z kapitoly 2. Využití najde u obsáhlých složitějších aplikací. Nevýhodou je ale vysoká pořizovací cena a také celková míra abstrakce přístupu k datovým zdrojům, která se projevuje i na celkovém výkonu výsledné aplikace.

3.4 Cuberry

Cuberry je základní platforma pro modulární tvorbu IS. Aplikace se skládá ze samostatných modulů obsahujících jak logiku, tak uživatelské rozhraní. Lze je jednoduše přidávat nebo aktualizovat bez dalších úprav zbytku aplikace. Cuberry obsahuje základní moduly pro správu uživatelů, jejich oprávnění a dalších modulů. Staví na moderních technologiích platformy .NET Framework – ADO.NET Entity Framework [6] a WPF [14]. Více informací v [11].

Výrobce:	Peacequare International
Domovská stránka:	http://www.cuberry.net
Dostupnost:	GNU General Public License
Výhody:	• Licenční politika, dostupnost kompletních zdrojových kódů • Modulární stavba • Podpora master-detail vztahů
Nevýhody:	• Pouze jeden typ uživatelského rozhraní (WPF)

- Nepodporuje automatické vytváření uživatelského rozhraní – všechny dialogy aplikace je nutné vytvořit ručně
- Izolace aplikačních modulů, správa oprávnění jen na úrovni modulů
- Podporuje pouze databázi Microsoft SQL Server
- Příliš nezjednodušuje práci

Jednoznačnou výhodou je licenční politika a dostupnost zdrojových kódů. Kvůli nutnosti ručně vytvářet uživatelské rozhraní, business objekty i datové modely ADO.NET Entity Frameworku příliš práci nezjednodušuje.

3.5 MM .NET Application Framework

Jedná se o aplikační framework, který napomáhá k tvorbě správně navržených a výkonných aplikací [12]. Podporuje celou řadu technologií pro uživatelská rozhraní (WPF [14], Windows Forms, ASP.NET, Silverlight [9]). Přístup k datovým zdrojům zprostředkovává buď vlastní technologie *MM .NET entity objects* nebo ADO.NET Entity Framework [6]. Oba způsoby lze kombinovat [13]. Uživatelské rozhraní je nutné vytvářet vždy ručně. Datový model aplikace je definován kódem business objektů a lze jej pomocí průvodce vygenerovat z existující relační databáze. Více podrobností o tomto frameworku v [13] a [12].

Výrobce: Oak Leaf Enterprises Inc.

Domovská stránka: http://www.oakleafsd.com/pgProducts_mmnet.htm

Dostupnost: Komerční produkt, demoverze k vyzkoušení na vyžádání

Výhody:
- Široká podpora uživatelských rozhraní a dalších technologií
- Rozsáhlá dokumentace včetně návodů a vzorů pro dodržení správného návrhu aplikace [13]
- Oddělení aplikační logiky od uživatelského rozhraní

Nevýhody:
- Nepodporuje automatické vytváření uživatelského rozhraní
- Zdroje dat mohou být pouze relační databáze
- Komplikované ruční vytváření business objektů (bez použití generátoru z existující databáze)

MM .NET Application Framework je propracovaný produkt, který umožní vytváření rozsáhlých složitých aplikací. Významným nedostatkem je ale nutnost ručního návrhu všech formulářů aplikace, což vzhledem k možnostem vytvoření různých typů uživatelských rozhraní znamená navrhovat formuláře pro každou technologii znovu.

4 Návrh vlastního frameworku

V předchozí kapitole byly stručně popsány výhody a nevýhody některých dostupných nástrojů. I když některé z nich splňují téměř všechny požadavky stanovené v kapitole 2, tak stále existuje prostor pro vylepšení. Především jde o odstranění uvedených nevýhod. V této kapitole bude uveden návrh vlastního řešení takového frameworku, který bude splňovat všechny požadavky z kapitoly 2.

4.1 Architektura

Prvním krokem je návrh celkové architektury frameworku a výsledné aplikace na něm založené. V základních rysech se lze inspirovat u existujících řešení uvedených v kapitole 3, především u XAF [7] nebo MM .NET Framework [13], které rovněž splňují nejvíce stanovených požadavků. Základem jsou 4 vrstvy:

1) Fyzické úložiště, které slouží k perzistentnímu uchovávání dat. Např. relační databáze nebo XML soubor (podrobnosti v kapitole 2.3)

2) Vrstva datového úložiště – zpřístupňuje aplikaci data z fyzického úložiště a zajišťuje tzv. objektově-relační mapování (ORM).

3) Logika aplikace – slouží k definici podnikových pravidel a logiky celé aplikace. Tato vrstva má být čím jak nejvíc nezávislá na vrstvě uživatelského rozhraní, aby mohla být snadno opětovně použita v jiné aplikaci nebo pro jiný typ uživatelského rozhraní [15].

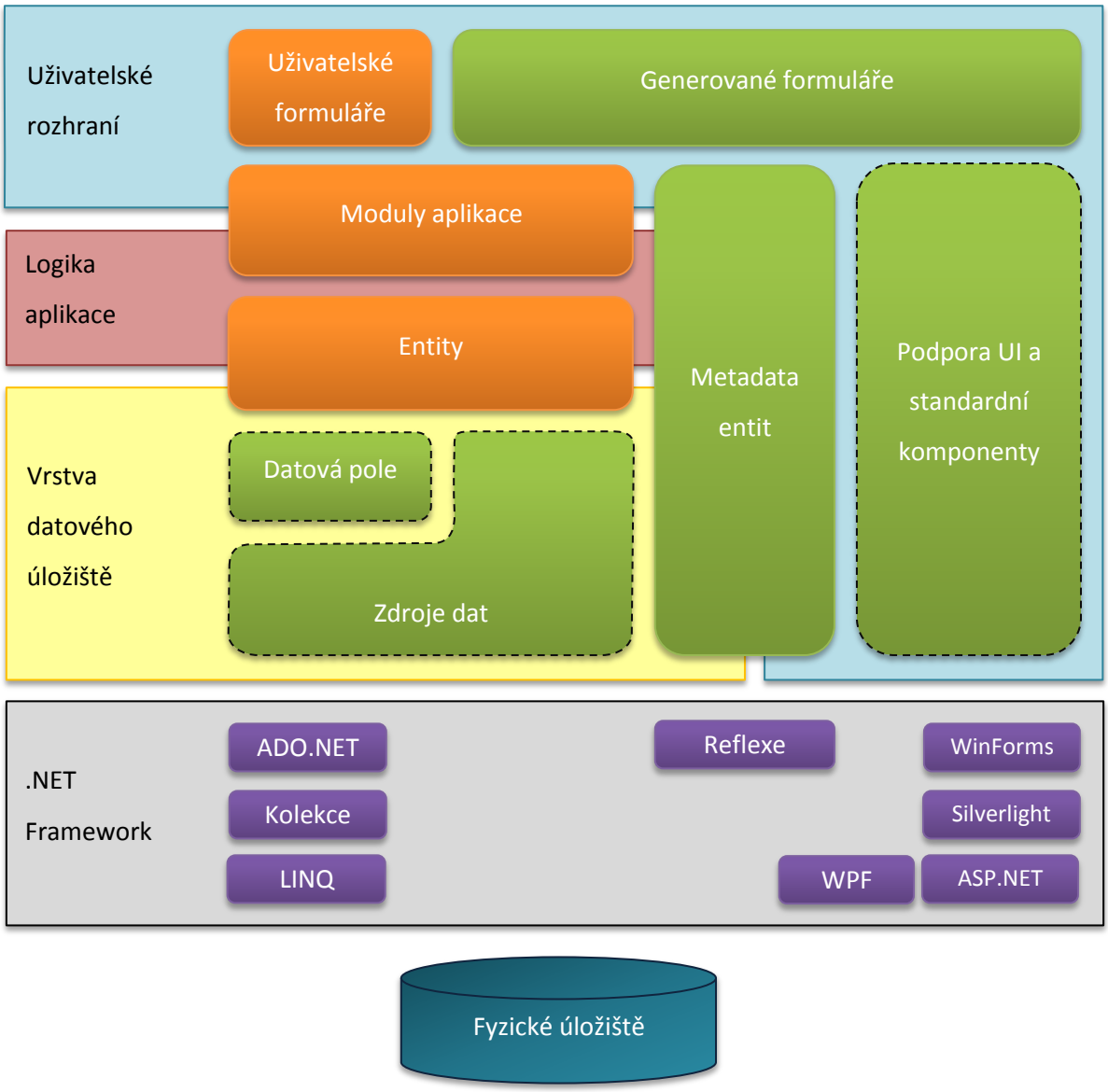
4) Uživatelské rozhraní, které pouze zprostředkovává komunikaci mezi uživatelem a aplikací.

Uvedené vrstvy jsou patrné na obrázku 4.1, který navíc také zobrazuje další členění vrstev. Zelenou barvou jsou zobrazeny součásti samotného frameworku, oranžovou barvou součásti specifické pro konkrétní aplikaci. Některé zelené bloky jsou ohraničeny přerušovanou čárou, která naznačuje, že daný blok má být uživatelsky rozšiřitelný. Tím je téměř zcela pokryt požadavek na rozšiřitelnost frameworku. Obrázek dále znázorňuje vrstvu .NET Framework, která slouží k ilustraci technologií, jež mohou vyšší vrstvy využívat.

Jádrem architektury jsou bloky s názvem Entity a Metadata entit. Kolem nich jsou postaveny všechny další bloky, které jejich služby využívají ke své činnosti.

Entity

Jedná se o datové objekty modelující reálné objekty, resp. záznamy o nich uložené ve fyzickém úložišti. Tyto objekty jsou vždy specifické pro danou aplikaci. Framework však může obsahovat



Obrázek 4.1: Návrh architektury frameworku

i některé připravené entity modelující standardní prvky systému (např. uživatele systému). Entity tedy definují objekty a jejich atributy a metody, s nimiž daná aplikace pracuje.

Metadata entit

Metadata jsou sestavována na základě struktury entit. Díky jejich existenci mohou ostatní bloky snadno zjišťovat informace o entitách, jejich datových polích a dalších vlastnostech. Stávají se tak základním zdrojem informací o struktuře dat dané aplikace. Dohromady s entitami tvoří datový model aplikace (viz kapitola 2.3) a jsou podobná např. Application Model u XAF (kapitola 3.3). Na rozdíl od XAF ale tyto informace nemusí být extrahovány ze zdrojových souborů pomocí dalšího nástroje a následně ukládány ve formě XML souboru. Platforma .NET totiž nabízí vestavěné prostředky pro prozkoumávání libovolných datových typů za běhu. Tento postup se označuje jako *reflexe* [16] a potřebné třídy se nacházejí v oboru názvu

`System.Reflection`. Rovněž lze ve značné míře využít možnosti doplňovat *atributy* k jednotilvým konstrukcím programovacích jazyků platformy .NET. Tyto atributy jsou speciální třídy odvozené od třídy `System.Attribute`. Během překladu jsou umístěny do výsledného sestavení (assembly) jako metadata prvků kódu a jsou rovněž přístupné přes typové informace získané pomocí reflexe [16]. Díky tomu je možné získávat metadata od běhového prostředí CLR. Metadata entit lze získávat přímo za běhu aplikace z existujících objektů. Odpadá tak nutnost definovat nějaký speciální formát pro ukládání metadat datového modelu a vytvářet další nástroje pro jeho generování a editaci.

Zdroje dat

Tento blok zajišťuje přístup k různým fyzickým úložištím. Provádí základní CRUD operace na nejnižší úrovni. Ke své činnosti může používat některé technologie dostupné jako součást platformy .NET (např. ADO.NET pro přístup k relačním databázím). Zároveň je tento blok zodpovědný za správnou inicializaci nebo vytvoření struktury v úložišti a to na základě informací dostupných z metadat entit. Pro každý typ úložiště musí existovat odpovídající objekt. Rovněž zde musí existovat možnost vytvoření a zařazení vlastního objektu pro přístup k dalšímu typu úložiště dat.

Datová pole

Datová pole slouží k uchovávání hodnot atributů entit. I když by bylo možné pro jejich reprezentaci využít elementární datové typy prostředí .NET, existuje flexibilnější řešení, při kterém mohou být typy atributů složitější. Jednak se tím přenáší zodpovědnost za správnou reprezentaci a případně konverzi hodnoty získané z úložiště dat na informaci poskytovanou entitou vyšším vrstvám, ale také se otevírá možnost pro další budoucí rozšiřování vlastností frameworku. Aplikace pak může pracovat se specifickými datovými typy (např. reprezentace různých prostorových dat uložených v databázi). Datová pole tedy tvoří hlavní spojovací článek mezi blokem entit a zdrojů dat.

Podpora UI a standardní komponenty

Součástí frameworku musí být podpora uživatelského rozhraní a soubor základních komponent a formulářů. Zde musí být možné doplňovat především vlastní nové komponenty, ale také musí být snadné připravit podporu pro zcela nový typ uživatelského rozhraní. Jedná se tedy především o podpůrnou součást pro další bloky – Generované formuláře a Uživatelské formuláře.

Generované formuláře

Blok znázorňuje důležitou součást frameworku – mechanismus automatického generování formulářů pro zvolený typ uživatelského prostředí. Tyto formuláře budou vytvářeny dynamicky za běhu aplikace na základě informací z metadat entit.

Uživatelské formuláře

Jde o speciální formuláře navrhované uživatelem v případě požadavku na složitý layout nebo při použití nějaké speciální komponenty apod.

Moduly aplikace

Tvoří rozhraní mezi čistě logickou nezávislou částí a konkrétním použitým uživatelským rozhraním. Dále definují klíčové prvky pro uživatelské rozhraní – hlavní moduly, které musí být dostupné z hlavního formuláře uživatelského rozhraní, uživatelské formuláře a jejich napojení na entity. Dále pak obsahují *kontroléry*, jejichž účelem je implementovat obsluhy událostí a akcí vzniklých v uživatelském rozhraní a tak doplňovat aplikační logiku. Aby byl dodržen požadavek na minimalizaci množství kódu umístěného ve formulářích, musí být kontroléry ještě součástí vrstvy aplikační logiky. Díky tomu jsou společné pro různé typy uživatelského rozhraní.

Základní architektura tak přispívá k dodržování důsledného oddělení logiky aplikace a uživatelského rozhraní. Aby to bylo možné, je ještě potřeba dodržet jeden architektonický požadavek. Celý framework (i výsledná aplikace) musí být rozdělena do několika sestavení (assembly). Jedno sestavení bude obsahovat bloky související pouze s logikou aplikace (zdroje dat, datová pole, metadata entit, univerzální část bloku entit a modulů), další sestavení budou obsahovat specifické implementace bloků pro podporu UI a generovaných formulářů určené pro daný typ uživatelského rozhraní. Obdobné rozdělení platí v případě uživatelských částí. Konkrétní entity a kontroléry budou v jednom sestavení, uživatelské formuláře a ostatní části přímo související s konkrétním typem uživatelského rozhraní v dalších sestaveních.

4.2 Návrh klíčových částí frameworku

V této kapitole bude popsán konkrétní podrobný návrh klíčových částí frameworku z jednotlivých výše uvedených vrstev.

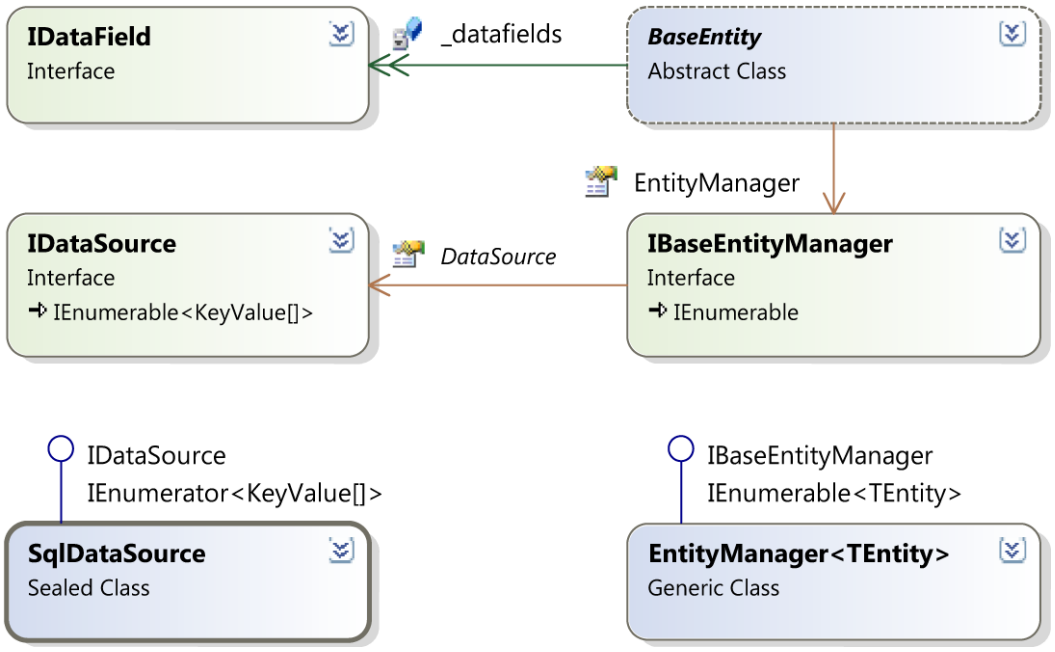
Jedním ze základních požadavků na framework stanovených v kapitole 2.3 je vysoká míra rozšiřitelnosti jednotlivých částí. Ten je nutné mít při návrhu stále na paměti. Prvním krokem bude stanovení základních klíčových objektů z vrstvy datového úložiště a logiky aplikace (obrázek 4.1), s nimiž bude framework interně pracovat.

Entity

Nejdůležitější objekty jsou samotné *entity*. Konkrétní entity se budou sice tvořit až pro danou aplikaci, framework by ale měl poskytovat předka všech entit, který bude definovat společné základní rozhraní. Jedná se například o metody pro ukládání, odstraňování, validaci hodnot, kontrolu oprávnění, zjištění aktuálního stavu entity atd. Dále by měl obsahovat některé virtuální metody, do nichž pak bude možné vkládat kód specifický pro konkrétní implementovanou

entitu. Tento předek se společnými vlastnostmi všech entit bude reprezentován abstraktní třídou `BaseEntity`.

Framework musí rovněž počítat s důležitým konceptem dědičnosti entit. Generalizace resp. specializace jsou často používané postupy při návrhu informačních systémů. Příklad generalizace lze nalézt i u jedné ze základních entit v každém systému – uživatel. O uživatelích je zpravidla nutné evidovat minimálně jejich jméno, příjmení, email, adresu apod. Stejné vlastnosti ale může mít i kontaktní osoba. Často tak dochází ke generalizaci společných vlastností do abstraktní třídy (entity) osoba. Obdobných případů existuje velké množství, proto je důležité, aby framework s tímto konceptem počítal. Z podstaty návrhu entit se již s dědičností počítá, jen je nutné, aby tuto situaci správně interpretovaly i další vrstvy, především vrstva metadat (kapitola 4.4).



Obrázek 4.2: Diagram tříd klíčových částí frameworku

Datová pole

Každá konkrétní entita má nějaké vlastnosti, které jsou uchovávány v jejich *datových polích*. Již v předchozí kapitole byly zmíněny dva možné přístupy v implementaci a také zvolený přístup pro potřeby frameworku. Datová pole budou reprezentována speciálními objekty. Na rozdíl od obdobných produktů, kdy jsou většinou datová pole reprezentována pouze základními primitivními datovými typy programovacího jazyka, se zde předpokládá, že budou obsahovat i část logiky, jako například základní validace, konverze na různé jiné reprezentace nebo formátování hodnoty jako řetězce zobrazovaného v uživatelském rozhraní aplikace. Základní činností datového pole je:

- Schopnost převodu jeho hodnoty na reprezentaci vhodnou pro fyzické uložení v úložišti dat (a rovněž zpětný postup).

- Spravovat základní údaje o svém stavu.

Druhý bod souvisí především s monitorováním změny hodnoty v datovém poli. Pokud dojde k nastavení nové hodnoty do pole, mělo by o tom vědět a díky tomu pak může tuto informaci získat i celá entita, což je důležité při jejím ukládání. Zároveň v návaznosti na uživatelské rozhraní bude pro koncového uživatele aplikace určité příjemné, když mu bude např. textové pole vizuálně signalizovat, že se jeho hodnota změnila oproti původní uložené hodnotě. Rovněž by si samo datové pole mělo uchovávat informaci o tom, zda je jeho hodnota pouze pro čtení, a zda je její zadání povinné. Tyto dvě vlastnosti některé obdobné nástroje a frameworky řeší fixním nastavením na úrovni definice entit, což ale není nejlepší přístup. V reálných aplikacích často nastává problém, kdy nutnost vyplnit hodnotu některého pole nebo naopak nemožnost změnit jeho hodnotu vyplývá z aktuálního kontextu (stavu) a aktuálních hodnot jiných datových polí. Při fixním nastavení toho nelze docílit. Když ale tyto vlastnosti budou přímo v režii objektu reprezentujícího datové pole, bude možné je kdykoliv za běhu měnit pro konkrétní instanci entity a její stav. Podmínkou samozřejmě je, aby datové pole o změně těchto vlastností informovalo své okolí, resp. svázané komponenty uživatelského rozhraní.

Dalším logickým požadavkem je, aby existovala datová pole různých datových typů. Framework by měl dodávat sadu datových polí obecně použitelných, především tedy elementárních datových typů (číselných, řetězcových, pro datum a čas), ale zároveň musí poskytovat možnost tvorby vlastních datových polí spravujících složitější datové typy nebo i datové typy vlastní, specifické pro konkrétní aplikaci. Jak je patrné z obrázku 4.2, interně bude pracovat framework pouze s rozhraním datového pole `IDataField`, které je veřejné a poskytuje tak možnost vlastní implementace datového pole. Zároveň bude poskytovat řadu konkrétních implementací tohoto rozhraní. Vzhledem k množství společných vlastností jednotlivých typů datových polí bude vhodné společnou část logiky datového pole osamostatnit do zvláštní třídy. Tato třída by mohla být přístupna i mimo framework jako předek uživatelských implementací datových polí.

Důležitá je rovněž možnost vytváření tzv. *počítaných datových polí*, což jsou pole, jejichž hodnota není fyzicky ukládána do úložiště dat, ale stanovuje se na základě hodnot ostatních datových polí. Příkladem může být celková cena položky faktury (cena za kus × počet kusů). Takové datové pole ale musí pro okolní (vyšší) vrstvy vypadat naprosto stejně jako běžné datové pole. Nižší vrstvy jej naopak musí ignorovat. Podstatně se však liší v nutnosti definovat pro takové pole úsek kódu, v němž dochází k vyhodnocení jeho hodnoty. Bližší informace o datových polích a použitému způsobu jejich definice a implementace je uveden v kapitole 5.2.

Správce entit

Dalším důležitým prvkem je tzv. *správce entit* (entity manager) neboli jejich kolekce. To vychází z předpokladu, že entity reprezentují jednu konkrétní instanci nějakého reálného objektu určitého typu. V datovém úložišti pak většinou uchováváme informace o více různých

objektech tohoto typu, proto je s nimi nutné pracovat jako s kolekcí. Kromě správy kolekce řídí činnost vrstvy pro přístup ke zdroji dat, tedy inicializuje serializaci a deserializaci[5] entit a vydává požadavky na jejich načtení, resp. uložení nebo smazání.

Správce entit je na obrázku 4.2 reprezentován rozhraním `IBaseEntityManager`, které definuje základní operace, jež musí implementace poskytovat. Opět tak vzniká možnost pro přizpůsobení činností frameworku, tentokrát v podobě vlastní implementace správce entit. Rozhraní je ale poměrně obecné, proto bude existovat i hotová konkrétní implementace v podobě třídy `EntityManager<TEntity>`. Tento správce obsahuje všechny potřebné operace připravené pro obecné použití. Z deklarace je patrné, že se jedná o tzv. generickou třídu, která na rozdíl od `IBaseEntityManager` bude obsahovat silně typované rozhraní určené konkrétním typem entity zadaným v typovém parametru `TEntity`. Svou podstatou jde o obdobu generických kolekcí ze jmenného prostoru `System.Collections.Generic`. Jak uvádí např. [3], největší výhodou generických datových typů je zajištění větší typové bezpečnosti, protože na rozdíl od obecných kolekcí, jejichž prvky jsou typu `object`, zde má překladač přesnou informaci o datovém typu prvku kolekce a může tak množství problémů odhalit již při překladu. Podrobnější informace o generických kolekcích jsou v [17] a [3]. Třídu `EntityManager<TEntity>` lze navíc použít jako předka pro vlastní úpravy a doplnění nějaké rozšiřující logiky na úrovni celé kolekce daného typu entit. Pro tyto účely bude poskytovat sadu virtuálních metod, jejichž přepsání nebo doplnění bude ve většině případů dostačovat. Jen ve velmi specifických případech může dojít k implementaci zcela jiného správce entit podle rozhraní `IBaseEntityManager`.

Zdroj dat

Poslední klíčový prvek popisuje již výše kapitola 4.1 – jde o *zdroj dat*. Jeho základní činností je zpřístupnění nějakého fyzického úložiště vyšším vrstvám frameworku. Z dříve uvedených požadavků plyne, že taková úložiště mohou být různých druhů. Proto je opět definováno pouze rozhraní `IDataSource`, se kterým ostatní součásti komunikují. Tím je dosaženo požadované úrovně abstrakce fyzického způsobu ukládání dat a vyšší vrstvy tak pracují nezávisle na použitém zdroji dat. Objekty implementující rozhraní `IDataSource` představují vrstvu převodníků zpravidla dvou odlišných rozhraní. Jedná se tedy o typické použití návrhového vzoru *adaptér* [18]. Stěžejním problémem při návrhu tohoto rozhraní je určení vhodné reprezentace přenášených dat. Jako nejflexibilnější se jeví přenos dat v podobě kolekce dvojic (název hodnoty, hodnota). Název hodnoty reprezentovaný textovým řetězcem je v rámci kolekce unikátní a může se jednat například o název sloupce v SQL tabulce. Aby byl umožněn přenos libovolných hodnot, musí být druhá složka dvojice obecného typu `object`. Tím se zpřístupní možnost návrhu datového pole, jehož hodnota se bude do zdroje dat přenášet jako složitější datový typ (např. struktura) a zároveň vlastní implementace rozhraní `IDataSource` může hodnotu korektně zpracovat. Tento postup však vytváří přílišnou závislost mezi vrstvami

[5] Na serializaci a deserializaci se ve skutečnosti podílí všechny 4 klíčové prvky. Správce entit celou činnost řídí, entita musí mít možnost ovlivnit průběh své serializace/deserializace a datová pole jsou zodpovědná za vytvoření vhodné reprezentace své hodnoty pro fyzické uložení. Zdroj dat pak tuto kolekci hodnot fyzicky zapíše nebo přečte z úložiště.

frameworku, a proto by se hodnoty datových polí měly raději vždy transformovat na základní datové typy dostupné v prostředí .NET.

Obrázek 4.2 zobrazuje i pravděpodobně nejčastěji využívaný zdroj dat s názvem `SqlDataSource`, který slouží ke zpřístupnění hodnot z jedné tabulky SQL databáze. Jelikož se jedná o primární zamýšlené úložiště, musí být tento zdroj dat základní součástí frameworku. Zároveň musí být náležitě odladěn a optimalizován, protože z těchto hledisek jde o klíčový prvek. Důležitá je rovněž podpora různých databázových systémů. Produkované SQL dotazy tak musí primárně dodržovat platné standardy, aby je správně zpracovalo široké spektrum různých databázových systémů. Více k tomuto tématu v kapitole 5.3.

Shrnutí

Z informací uvedených v této kapitole plyne jednak stanovení klíčových částí frameworku, na které bude nutné klást při implementaci velkou pozornost, ale také základní *koncept rozdělení logiky výsledné aplikace* a její skládání z jednotlivých elementárních částí. Pro přehlednost uvádí klíčové zodpovědnosti tabulka 4.1.

Část frameworku	Zodpovědnosti a implementovaná logika
Datová pole	Spravují logiku změny hodnoty a notifikaci dalším vrstvám, základní vlastnosti datových polí (povinné zadání, pole jen pro čtení). Řeší potřebné konverze hodnot.
Entity	Řeší logiku na úrovni jedné entity. Validují hodnoty svých datových polí, nastavují jejich implicitní hodnoty, mění jejich vlastnosti. Spravují stav celé entity a svou serializaci/deserializaci.
Správce entit	Implementují část aplikační logiky pracující s celou kolekcí entit stejného typu.
Zdroj dat	Je to pouze adaptér mezi jednotným rozhraním vyšších vrstev a rozdílnými rozhraními různých typů fyzických úložišť dat. Neobsahuje žádné části aplikační logiky.

Tabulka 4.1: Koncept rozdělení zodpovědností mezi klíčové části frameworku

Uvedené rozdělení by měl zároveň dodržovat i uživatelský kód entit, a rovněž případných rozšíření nebo alternativních implementací některých částí.

4.3 Vztahy mezi entitami

Základní koncept popsaný v předchozí kapitole umožní plnohodnotnou práci s kolekcemi entit stejných typů. Reálné aplikace ale vyžadují možnost modelování a realizace různých vztahů a vazeb mezi entitami, především následujících dvou důležitých typů.

Vazba typu master-detail

Jedná se o klasickou vazbu typu 1:N, kde jedna entita vystupuje jako nadřízená a N entit jako podřízených této entitě. Příkladem tohoto vztahu mohou být například faktura a její položky. Většinou platí, že podřízené entity nemohou existovat samostatně. Téměř vždy však platí, že je mezi těmito typy entit definován nějaký vztah, resp. mapování hodnot odpovídajících si datových polí. Na úrovni frameworku bude tento vztah implementován jako vlastnost entity (dále označovaná jako *podřízený modul*), která bude typu správce entit podřízeného typu entity, společně s definicí mapování polí. To pak bude použito jako filtr v kolekci podřízených entit, aby vlastnost entity vracela pouze odpovídající údaje.

Tento postup je dostatečný pro čtení dat, ale v případě editace nadřízené entity se situace stává složitější. Je nutné zajistit, aby během ukládání nadřízené entity byly uloženy i všechny změny v podřízených entitách a při odstranění nadřízené entity byly odstraněny i všechny podřízené. Opačným směrem musí probíhat notifikace – o změně v libovolné podřízené entitě musí být informována nadřízená entita, která pak změní svůj stav. Zajištění uvedeného si vyžádá následující změny v doposud uvedeném konceptu:

- Entita musí uchovávat v jednoduše přístupné struktuře (například kolekci) seznam všech svých podřízených modulů.

- Správce entit musí mít v sobě uloženou informaci o své případné nadřízené entitě.

Do obrázku 4.2 tak přibydou další vazby mezi třídou `BaseEntity` a rozhraním `IBaseEntityManager`.

Další zásadní úprava plyne z častého přístupu k editaci podřízených entit především pomocí grafického uživatelského rozhraní aplikace. Například při úpravě faktury a vkládání nových položek nebo úpravě existujících jsou všechny změny v položkách fyzicky uloženy až tehdy, je-li uložena celá faktura. Pokud se uživatel rozhodne fakturu neukládat, všechny změny, které provedl v položkách, budou zapomenuty. To znamená, že podřízený správce entit musí umožňovat dočasně si zapamatovat provedené změny, tak aby je uživatel v uživatelském rozhraní viděl a teprve po uložení nadřízené entity promítnout tyto změny do fyzického úložiště. Správce entit tak bude mít dva režimy činnosti. Buď klasický, nebo tzv. *režim paměťových editací*. Další informace o implementaci tohoto režimu jsou uvedeny v kapitole 5.4.

Vazba typu odkaz

Druhý typ vazby lze většinou chápat jako inverzní vztah k předešlému. Často ale může existovat pouze jedna z nich. Například faktura musí být přiřazena k určitému zákazníkovi, entita reprezentující zákazníka ale nemusí mít vlastnost s kolekcí všech faktur vystavených na tohoto zákazníka. V analogii na relační databázové systémy lze tento typ vazby přirovnat k cizím klíčům. Systém musí při zadávání faktury umožnit výběr z existujících zákazníků. Obdobná operace bývá v různých informačních systémech natolik častá, že je nutné její přímou podporu zapracovat i do frameworku a to nezávisle na předchozím typu vazby. Na úrovni modelu dat spočívá podpora pouze v možnosti definice vazby, která je pak součástí metadat (více v kapitole 4.4). Je nutné uchovávat tyto informace:

- Typ zdrojové entity (např. faktura)
- Typ cílové entity (např. zákazník)
- Mapování datových polí (např. pole „IDZakaznika“ z faktury se mapuje na pole „ID“ ze zákazníka)
- Unikátní identifikátor vazby (pro její snadné použití)

Dále musí být zavedena podpora pro snadné získání odkazované entity. Tím je myšleno jednoduché zavolání metody, které se na vstup předá unikátní identifikátor vazby, na zdrojové entitě a ona následně vrátí odpovídající entitu cílovou. V případě dříve uvedeného příkladu se metoda bude volat na faktuře a vrátí instanci zákazníka. Deklarace metody tedy bude vypadat následovně:

```
public BaseEntity GetLinkedEntity(long linkID);
```

Případně lze využít generickou alternativu, jejíž výhodou je zaručení typové bezpečnosti (více o generických metodách v [3]):

```
public T GetLinkedEntity<T>(long linkID)
    where T : BaseEntity, new()
```

Největší úroveň podpory však tento typ vazby musí mít na úrovní uživatelského rozhraní, kde bude nutné připravit speciální komponentu sloužící k zobrazení výběru odkazované entity. Další podrobnosti o této komponentě naleznete v kapitole 4.6.1 věnované uživatelskému rozhraní.

4.4 Model metadat

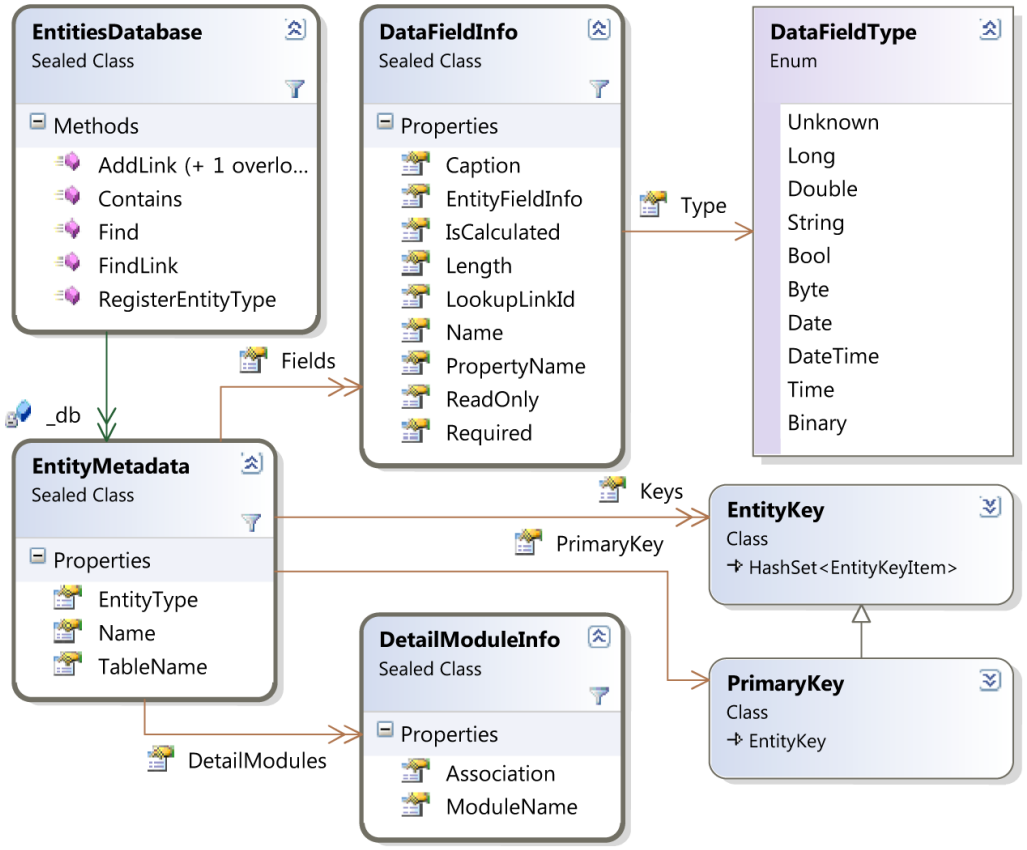
V této kapitole bude popsán model metadat, postupy pro jeho získávání a definici a dále některé způsoby využití. Největší význam mají metadata pro vrstvu uživatelského rozhraní.

Prakticky všechny výše uvedené klíčové elementy frameworku musí mít svá metadata, s nimiž pak mohou pracovat další části. Základní struktura metadat tak do značné míry kopíruje strukturu a vztahy mezi elementy jakými jsou datová pole, entity, kolekce podřízených entit nebo odkazy. Vzhledem k jednomu ze základních požadavků z kapitoly 2.3 na snadnou definici datového modelu je nutné čím jak nejvíce zjednodušit i definici metadat a propracovat jejich automatizované získávání.

Struktura metadat

Základní strukturu metadat s význačnými vlastnostmi zobrazuje diagram tříd na obrázku 4.3. Základním prvkem metadat je informace o celé entitě, kterou reprezentuje třída `EntityMetadata`. Ta obsahuje základní informace o entitě – její zobrazitelný název, seznam datových polí, seznam podřízených modulů (podřízené správce entit vztahu typu master-detail,

viz kapitola 4.3), definici *primárního klíče entity*[6] a případných doplňujících klíčů. *Klíče entit* slouží především k možnosti definovat seřazení ve správci entit, ale také je na jejich základě možné získat konkrétní instanci entity nebo omezit množinu načtených entit nějakou podmínkou. Vždy se jedná o seznam datových polí entity, která jsou využita k těmto účelům. Každá entita musí mít definován alespoň jeden klíč – primární klíč, obdobně jako tomu je ve většině případů u tabulek v relační databázi. Definice klíče je reprezentována třídou `EntityKey`. Pro snazší práci s klíči je možné si je pojmenovat. Metadata entity mohou obsahovat i další informace cílené pro konkrétní vrstvy frameworku (např. název tabulky v databázi pro zdroj dat typu `SqlDataSource`).



Obrázek 4.3: Diagram tříd metadat popisujících entity

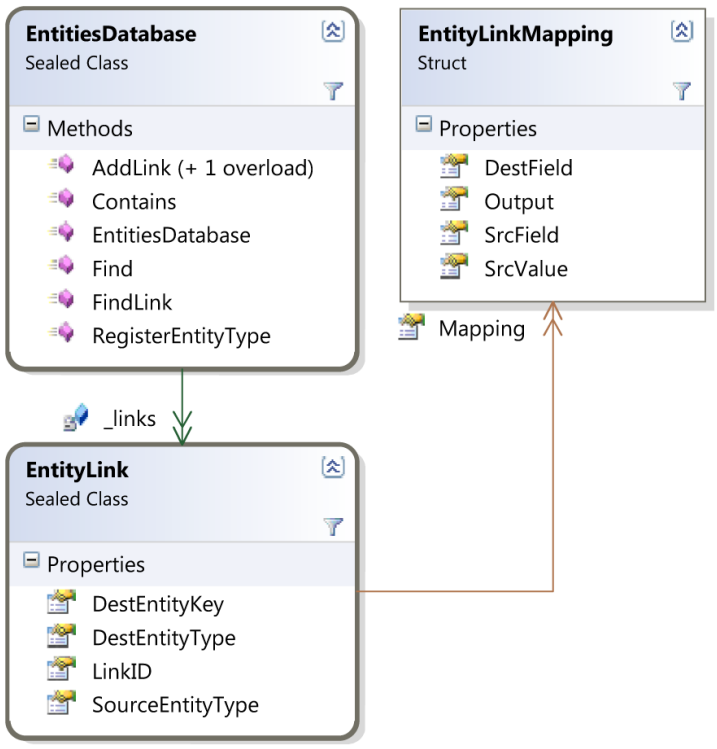
Dalším prvkem, o němž je nutné uchovávat množství informací, je datové pole. Odpovídající objekt metadat má název `DataFieldInfo`. Kolekci informací o datových polích uchovává ve své vlastnosti `Fields` objekt `EntityMetadata`. Jak je patrné i z obrázku, třída `DataFieldInfo` uchovává nejvíce údajů, protože datová pole mohou mít množství různých parametrů a nastavení. V prvé řadě musí být určen název datového pole a datový typ pro interní účely frameworku. Možné datové typy jsou uvedeny ve výčtu `DataFieldType`. Zároveň se jedná o přehled všech elementárních typů, s nimiž framework bude umět pracovat. Podle této hodnoty pak bude s datovým polem nakládat. Pokud například

[6] Primární klíč entity je seznam datových polí, která jednoznačně identifikují entitu daného typu.

dojde k implementaci vlastního datového pole zcela odlišného typu, jeho interní hodnota, která je ukládána do fyzického úložiště dat, se pravděpodobně bude stejně mapovat na některý z elementárních typů. V případě speciální hodnoty (například prostorová rozšíření některých databázových systémů) je možné využít hodnoty `DataFieldType.Unknown`. Vlastnost `IsCalculated` určuje, zda jde o počítané datové pole nebo klasické, vlastnosti `ReadOnly`, `Required` a `Length` ovlivňují výchozí parametry datového pole a `Caption` obsahuje čitelný název datového pole, který se zobrazuje uživateli například v podobě popisů komponent.

Třída `DetailModuleInfo` reprezentuje podřízený modul entity, resp. informace o něm. Obdobně jako třída `EntityMetadata` obsahuje kolekci informací o datových polích, má rovněž vlastnost `DetailModules`, která obsahuje kolekci informací o všech podřízených modulech. Nejdůležitější vlastností této třídy je `Association`, která definuje mapování datových polí mezi podřízenou a nadřízenou entitou.

Poslední třídou z obrázku 4.3 je `EntitiesDatabase` (databáze entit). Ta představuje nejvyšší úroveň v hierarchii metadat, protože je jejím úkolem udržovat databázi informací o entitách. Zároveň zodpovídá za registraci entit a korektní proces sestavení metadat o registrované entitě. Poskytuje sadu metod jednak k registraci, tak i k vyhledání informací v databázi. Dále pak spravuje údaje o zaregistrovaných odkazových vazbách mezi entitami. Odkazovou vazbu představuje objekt třídy `EntityLink` (obrázek 4.4), uchovávající základní údaje popsané v kapitole 4.3. Mapování datových polí je uloženo ve vlastnosti `Mapping`, která je kolekcí záznamů struktury typu `EntityLinkMapping`. Databáze entit má rovněž metody pro zaregistrování nové odkazové vazby a pro přístup k informacím o již zaregistrovaných vazbách. Jádrem celého modelu metadat je tak právě tato třída `EntitiesDatabase`.

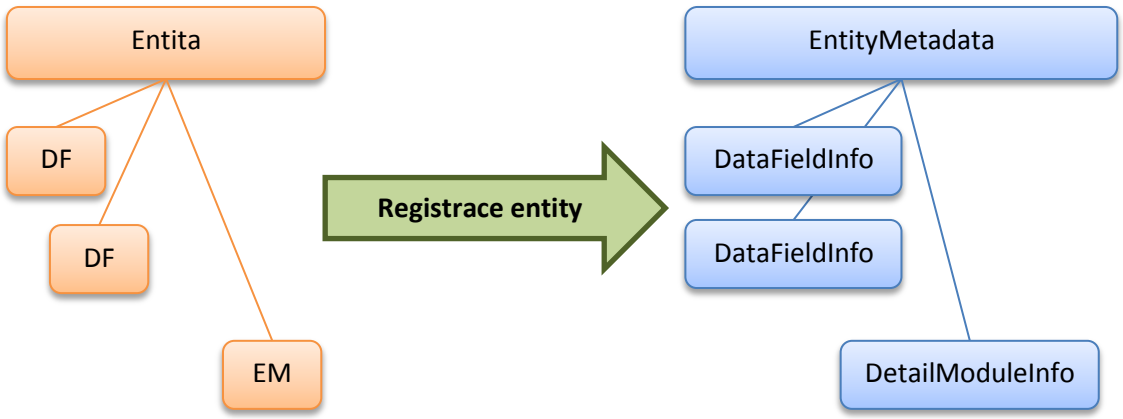


Obrázek 4.4: Diagram tříd metadat odkazových vazeb

Získávání metadat

Princip získávání metadat znázorňuje obrázek 4.5. Oranžové bloky představují definici entity (třídu), bloky označené DF jsou datová pole a EM je podřízený modul entity. Jak je patrné, metadata zakreslena modrými bloky přesně kopírují strukturu původní entity. K získání metadat

dojde během procesu pojmenovaného jako *registrace entity*. Ten provádí metoda databáze entit (třídy `EntitiesDatabase`). Teprve po registraci entity jsou pro ostatní části frameworku i konkrétní aplikace dostupná metadata dané entity. Bylo by možné registraci všech entit provádět povinně z určitého místa aplikace. Takový postup ale není příliš efektivní v případě, kdy aplikace obsahuje velké množství entit. Uvážíme-li, že u velkých informačních systémů (například typu ERP) může datový model obsahovat řádově tisíce až desetitisíce tabulek [1] a zároveň, že obecně bude každá z těchto tabulek reprezentována nějakou entitou, celkový počet entit v systému se bude pohybovat rovněž v tomto řádu[7]. Přitom ale pravděpodobně uživatel nebude ihned po jeho spuštění pracovat se všemi entitami systému. Režie způsobená registrací tisíců entit najednou by byla navíc velmi znatelná. Z těchto důvodů bude vhodné zvolit tzv. přístup *lazy loading*. Jedná se o načtení nebo inicializaci nějakého objektu teprve ve chvíli, kdy s ním nějaký jiný objekt potřebuje pracovat. Příklady takových postupů jsou popsány v [18]. V případě registrace entit to znamená, že entita bude registrována až ve chvíli, kdy s ní bude potřeba pracovat. Vhodným místem volání registrace tak může být konstruktor správce entit. Pokud by však aplikační kód potřeboval pracovat s metadaty entit dřív, než dojde k vytvoření první instance správce entit, musí si zajistit spuštění registrace typu entity samostatně. Alternativním řešením je umístění registrace entity do *statického konstruktoru třídy*, který je v jazyce C# volán jednou pro každou třídu a to ihned po použití názvu třídy kdekoliv v programu [3]. Může se jednat např. o deklaraci proměnné tohoto typu, práci se statickou vlastností nebo volání statické metody třídy. Statický konstruktor entity je rovněž vhodné místo pro umístění kódu registrace odkazových vazeb. S výhodou se zde využije toho, že je volán pouze jednou pro danou třídu.



Obrázek 4.5: Princip získávání metadat o entitách

Nyní je potřeba navrhnout vlastní postup procesu registrace entit. Celý postup bude založen na získávání informací o prvcích jazyka pomocí techniky nazývané reflexe (viz kapitola 4.1) a základním vstupem budou typové informace entity. Ty lze v prostředí programovacího jazyka C# získat buď vestavěným operátorem `typeof` s parametrem nastaveným na název

[7] Počet entit v systému bude zpravidla dokonce větší, než je počet tabulek v databázi, protože systém může data z tabulek agregovat do dalších pomocných entit, které mají pro uživatele vyšší vypovídající hodnotu.

třídy nebo voláním metody `GetType` na libovolné instanci třídy. V obou případech je návratovou hodnotou typ `System.Type`, s jehož pomocí lze získat všechny podstatné informace o třídě. Mezi nimi je i seznam všech vlastností třídy, který obsahuje datová pole a podřízené moduly. Třída ale může mít i další vlastnosti, které jsou z pohledu metadat nepodstatné a je nutné je odlišit. Platforma .NET k tomu poskytuje vhodný nástroj – atributy, jejichž základní vysvětlení je uvedeno v kapitole 4.1. Aby bylo možné odlišit datová pole od ostatních vlastností třídy, bude potřeba k nim umístit speciální atribut `DataFieldAttribute`. Atribut může mít zároveň další doplňující parametry, takže jeho součástí mohou být i dříve popsané parametry datového pole uchovávané třídou `DataFieldInfo`. Obdobné řešení platí i pro podřízené moduly, které budou uvozeny atributem `DetailModule`. Základní schéma pro nalezení datových polí entity pak může vypadat následovně:

```
foreach (PropertyInfo propInfo in
EntityType.GetProperties(BindingFlags.Instance | BindingFlags.Public))
{
    Object[] attribs =
      propInfo.GetCustomAttributes(typeof(DataFieldAttribute), true);

    if (attribs.Length > 0) {
        DataFieldAttribute dfAttr = (DataFieldAttribute)attribs[0];

        // zpracování datového pole

    }
}
```

Tučně zvýrazněná proměnná `EntityType` obsahuje typové informace entity získané operátorem `typeof` nebo metodou `GetType`. Následuje enumerace všech veřejných vlastností instance třídy (tedy nezajímají nás statické vlastnosti) a vyhledání atributu typu `DataFieldAttribute`. Pokud vlastnost má tento atribut definován, dojde ke zpracování vlastnosti jako datového pole.

Princip s definicí atributů lze použít i pro další doplňkové informace metadat, které pak využívají ostatní vrstvy frameworku nebo přímo konkrétní aplikace na něm založená. V následujících kapitolách budou postupně uváděny další možnosti využití atributů datových polí, podřízených modulů nebo celé entity. Zde ještě uvedu jeden ze stěžejních atributů vztahujících se k podřízeným modulům. Jedná se o atribut `AssociationAttribute` ze jmenného prostoru `System.ComponentModel.DataAnnotations`, který slouží k definici vazby (mapování datových polí) mezi nadřízenou a podřízenou entitou. Popis použití atributu je uveden v MSDN dokumentaci [19]. Na tomto příkladu je patrné, že samotná knihovna základních tříd platformy .NET obsahuje některé užitečné nástroje, které lze do frameworku zakomponovat a není nutné je vytvářet znova. I když by vylo možné doplnit definici vazby k atributu `DetailModuleAttribute`, využití standardních prostředků platformy se jeví jako výhodné i z hlediska kompatibility s dalšími produkty a technologiemi. Obzvlášť výše uvedený jmenný prostor obsahuje několik zajímavých atributů pro anotaci datových objektů. Během návrhu dalších části frameworku tak bude vhodné využít čím jak

nejvíce standardních prostředků pro anotaci entit. Kompletní seznam atributů jednotlivých prvků entit a návod k jejich použití je vzhledem ke svému popisnému charakteru umístěn v příloze A.

4.5 Validace entit

Tato kapitola se bude věnovat validaci vstupních dat. Vzhledem k tomu, že tento problém je nutné řešit prakticky v každé aplikaci a o to více v informačních systémech, které jsou převážně zaměřené na zpracování dat, měl by framework poskytovat robustní nástroje určené k podpoře validací a ověřování dat zadaných v entitách. V prvé řadě je nutné zdůraznit, že validace hodnot je úkolem nižších datových vrstev a vrstev s logikou aplikace, nikoliv vrstvy na úrovni uživatelského rozhraní. Již základní koncepce oddělení logiky aplikace od prezentační vrstvy uvedená v kapitole 4.1 počítá s možností použití celé části logiky a datových vrstev různými aplikacemi. Z nich některé ani nemusí mít uživatelské rozhraní (např. různé webové služby). Všechno toto jsou důvody pro umístění validací již na úroveň samotných entit případně datových polí. Datové pole by mělo zodpovídat za kontrolu platnosti své hodnoty a to, zda má být vyplněné nebo ne. Entita je pak zodpovědná za kontrolu v rámci celé entity (například více závislých hodnot mezi sebou).

Validace datových polí

Datové pole musí především zajistit kontrolu vyplnění hodnoty v případě, kdy je jeho zadání povinné. V části o datových polích kapitoly 4.2 bylo uvedeno, že stav povinnosti zadat hodnotu pole může být měněn v závislosti na okolnostech. Toto budou ale poměrně ojedinělé případy a častější bude požadavek na fixní nastavení povinnosti vyplnit hodnotu pole. Implicitní hodnotu této vlastnosti datového pole tak bude výhodné zadávat jako parametr atributu `DataField` (více informací o parametru je v příloze A).

Výše zmíněný jmenný prostor `System.ComponentModel.DataAnnotations` nabízí i další zajímavý atribut. Tím je `ValidationAttribute`, který jako abstraktní třída slouží pro implementaci jednoduchých validačních logik. Opět se jeví jako výhodné tento mechanizmus jednoduchého deklarativního způsobu zadávání validací využít a implementovat jej ve frameworku. Konkrétní atributy zděděné od třídy `ValidationAttribute` jsou během procesu registrace entity nalezeny a v metadatech zaznamenány jako validátory datového pole. Nová hodnota přiřazená do datového pole je postupně zaslána všem validátorům. Pokud některý z nich zjistí, že je hodnota chybná, měl by vrátit frameworku chybovou zprávu, která se pak po dalším zpracování dostane až k uživateli aplikace. Vytvoření vlastního validátoru spočívá pouze v přepsání metody `IsValid` abstraktní třídy `ValidationAttribute`. Podrobněji je problematika popsána v [20], kde je rovněž uveden seznam standardních validátorů obsažených ve jmenném prostoru `System.ComponentModel.DataAnnotations`. Jejich stručný popis je rovněž součástí přílohy A.

Validace na úrovni entit

Tato validace se uplatní především před ukládáním změn v entitě do úložiště dat a skládá se v podstatě ze dvou kroků. Prvním krokem je kontrola všech datových polí entity, zda se u nich nevyskytuje nějaká chyba validace. Pokud se nevyskytuje, přechází se ke kontrole vzájemné korespondence hodnot datových polí mezi sebou, případně i ve vztahu k hodnotám z jiných entit apod. Tato kontrola je tak jednou z klíčových prvků vynucujících pravidla podnikové logiky v informačním systému. Vzhledem k různorodosti možných kontrol na této úrovni, bude vhodné nechat uživatelům frameworku volnost v podobě virtuální metody entity, kterou budou moci přepsat a doplnit do ní vlastní kód kontrol. Tato metoda musí být interně volána frameworkem před fyzickým uložením entity a v případě neúspěchu validace nesmí k fyzickému uložení dojít. Deklarace této metody bude vypadat následovně:

```
protected virtual bool OnSaveValidation(
    bool newEntity,
    ValidationResults validationRes);
```

Metoda musí vrátit hodnotu `true` v případě, že validace uspěla, v opačném případě hodnotu `false`. První parametr slouží k rozlišení mezi validací nové entity nebo již existující změněné entity. Druhý parametr pak slouží k zaznamenávání chybových zpráv, které jsou pak předány vrstvě uživatelského rozhraní k zobrazení.

Doplňujícím způsobem může být opět využití atributu `ValidationAttribute`, který je navržen tak, aby mohl být uveden i u celé entity. V takovém případě jeho metoda `IsValid` dostává jako parametr instanci celé entity a může provést obdobné validace jako předchozí způsob. Tento způsob ale může být výhodnější v případě podobných typů validací opakujících se ve více různých entitách. Místo duplicitního kódu v metodách `OnSaveValidation` několika entit je lepší tento kód umístit na jedno místo do nového validačního atributu a ten uvést ke všem těmto entitám.

Předávání chyb validací vyšším vrstvám

Posledním problémem souvisejícím s validacemi je přenos informace o chybě při validaci vyšším vrstvám, především uživatelskému rozhraní. K tomu je potřeba na úrovni datových vrstev definovat třídu `ValidationResults`, která obsahuje kolekci zpráv (chybových hlášení) vzniklých během validace. Pokud během validace dojde k zápisu nějaké zprávy do objektu této třídy, musí být po ukončení validace spuštěna speciální událost. Na tuto událost se může připojit kód z vrstvy uživatelského rozhraní a z objektu `ValidationResults` přečíst chybové zprávy a následně je zobrazit například formou nějakého dialogu. V případě, kdy událost nebude nijak zpracována, framework by měl vyvolat aplikační výjimku, protože porušení validací a tedy i pravidel podnikové logiky aplikace je významná událost, kterou nelze ignorovat.

4.6 Uživatelské rozhraní

V kapitole 2.3 byly stanoveny základní koncepční požadavky na uživatelské rozhraní, kterých je nutné se během jeho návrhu probíraného v této kapitole držet. Jeden ze základních požadavků říká, že je nutné zohlednit možnost volby resp. záměny celé technologie použité pro uživatelské rozhraní při zachování nižších vrstev implementujících základní logiku aplikace. Proto také během návrhu všech předchozích součástí bylo striktně dodržováno pravidlo nezávislosti na vrstvě uživatelského rozhraní. Díky robustnímu návrhu systému metadat (kapitola 4.4) a mechanizmu validací entit (kapitola 4.5) lze implementovat uživatelské rozhraní jako samostatnou vrstvu postavenou nad datovým jádrem frameworku a navíc je umožněno těchto vrstev uživatelského rozhraní vytvořit více (např. pro různé technologie), což ve výsledku významnou mírou přispívá k rozšiřitelnosti a přizpůsobitelnosti celého frameworku.

Dále v této kapitole se omezím na návrh grafického uživatelského rozhraní. To ovšem neznamená, že by nebylo možné nad datovým jádrem frameworku vytvořit rozhraní konzolové aplikace.

4.6.1 Základní prvky

Data binding

Každé standardní grafické uživatelské rozhraní (GUI) se skládá z formulářů a na nich umístěných komponent. To platí obzvlášť pro aplikace zaměřené na zpracování dat. V podnikových informačních systémech potřebujeme mít možnost editovat vlastnosti entit, což znamená mít pro každý typ entity minimálně jeden formulář. Každý takový formulář obsahuje komponenty, které jsou propojeny s vlastnostmi entity a zobrazují se v nich, příp. umožňují editaci aktuálních hodnot připojených vlastností. Tento postup se nazývá *data binding*. V současnosti nabízí snad každá moderní technologie pro vývoj GUI nějaké prostředky pro nastavení takového propojení. Výjimkou není ani platforma .NET. Bohužel ale i v jednotlivých technologiích pro GUI platformy .NET je data binding řešen odlišným způsobem, což využití standardních prostředků poněkud komplikuje. Například technologie Windows Forms umožňuje téměř všechny vlastnosti komponent napojit na hodnoty z tzv. zdroje dat, který musí být součástí projektu, pomocí vlastnosti *(DataBindings)* ve vlastnostech komponenty. Zdrojem dat může být kromě databáze i vlastní kolekce nebo pole. Další informace o data bindingu ve Windows Forms lze nalézt v [21] a [16]. V technologiích založených na jazyku *XAML* (WPF, Silverlight) jsou možnosti podobné, jen je jejich použití díky přímému zápisu v kódu XAML jednodušší. Podrobný popis a příklady naleznete v [16] včetně postupů u technologie ASP.NET.

Pomineme-li skutečnost, že se postupy v různých technologiích GUI liší, v zásadě by jejich použití k napojení na datová pole entit nic nebránilo. Problém ale nastává s doplňujícími vlastnostmi datových polí, jak bylo uvedeno v kapitole 4.2. Komponenta napojená na datové pole by měla reflektovat nejen aktuální hodnotu pole, ale i další možné stavy (např. chybu validace, změněnou hodnotu apod.). To by ovšem znamenalo pomocí standardních prostředků data bindingu propojit i další vlastnosti komponenty. Takový postup je značně neefektivní, proto je nutné nalézt jiné řešení. Napojení komponenty na datové pole musí být provedeno čím

jak nejjednodušším způsobem, ideálně pouze nastavením názvu datového pole, ke kterému má být komponenta připojena. Všechny ostatní činnosti by už měla zajistit komponenta sama v součinnosti s datovým polem a metadaty entit. To je možné zajistit následujícím rozšířením datového jádra frameworku.

- Datové pole musí poskytovat události, které informují o změně hodnoty a stavu (příznaků „jen pro čtení“, „povinné“ a „platná hodnota“).

- Obecný předek entit musí umožnit přístup k vnitřním instancím datových polí (tříd odvozených od `IDataField`, které běžně nejsou veřejně přístupné mimo entitu) na základě názvu datového pole.

Komponenta pak na základě nastaveného názvu datového pole získá její instanci, zaregistruje si odpovídající události a jejich zpracováním může ovlivňovat svou grafickou podobu. Jedinou nevýhodou tohoto postupu je nutnost vytvoření speciálních komponent, které pak musí programátor uživatelského rozhraní použít, aby byla zajištěna uvedená funkcionalita.

Volba technologie pro GUI

Pro účely této práce bude nutné zvolit některou z dostupných technologií platformy .NET pro vytváření GUI a na ní prezentovat možnosti frameworku. I když dnes stále více roste popularita vícevrstvých aplikací s tzv. tenkými klienty, pro rozsáhlé podnikové systémy jsou stále poměrně často využívány desktopové aplikace. Proto i primární GUI podporované frameworkem bude zaměřené na desktopové aplikace, pro něž jsou dostupné dvě technologie – Windows Forms a Windows Presentation Foundation (WPF). Popis rozdílů mezi oběma technologiemi a jejich výhody a nevýhody jsou nad rámec této práce. Informace však je možné nalézt v množství publikací, např. [21] se výhradně věnuje podrobnému představení Windows Forms, WPF pak [16], [17] a částečně i [22]. Vzhledem k zaměření frameworku na podnikové aplikace je možné využít obě technologie, avšak potenciál moderní WPF by zůstal prakticky nevyužit. Dalšími důvody pro volbu[8] Windows Forms jsou jednoduchost použití (programátoři se nemusí učit jazyk XAML (eXtensible Application Markup Language, [17]) a celkově jiný koncept GUI), ale také zpětná kompatibilita.

Základní potřebné komponenty

Podpora GUI, která je součástí frameworku bude obsahovat sadu základních komponent s podporou navrženého data bindigu. Komponenty by měly především pokrýt typy standardně dostupných datových polí a dále tabulkové zobrazení. Jejich seznam je uveden v tabulce 4.2. Jedná se vždy o komponenty odvozené od odpovídajících standardních komponent, které mají pro snadné rozlišení přidanou koncovku „ISF“.

[8] Problematice volby technologie GUI se věnuje množství publikací (např. i [20]). Každá technologie má mnoho zastánců i odpůrců a volba se obvykle odvíjí od předpokládaného zaměření aplikace. I když pro účely této práce bylo zvoleno Windows Forms, většina uvedených postupů je aplikovatelná i pro WPF.

Komponenta	Typ datového pole	Poznámka
`TextBoxISF`	Všechny pro zobrazení. Pro editaci String, Long, Byte, Double	
`CheckBoxISF`	Bool	Zaškrtávací pole.
`DateTimePickerISF`	Date, DateTime, Time	Výběr data a/nebo času.
`LookupISF`	Long, Byte, String, Binary	Komponenta pro podporu odkazových vazeb.
`DFCaptionLabel`	Všechny typy	Nezobrazuje hodnotu datového pole, ale jeho název.
`DataGridISF`		Tabulkové zobrazení všech entit ve správci entit. Komponenta je odvozena od třídy `DataGridView`. Podrobnosti o její implementaci v kapitole 5.5.

Tabulka 4.2: Přehled komponent s podporou data bindingu na entity frameworku

Komponenty s nastaveným názvem datového pole potřebují ještě nějakým způsobem získat instanci konkrétní entity, na jejíž datová pole se mají napojit. Proto bude vhodné komponenty sdružit v nějakém kontejneru, který bude znát instanci zobrazované entity. Zároveň může kontejner řídit stav komponent. Předpokládá se totiž, že komponenty mohou být použity také jen pro zobrazení detailů entity bez možnosti editace. Vytvářet pro zobrazení zvláštní formulář a pro editaci hodnot druhý formulář ale není vhodné. Mnohem výhodnější je mít jeden formulář pro danou entitu, který slouží jak k zobrazení detailních informací, tak i k jejich editaci. Tyto dva stavy je ale nutné odlišit například zakázáním komponent. Pokud bude režim komponent spravovat kontejner, bude navíc možné mezi stavy jednoduše přecházet (například zobrazit detail entity jen pro čtení, přejít do režimu editace, provést změny a po uložení přejít zpět do režimu jen pro čtení). Takovým kontejnerem pro komponenty by mohl být přímo formulář, nicméně univerzálnější bude pro tyto účely vytvořit speciální komponentu odvozenou od třídy `Panel`.

Aby mohly komponenty s panelem komunikovat, vzniknou dvě nová rozhraní. Prvním je `IISFBindableControl`, které musí být implementováno komponentami a má tyto metody a vlastnosti:

```
public interface IISFBindableControl
{
    String DataFieldName { set; get; }

    bool Initialize(EntityMetadata emd);

    void BindEntity(BaseEntity currentEntity);
```

```
    void RefreshValue();

    void StoreValue();

    void SetEditMode(bool edit);

}
```

Vlastnost `DataFieldName` obsahuje název datového pole entity. První dvě metody slouží k inicializaci datového pole a budou popsány dále. Název metod `RefreshValue` a `StoreValue` je dostatečně vypovídající a poslední metoda je volána právě při přechodu mezi režimy prohlížení a editace.

Druhým rozhraním je `IISFControlContainer`:

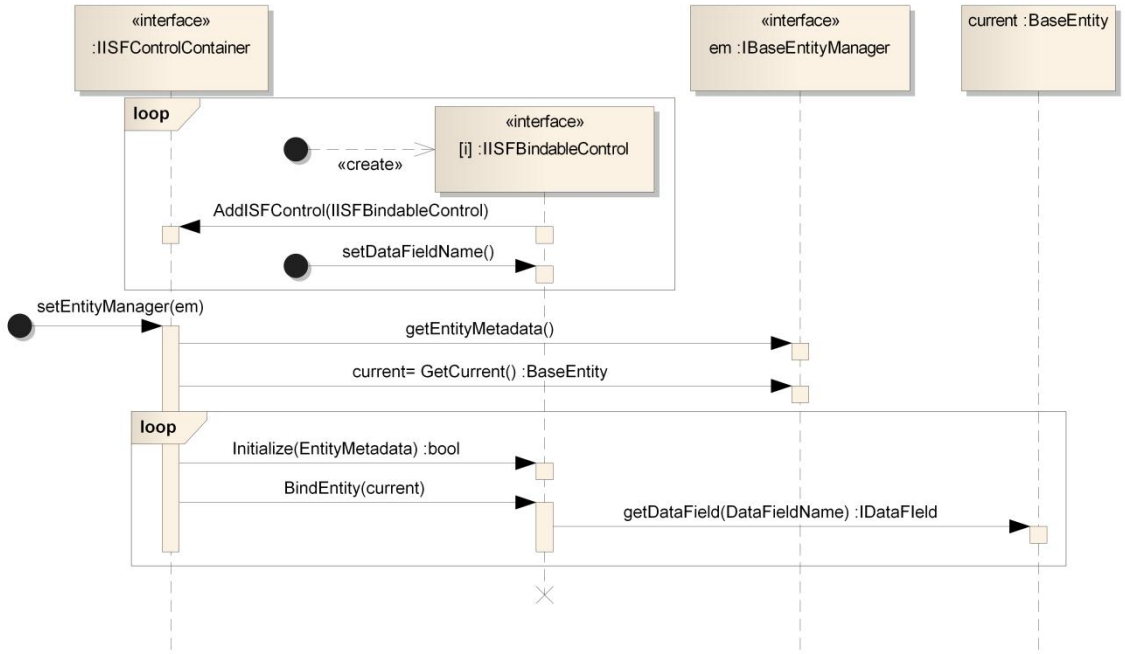
```
public interface IISFControlContainer
{

    IBaseEntityManager EntityManager { set; get; }

    void AddISFControl(IISFBindableControl control);

    bool StartEdit(bool refresh);

    bool EndEdit(bool save);

}
```

Význam metod těchto rozhraní přibližuje diagram sekvence na obrázku 4.6. Je na něm zachyceno nejprve vytváření komponent a jejich registrace v kontejneru voláním metody `IISFControlContainer.AddISFControl`. Každá komponenta frameworku po svém vytváření hledá v hierarchii svých rodičů komponentu, která implementuje rozhraní `IISFControlContainer` a voláním uvedené metody se u něj zaregistruje.

Druhá část diagramu zachycuje inicializaci komponent zahájenou přiřazením instance správce úloh do vlastnosti `EntityManager` kontejneru. Ten nejprve zjistí metadata a předá je všem svým registrovaným komponentám (metoda `Initialize`). Toto je místo, kde si můžou komponenty zjistit z metadat některá další nastavení datového pole. Jako poslední krok je zjištění aktuální entity ve správci úloh a předání této entity všem komponentám pomocí volání metody `BindEntity`. Každá komponenta si pak z entity získá instanci daného datového pole, z něho přečte hodnoty a zaregistruje pomocné obsluhy události (poslední dvě akce již diagram nezachycuje).

Poněkud odlišná je komponenta `LookupISF`. Jedná se o komponentu, která vytváří uživatelské rozhraní k odkazovým vazbám (kapitola 4.3). Nejvíce se podobá tzv. comboboxu, což je komponenta umožňující výběr jedné položky z nabízeného seznamu. Takový výběr přesně odpovídá sémantice odkazových vazeb. Komponenta `ComboBox` se ale k tomuto účelu příliš nehodí. Při výběru ze seznamu dostupných entit by mělo být možné minimálně zobrazit více sloupců z cílové entity, případně před výběrem zobrazit detailní informace o konkrétní entitě. Z těchto důvodu bude nutné komponentu implementovat od začátku. Dalším rozdílem je také způsob napojení na datové pole. Lookup nebude požadovat zadání názvu datového pole,

ale místo něj se bude zadávat unikátní identifikátor zaregistrované vazby (kapitola 4.3). Na základě mapování polí z definice vazby pak komponenta může po výběru správně naplnit odpovídající datová pole.



Obrázek 4.6: Diagram sekvence zachycující vytváření komponent a inicializaci kontejneru

Komponenta `DataGridISF` se jako jediná odlišuje od tohoto konceptu. Kromě jednoduchých komponent vázajících se na jednotlivá datová pole entit musí framework ještě poskytnout komponentu pro tabulkové zobrazení více entit najednou, přesněji všech entit ze správce entit. Poměrně běžný postup implementace GUI v podnikových aplikacích spočívá v tom, že je nejprve uživateli nabídnut seznam dostupných entit v podobě tabulky s několika klíčovými sloupci a uživatel si pak nějakou vybere a zobrazí všechny její detaily v určitém formuláři. K těmto účelům slouží ve Windows Forms standardní komponent `DataGridView`. Její podrobný popis včetně návodů na její úpravy je uveden v [21]. Upravená verze pro framework musí především opět podporovat snadný data binding. Na rozdíl od jednoduchých komponent bude její inicializace probíhat přiřazením celé instance správce entit. Inicializace spočívá především ve vytvoření patřičných sloupců a namapování datových polí jednotlivých entit na sloupce. Definice sloupců musí být ovlivnitelná zvenčí pro konkrétní instanci entity, proto bude nutné přidat komponentě vhodnou událost. Komponenta `DataGridISF` ale může být opakovaně použita na různých místech pro zobrazení stejného typu entit. Takto by bylo nutné opakovaně provádět totožnou definici sloupců. Nabízí se možnost opět využít metadat entit a atributů datových polí. K datovým polím lze přiřadit nový typ atributu (`ImplicitColumnAttribute`), jenž bude označovat datová pole, která se mají v tabulkách implicitně zobrazit. Další implementační detaily této komponenty zmiňuje kapitola 5.5.

Základní typy formulářů

S různými podnikovými informačními systémy zpravidla pracuje široká paleta uživatelů různých oborů i úrovně technických znalostí. Při návrhu uživatelského rozhraní takové aplikace se musíme snažit, aby její ovládání bylo co nejvíce intuitivní a přímočaré. Zejména je vhodné, aby se napříč celým systémem vyskytovaly jednotné prvky a způsoby jejich ovládání. To s sebou nese mj. sjednocení vzhledu a ovládání formulářů. I v tomto směru může navrhovaný framework usnadnit práci. Zvolená technologie pro GUI (Windows Forms) k tomu navíc poskytuje vhodný prostředek. Tzv. *dědičnost formulářů* funguje podobně jako dědění tříd (formuláře také nejsou nic jiného než třída a k ní soubor s prostředky). Jak uvádí příklad v [22], je možné určité společné prvky i chování implementovat v základním formuláři a dále specializovat a rozšiřovat je na odvozených formulářích. Výhodou je i podpora na úrovni návrháře formulářů ve vývojovém prostředí. Součástí frameworku ve vrstvě GUI tak bude několik předků aplikačních formulářů uvedených v následující tabulce 4.3.

Formulář	Vlastnosti
`ISFContainerForm`	Základní předek, který obsahuje pouze kontejner pro komponenty a implementuje základní ošetření chybových zpráv při validaci, kontrole oprávnění apod. Měl by být využit jako základ pro implementaci vlastních předků, v případě, že následující dva z nějakého důvodu nevyhovují.
`BaseDetailForm`	Předek formuláře určeného k zobrazování detailů a editaci klíčových entit v aplikaci. Umožňuje přechod mezi stavem prohlížení a editací a obsahuje k tomu již připravené ovládací prvky (tlačítka).
`BaseDialogForm`	Předek jednoduchých dialogů. Na rozdíl od předchozího obsahuje pouze dvě tlačítka „OK“ a „Storno“ a neumožňuje přechod mezi stavy prohlížení a editace. Režim, v jakém se formulář zobrazí, je dán předem. Měl by být použit především pro editaci jednodušších entit podřízených modulů s menším množstvím údajů.

Tabulka 4.3: Vlastnosti základních předků formulářů

4.6.2 Generované formuláře

Jednou z časově nejnáročnějších prací při vývoji IS je právě tvorba uživatelského rozhraní. V předchozí kapitole bylo popsáno zjednodušení v podobě vlastních komponent GUI, které zjednodušují tzv. data binding, a předdefinovaných formulářů implementujících základní funkcionalitu. Často však nastává situace, kdy je potřeba vytvořit pouze jednoduchý formulář s několika málo komponentami přímo napojenými na entitu. Tuto činnost může framework zcela eliminovat pomocí mechanismu *generovaných formulářů*. Jedná se o formuláře, jejichž obsah je dynamicky vytvořen až za běhu aplikace na základě popisu entity získaného z metadat.

Takový formulář tedy není nutné nikde předem vytvářet a programátorovi odpadá práce s jeho návrhem.

Základním principem dynamického vytváření obsahu formuláře je z metadat konkrétní entity získat informace o datových polích a vytvořit pro ně ve formuláři odpovídající komponenty včetně napojení a doplnění popisu pole. Výsledkem tedy bude uspořádání komponent do dvou sloupců, přičemž v prvním bude vždy textový popis s názvem datového pole a v druhém sloupci bude komponenta odpovídající datovému typu pole.

Podpora na úrovni metadat entit

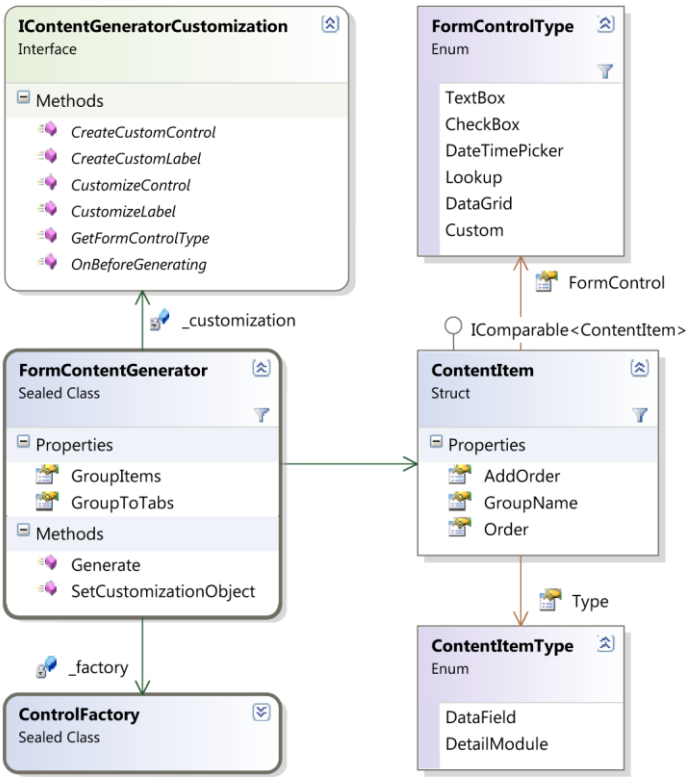
Metadata obsahují informace o všech datových polích a podřízených modulech entity. Většinou ale bude potřeba zobrazovat jen některá pole entit (např. automaticky generovaný číselný identifikátor entity je pro uživatele většinou nepodstatný údaj). Proto bude nutné na úrovni metadat omezit, která datová pole a podřízené moduly budou na formuláři zobrazeny. K tomuto účelu lze použít atribut `DisplayAttribute` z již dříve zmíněného jmenného prostoru `System.ComponentModel.DataAnnotations` (kapitola 4.4). Jedná se o atribut ze základní knihovny tříd, který je již zároveň zpracováván některými technologiemi platformy .NET. Jeho primárním účelem je označení vlastností objektu (entity), které mají, resp. nemají být zobrazeny v uživatelském rozhraní. Dalšími vlastnostmi atributu lze ovlivnit zobrazované jméno datového pole (což lze využít jako zdroj i pro standardní položku metadat), název skupiny, v níž se má pole zobrazovat, anebo pořadí pole. Podrobný popis atributu je uveden v [23] a také v příloze A.

Druhým problémem bude vytváření komponenty `LookupISF`. Odkazové vazby jsou registrovány k určitým datovým polím entity. Toto datové pole může mít definován atribut `DisplayAttribute`, ale vytvořila by se pouze standardní komponenta `TextBoxISF`, která jen zobrazí zpravidla číselnou hodnotu. Framework navíc umožňuje k jednomu datovému poli zaregistrovat více různých odkazových vazeb, které se pak používají podle kontextu. To má za následek, že prakticky jakýkoliv způsob automatické detekce, zda použít `LookupISF` nebo `TextBoxISF`, není příliš vhodný, protože nebude spolehlivě pracovat ve všech případech. Situaci by šlo sice obejít přizpůsobením generování obsahu, jak je popsáno dále v této kapitole, ale zase to pro standardní případy komplikuje použití. Proto bude vhodnější k atributu `DataFieldAttribute` (kapitola 4.4) přidat další volitelný parametr (`LookupLinkId`), do něhož lze zadat unikátní identifikátor vazby. Jakmile je během generování obsahu tento identifikátor nalezen, bude k datovému poli automaticky vytvořena komponenta `LookupISF`.

Návrh řešení

Automatické generování obsahu formuláře za běhu je jednou z klíčových částí frameworku v oblasti usnadnění práce a zrychlení vývoje IS. Proto by měl být návrh této funkcionality co nejrobustnější. Vzhledem k tomu, že zbylé části frameworku i v oblasti uživatelského rozhraní poskytují programátorům značnou volnost a možnost přizpůsobení, bude se tohoto konceptu držet i systém generování formulářů. Zejména se jedná o možnost aplikovat generování obsahu na libovolný uživatelský formulář, který není součástí frameworku.

Na obrázku 4.7 je ukázán diagram tříd, které se podílejí na generování obsahu. Jak je z něj patrné, nenachází se v něm žádný konkrétní formulář. Veškerá logika spojená s generováním bude naopak umístěna ve třídě s názvem `FormContentGenerator`, která vytváří dynamický obsah do libovolného určeného kontejneru. Kontejnerem se zde myslí komponenta formuláře, která může obsahovat další komponenty. Pro tyto účely lze za kontejner považovat libovolnou třídu odvozenou od třídy `System.Windows.Forms.ScrollableControl` [21], jejíž vlastnost automatického zobrazení posuvníků v případě zobrazení obsahu, který přesahuje rozměr kontejneru, s výhodou využijeme. Díky tomuto postupu je například možné použít generování obsahu jen na část formuláře nebo ve formuláři zobrazit pole různých entit apod. Pro tabulkové zobrazení komponent ve dvou sloupcích (název pole, samotná komponenta) lze s výhodou použít další třídu typu kontejner – `System.Windows.Forms.TableLayoutPanel`, která bude vložena do předaného kontejneru. Podrobnosti o `TableLayoutPanel` naleznete v [21], její výhodou je především automatické přizpůsobování rozměrů sloupců příp. i řádků při změně velikosti nadřízeného objektu (např. celého formuláře).



Obrázek 4.7: Diagram tříd podílejících se na dynamickém generování obsahu formuláře

Další vlastností, kterou by měl generátor obsahu poskytovat, je vizuální seskupování komponent. V případě, kdy má entita větší počet datových polí, je vhodné logicky související seskupit k sobě (např. základní informace o osobě, kontaktní informace, apod.). K vizuálnímu seskupení lze využít komponent `GroupBox`, která pouze kolem obsažených komponent vykreslí orámování se zadaným nadpisem. Nástroj pro generování obsahu ale musí zajistit její vytvoření. K povolení tohoto chování slouží vlastnost `GroupItems` třídy `FormContentGenerator`. Druhou možností jak komponenty seskupit je pomocí tzv. záložek neboli komponenty `TabControl`. Tento druhý postup je vhodnější v případě, kdy má entita velký počet datových polí, které by se na formulář nevešly. Toto chování ovlivňuje vlastnost `GroupToTabs`.

Obrázek 4.7 dále zobrazuje rozhraní `IContentGeneratorCustomization`, které slouží k přizpůsobení vytvořených komponent. Během postupu vytváření obsahu (viz dále) jsou metody tohoto rozhraní volány a jejich implementací je možné výsledný obsah vhodně přizpůsobit. Rovněž jej lze využít v případě, kdy je pro některé datové pole nutné použít jinou

(např. i vlastní) komponentu. Nastavit vlastní implementaci objektu pro přizpůsobení lze voláním metody `SetCustomizationObject`.

Podpůrnou funkci plní objekt třídy `ControlFactory`, který obsahuje tovární metody pro vytváření prototypů komponent jednotlivých typů. Jsou vyčleněny do samostatné třídy z důvodu možnosti jejich snadného využití v objektech pro přizpůsobení.

Postup generování obsahu

1) Samotné spuštění generování obsahu zajišťuje metoda `Generate` třídy `FormContentGenerator`. Vstupem generování je cílový kontejner, do něhož se bude vytvářet obsah, instance entity, pro niž se formulář vytváří, a případně další parametry (seskupování, objekt pro přizpůsobení).

2) V metadatech vstupní entity jsou vyhledána datová pole a podřízené moduly s odpovídajícím označením atributem `DisplayAttribute`. Zároveň je interně vytvořena kolekce záznamů typu `ContentItem` obsahující informace o zobrazovaných prvcích formulářů. Tato kolekce je následně seřazena podle pořadí z hodnoty atributu nebo umístění definice datového pole v entitě. Pokud je povoleno seskupení, musí se navíc prvky interně seskupit.

3) Pro všechny prvky kolekce je nutné stanovit odpovídající typ komponenty (hodnotu výčtového typu `FormControlType`). Základní typy datových polí, které jsou součástí frameworku, se mapují na komponenty podle tabulky 4.2. Pro možnost ovlivnění typu komponenty je ještě zavolána metoda `GetFormControlType` objektu pro přizpůsobení.

4) V kontejneru jsou vytvořeny základní komponenty (např. `TableLayoutPanel` nebo `TabControl`).

5) Pro každý prvek je vytvořena odpovídající komponenta a také komponenta s popiskem. Následně jsou tyto dvě komponenty předány metodě `CustomizeControl` resp. `CustomizeLabel` objektu pro přizpůsobení, kde je možné upravit jejich vlastnosti. Nakonec jsou vloženy do `TableLayoutPanel`. V případě, kdy volání metody `GetFormControlType` vrátilo hodnotu `FormControlType.Custom`, generátor nevytváří automaticky žádné komponenty, ale volá metody `CreateCustomControl` a `CreateCustomLabel`, které jsou plně zodpovědné za jejich vytvoření.

Uvedený postup zajistí poměrně široké možnosti přizpůsobení postupu generování obsahu, především jednotlivých komponent pro datová pole a podřízené moduly. V mnohých případech bude pravděpodobně méně náročné vytvořit jednoduchou implementaci rozhraní `IContentGeneratorCustomization` a využít konceptu dynamického vytváření obsahu formulářů místo jejich ručního návrhu v grafickém designeru, jež je součástí vývojového prostředí.

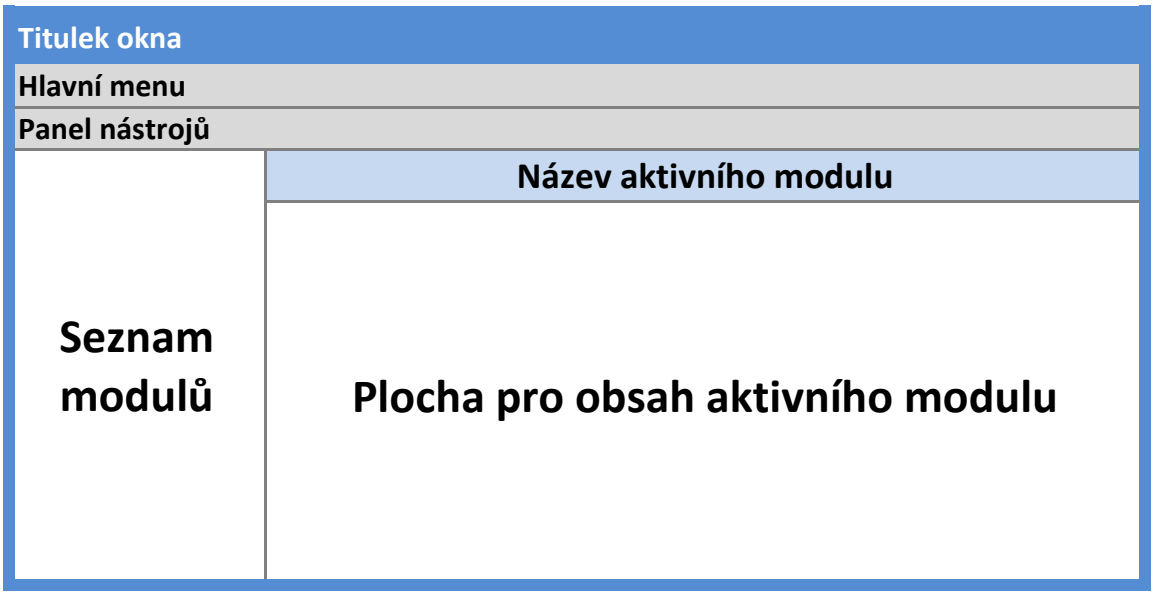
Aby byly důkladně pokryty i nejstandardnější případy, kdy bude využito základní generování obsahu a zároveň standardních šablon formulářů (tabulka 4.3), budou vytvořeny dva nové obecné formuláře – `GeneratedDetailForm` a `GeneratedDialogForm`. Oba vycházejí z analogicky pojmenovaných šablon formulářů (`BaseDetailForm`, resp. `BaseDialogForm`) a navíc interně využívají objektu `FormContentGenerator` pro automatické vytvoření obsahu.

4.6.3 Moduly a hlavní formulář aplikace

Poslední zbývající částí uživatelského rozhraní je *hlavní formulář aplikace*. I když požadavky na něj mohou být pro různé aplikace poměrně rozdílné, framework by měl ve svém základu poskytovat jeho jednoduchou verzi. Zároveň je ale nutné, aby měli programátoři možnost tento formulář zcela nahradit svým vlastním. Hlavní formulář slouží jako vstup do všech částí výsledného IS. Uživateli se zobrazí po spuštění a jeho zavřením dojde k ukončení aplikace. Zobrazovat by měl především dostupné moduly informačního systému. *Moduly* jsou pomocné logické struktury na úrovni uživatelského rozhraní, které zastřešují související části IS. Jedná se o typ entity, správce těchto entit, zobrazovací a editační formuláře pro dané entity a případně doplňující aplikační logiku. Každý takový modul pak bude spouštěn z hlavního formuláře aplikace.

Hlavní formulář

Způsobů, jak celkově koncipovat hlavní formulář, lze vymyslet velké množství a pravděpodobně budou odlišné v různých aplikacích. Programátor nicméně bude mít možnost vytvoření vlastního aplikačního formuláře. Pro účely standardního hlavního formuláře dodávaného jako součást frameworku bude zvolen koncept rozdělení jeho plochy na několik částí, které znázorňuje obrázek 4.8.



Obrázek 4.8: Rozvržení ovládacích prvků hlavního formuláře aplikace

Největší část formuláře zabírá *plocha pro obsah aktivního modulu*. Do této plochy může modul vykreslit prakticky libovolný obsah. Nejčastěji se bude jednat o nějaké tabulkové zobrazení nejdůležitějších entit pro daný modul v podobě komponenty `DataGridView`. Doplňkem mohou být další ovládací prvky např. pro možnost filtrovaní a řazení zobrazených dat. Další důležitou částí formuláře je *seznam modulů*, v němž se bude zobrazovat seznam dostupných modulů aplikace. Výběrem některého z nich se daný modul aktivuje a převezme zodpovědnost za vykreslování plochy pro obsah. *Panel nástrojů* bude obsahovat některé základní společné operace pro všechny moduly (např. zobrazit detail záznamu, editace záznamu, vložení nového) a dále bude nutné umožnit doplňování dalších tlačítek pro specifické akce nad aktivním modulem.

Hlavní formulář bude navíc implementovat rozhraní `IAppForm`. Toto rozhraní bude využívat část řízení stavu a správy modulů aplikace (viz dále) pro notifikaci některých událostí formuláři. Jde tak o součást podpory pro zcela vlastní implementace hlavních formulářů aplikace. Definice rozhraní vypadá takto:

```
public interface IAppForm
{

    void ModuleSwitched(BaseModule newModule);

    void NotifyStdActionsAvailability();

    void AddExtraAction(string actionName, EventHandler onActivation,
                        string displayName, Image icon);

    void RemoveExtraAction(string actionName);

}
```

První dvě metody slouží k oznámení pro formulář, že došlo ke změně aktivního modulu, resp. že došlo ke změně dostupnosti základních operací nad modulem. Během přepnutí aktivního modulu je zapotřebí zaměnit obsah[9] vykreslovaný v ploše okna, změnit zobrazený název aktivního modulu a případně upravit dostupnost akcí v panelu nástrojů. Druhé dvě metody slouží k přidávání a odebírání dalších vlastních akcí, které se ve výchozím formuláři zobrazují jako další tlačítka v panelu nástrojů. Jiná implementace hlavního formuláře může samozřejmě zobrazení nabízených akcí řešit jinak.

Moduly

Framework musí na úrovni vrstvy uživatelského rozhraní spravovat moduly aplikace. Pro tyto účely bude vytvořen pomocný objekt `ModulesManager`, u něhož se budou jednotlivé moduly aplikace registrovat, bude spravovat jejich kolekci, řídit inicializaci a přepínání aktivního modulu. Každý modul systému bude reprezentován objektem odvozeným od třídy `BaseModule`. Třída modulu musí poskytovat tyto informace:

[9] Předpokládaným zobrazovaným obsahem bude komponenta DataGridView nebo jiná vlastní složená komponenta (UserControl).

- Interní identifikátor modulu sloužící k aktivaci modulu pomocí metody třídy `ModulesManager`
- Zobrazovaný název modulu.
- Primární správce entit daného modulu (např. pro modul správy uživatelů to bude entita „uživatel“, modul ale může při dalších činnostech pracovat i s jinými typy entit).
- Komponentu, která má být vykreslována v ploše pro obsah okna.
- Stav povolení základních společných akcí.
- Zda je možné daný modul spustit (např. pokud uživatel nemá dostatečná oprávnění).
- Musí poskytovat metody implementující činnosti provedené při aktivaci společné akce.

Uvážíme-li, že nejběžnější modul bude dle tohoto schématu mít jako komponentu v ploše pro obsah `DataGridView` zobrazující entity z primárního správce entit, povoleny standardní operace a pro zobrazování a editaci se využije generovaných formulářů (kapitola 4.6.2), lze prakticky všechny činnosti, které má modul na starosti, zautomatizovat. Jedinými potřebnými parametry jsou typ primární entity (typ primárního správce entit) a případně identifikátor modulu a zobrazovaný název. Takto upravená implementace modulu bude umístěna do generické třídy `DataGridModule<TEntityManager>`, kterou bude možné použít buď přímo, nebo od ní odvodit vlastní upravené třídy modulů.

Jeden ze zaregistrovaných modulů bude sloužit jako „domovský“ modul a bude automaticky aktivován po spuštění aplikace. Tento modul může například v ploše zobrazovat nějaké základní údaje, přehled agendy přihlášeného uživatele apod.

4.7 Silverlight klient

Dosavadní návrh byl zaměřen na tzv. *dvouvrstvou architekturu* (také označovanou jakou *two-tier*). V tomto případě klientská aplikace využívající framework přímo komunikuje s databázovým serverem (obecně nějakým úložištěm), což je dostačující v případě vnitrofiremního použití. Na stanicích je nainstalován IS, který se připojuje po interní síti ze všech stanic k jednomu databázovému serveru. Problém tohoto řešení nastává v případě požadavku na práci s daty v systému i mimo interní firemní síť. V současnosti je potřeba takového přístupu nazývaného *vícevrstvá architektura* (také *multi-tier*) pomocí tzv. *tenkých klientských aplikací* nebo webového rozhraní stále častější. Podpora vícevrstvé architektury tak může značně rozšířit oblasti využití celého frameworku.

Základní charakteristikou vícevrstvé architektury je vytvoření prostřední vrstvy nacházející se mezi klientem a databázovým serverem. Ta řeší většinu aplikační logiky a zpravidla je implementována jako serverová aplikace (služba). Klientská aplikace pak obsahuje pouze prezentační vrstvu, komunikuje určitým síťovým protokolem s prostřední vrstvou a obsahuje minimum nebo žádnou aplikační logiku.

Vhodnou technologií pro implementaci tenkého klienta je např. *Silverlight*. Podle [9] se jedná o multiplatformní prezentační technologii určenou primárně pro běh v rámci internetového prohlížeče (resp. pluginu v něm nainstalovaném). Rovněž využívá běhového prostředí CLR, které ale poskytuje pouze podmnožinu funkcionality oproti plné platformě .NET. To vyplývá především z bezpečnostních důvodů, protože Silverlight aplikace může být umístěna na libovolné internetové stránce. Koncept tvorby uživatelského rozhraní v Silverlight je založen na jazyku XAML [17], resp. jeho podmnožině oproti WPF. Více podrobností o této technologii naleznete v [9]. Verze 4 a současná verze 5 je navíc rozšířena o poměrně dobrou podporu pro podnikové (business) aplikace. Nejvýznamnější podporu zajišťuje na úrovni komunikace mezi klientskou aplikací a prostřední vrstvou, kde je potřeba přenášet po síti celé entity a spouštět různé operace implementující aplikační logiku. Komunikace mezi klientem a prostřední vrstvou zpravidla probíhá po internetu, proto musí být zabezpečená. Všechny tyto požadavky zajišťuje vrstva *WCF RIA Services*, která staví na využití technologie *WCF*[10] (*Windows Communication Foundation*). WCF RIA Services lze definovat jako framework, který zjednodušuje zpřístupnění dat ze serveru v klientské Silverlight aplikaci a zároveň umožňuje sdílet validační logiku [20]. Další klíčovou částí je podpora ve vývojovém prostředí Visual Studio, které umožňuje definovat vazbu mezi projektem obsahujícím webovou službu a projektem se Silverlight aplikací. Této vazby následně využívá *generátor kódu* klientské aplikace. Generování kódu zajistí definici odpovídajících entit na straně Silverlight aplikace automaticky podle jejich ekvivalentů na straně webové služby a dále vytvoření potřebných objektů zajišťujících komunikaci mezi klientem a serverem. Podrobné vysvětlení problematiky generování kódu je uvedeno v [20].

Vhodnou kombinací a vzájemnou spoluprací WCF RIA Services a doposud navrženého frameworku bude možné zajistit podporu vícevrstvé architektury aplikace s těmito klíčovými vlastnostmi:

- Entity pro konkrétní aplikaci budou definovány na jednom místě – v projektu obsahujícím webovou službu, případně ve zvláštní assembly pro možnost využití entit a logiky ve více projektech.

- Přenos entit na stranu klientské aplikace zajistí modifikované generování kódu z WCF RIA Services.

- Podpora na straně Silverlight bude zajištěna vytvořením speciální knihovny obsahující podmnožinu tříd plného datového jádra frameworku. Především půjde o nástroje nutné k extrakci metadat z definice entit a pro přístup k matedatům (kapitola 4.4).

- Dostupnost metadat i na straně Silverlight aplikace zpřístupní široké možnosti pro definici uživatelského rozhraní, generovaných formulářů (kapitola 4.6.2) apod. jako je tomu u Windows Forms uživatelského rozhraní.

[10] WCF je jednotná komunikační vrstva zastřešující některé další technologie za účelem implementace síťové komunikace mezi různými zařízeními a platformami. Více informací o WCF naleznete v [17].

- Webová služba bude definovat operace potřebné pro zpřístupnění entit dle konceptů uvedených v [20] a k jejich implementaci bude využito plného rozhraní datového jádra frameworku popsaného v předešlých kapitolách.

- Další vlastnosti vyplývají z použití nástroje WCF RIA Services [20], na němž je veškerá komunikace mezi vrstvami postavená.

Nutné úpravy a rozšíření oproti předchozímu návrhu

Nejvýznamnějším aspektem zavedení podpory pro vícevrstvou architekturu je nutnost optimalizace především základních operací datového jádra frameworku pro vícevláknový přístup. Po zpřístupnění operací implementující logiku webovou službou mohou být jednotlivé operace volány souběžně různými klienty, což vede ke spuštění operací ve více vláknech paralelně. Především je nutné řešit souběžný přístup k registraci metadat entit, jejich získávání a paralelním přístupu k databázovému serveru. Metadata entit mohou být sdílena mezi jednotlivými vlákny, je však nutné zajistit synchronizaci. K těmto účelům nabízí verze 4 platformy .NET jmenný prostor `System.Collections.Concurrent` [24] obsahující generické kolekce implementující synchronizovaný přístup. Užitečná je např. kolekce `ConcurrentDictionary<TKey, TValue>`.

Další nutnou úpravou je extrakce společného kódu sdíleného mezi podpůrnou knihovnou pro Silverlight a plnou variantou frameworku. Bude se jednat především o kompletní podporu získávání metadat z entit, doplňující atributy a některé základní rysy obecného předka entit (`BaseEntity`). Následující úprava spočívá v doplnění potřebných atributů [20] zajišťujících korektní přenos entit pomocí WCF RIA Services. Přenášeny by měly být primárně pouze hodnoty datových polí entity, další podpůrné vlastnosti uchovávající např. stav entity není nutné přenášet.

Poslední nutná změna se týká již konkrétních entit pro danou aplikaci. Jejich programátor musí dodržovat některá pravidla zmíněná např. v [20] plynoucí z použití WCF RIA Services. Jedná se například o nutnost určení primárního klíče každé entity pomocí atributu `KeyAttribute` nebo označení podřízených modulů atributem `IncludeAttribute` a `AssociationAttribute`. Naopak při využití standardního navrženého způsobu validací hodnot entit v kapitole 4.4 pomocí validačních atributů zajistí generátor kódu automaticky i přenos těchto atributů a podpůrné vrstvy WCF RIA Services je dokáží na klientské straně vyhodnocovat před odesláním hodnot na server.

WCF RIA Services při generování klientského kódu zároveň hledá v projektu webové služby soubory s příponou *.shared.cs* a tyto soubory automaticky zkopíruje do Silverlight projektu, což umožňuje umístit např. definice počítaných datových polí do sdílených souborů obsahujících tzv. částečné (partial) třídy. Jedinou podmínkou je, že tyto částečné třídy nemohou obsahovat kód závislý na částech datového jádra, které pro Silverlight nejsou implementovány.

Generování kódu Silverlight aplikace

Aby bylo možné korektně rekonstruovat metadata entity a zpřístupnit podobné rozhraní pro prezentační vrstvu i na klientské straně, je nutné provést drobné úpravy ve vygenerovaném kódu

entit. K těmto účelům existují v zásadě dvě možnosti. Drobné úpravy lze provést pomocí tzv. *T4 text template*, větší zásahy pak vytvořením vlastní třídy generátoru kódu. Příklad prvního postupu uvádí [25]. Druhý postup je založen na umístění speciální třídy implementující rozhraní `IDomainServiceClientCodeGenerator` do projektu s webovou službou. Tato třída by musela obsahovat implementaci celého procesu generování klientského kódu. Jak uvádí [26], lze ale celý postup úpravy generovaného kódu zjednodušit pomocí využití dostupného balíku *WCF RIA Services Toolkit – T4 Code Generation*. Po nainstalování balíku budou dostupné dvě nová assembly `Microsoft.ServiceModel.DomainServices.Tools` a `Microsoft.ServiceModel.DomainServices.Tools.TextTemplate`. Jmenný prostor `Microsoft.ServiceModel.DomainServices.Tools.TextTemplate.CSharpGenerators` pak obsahuje třídy implementující standardní způsob generování jednotlivých prvků klientského kódu. Výhodou ale je, že tyto třídy obsahují virtuální metody umožňující upravit generovaný výstup jednotlivých částí generovaného kódu. Základem je pak definice třídy odvozené od `CSharpClientCodeGenerator` opatřené atributem `DomainServiceClientCodeGenerator`. Tato třída je zodpovědná za vytvoření instancí generátorů pro jednotlivé prvky kódu. Zde bude využit pouze generátor kódu entit – `EntityGenerator`.

První změnou oproti původnímu vygenerovanému kódu je změna předka entity. Standardní generátor odvozuje třídu entity od třídy `System.ServiceModel.DomainServices.Client.Entity`. Framework ale pro správnou funkci získávání metadat potřebuje, aby byly entity odvozeny od třídy `BaseEntity` (kapitola 4.2).

Další úprava se týká datových polí. Standardní generátor vytváří gettery a settery vlastností a privátní pole stejného datového typu pro uložení hodnoty. Jak bylo uvedeno v kapitole 4.2, framework ale předpokládá použití speciálních objektů pro uchovávání hodnot datových polí. Výsledek upraveného generování kódu pro datové pole bude tedy vypadat například takto: (pozn.: nejsou uvedeny atributy datového pole)

```
public long ID
{
	get {
		return this._ID.Value;
	}
	set {
		if (this._ID.Value != value)
		{
			this.OnIDChanging(value);
			this.RaiseDataMemberChanging("ID");
			this.ValidateProperty("ID", value);
			this._ID.Value = value;
			this.RaiseDataMemberChanged("ID");
			this.OnIDChanged();
		}
	}
}

public PD.ISFramework.Data.Entities.DFLong _ID = new
PD.ISFramework.Data.Entities.DFLong();
```

5 Implementace frameworku

Tato kapitola se zabývá implementací částí frameworku navržených v předchozí kapitole. Jsou zde uvedeny především důležité oblasti implementace jako je implementace klíčových prvků datového jádra frameworku a dále některá netriviální použitá řešení.

5.1 Rozvržení projektů v solution

Základní logické rozčlenění částí frameworku již bylo uvedeno v předchozí kapitole. Struktura projektů v *solution* se zdrojovými soubory celého frameworku a vzorových aplikací odráží stanovená pravidla. Pro větší přehlednost jsou projekty seskupeny do složek.

- **Framework** – složka s projekty frameworku.
  - **Data** – projekt s datovým jádrem frameworku.
  - **Data.DomainServices** – projekt obsahuje podpůrné třídy k vytvoření webové služby pro použití s WCF RIA Services.
  - **Data.SL** – projekt s datovým jádrem frameworku pro Silverlight.
  - **UI.Silverlight** – obsahuje podporu uživatelského rozhraní pro Silverlight klienty.
  - **UI.WinForms** – obsahuje kompletní sadu všech částí frameworku spojených s uživatelským rozhraním v technologii Windows Forms (kapitola 4.6).
- **Demos** – složka s ukázkovými aplikacemi. Podrobnější popis je v kapitole 5.7.
  - **LoginDemo**
  - **PartnersDemo**
  - **SilverlightClient**
    - **TasksDemo.Data** – knihovna obsahující entity pro příklad.
    - **TasksDemo.SL** – klientská aplikace v Silverlight.
    - **TasksDemo.SL.Web** – projekt s webovou službou pro Silverlight.
    - **TasksDemo.WinForms** – klasická aplikace ve Windows Forms.
- **Tests** – složka obsahující unit testy frameworku.

Veškeré třídy projektů samotného frameworku jsou umístěny do následujících jmenných prostorů podle logických vazeb:

- `PD.ISFramework.Data` – základní a podpůrné objekty především pro inicializaci datového jádra frameworku.
- `PD.ISFramework.Data.BaseLib` – základní knihovna tříd, např. obecně použitelné entity apod.
- `PD.ISFramework.Data.DataSources` – zdroje dat a jejich podpůrné třídy a rozhraní.
- `PD.ISFramework.Data.Entities` – obsahuje nejdůležitější třídy pro práci s entitami (obecný předek entity `BaseEntity`, správce entit, implementace standardních datových polí).
- `PD.ISFramerork.Data.Metadata` – objekty pro reprezentaci metadat a speciální atributy k datovým polím, podřízeným modulům a entitám.
- `PD.ISFramework.UI.Silverlight` – podpůrné objekty pro Silverlight prezentační vrstvu.
- `PD.ISFramework.UI.WinForms` – veškeré podpůrné třídy, standardní a generované formuláře a komponenty pro technologii Windows Forms.
- `PD.ISFramework.UI.WinForms.BaseLib` – standardně dodávané obecně použitelné prvky uživatelského rozhraní (formuláře a moduly).

5.2 Datová pole

Framework ve svém základu obsahuje implementace několika typů datových polí. Jedná se o základní datové typy jazyka C# – `long`, `double`, `string`, `byte`, `bool`, `DateTime`. Vzhledem k tomu, že datová pole musí řešit elementární logiku definovanou v kapitole 4.2, není jejich implementace zcela triviální. Tuto logiku však lze z větší části implementovat jednotně, proto byla umístěna do společné abstraktní třídy `BaseDataField`, která částečně implementuje rozhraní `IDataField`. Toto rozhraní definuje obecnou vlastnost `Value` typu `object` sloužící ke čtení nebo změně aktuální hodnoty datového pole. To je nutné především pro interní práci s datovými poli. Pro externí účely je mnohem vhodnější pracovat s konkrétním datovým typem. Jednak odpadá neustálé přetypovávání a zvyšuje se úroveň kontroly správnosti kódu již při překladu, a rovněž nebude neustále docházet k tzv. *zabalení a vybalení*[11] hodnotových typů, což může poměrně znatelně snižovat výkon. Z těchto důvodů všechny standardní implementace datových polí dědí od generické třídy `BaseTypedDataField<T>`, která vlastnost `Value` předefinuje na typ `T`. Tento předek je optimalizován především pro implementaci datových polí zastřešujících hodnotové datové typy, lze ji ale použít pro libovolný datový typ. Třída definuje pouze dvě abstraktní metody, které musí potomek implementovat. Ostatní funkcionalita datového pole je již v něm nebo třídě `BaseDataField` implementována. Jedna z metod slouží ke konverzi hodnoty typu `object` na hodnotu interně uchovávaného typu. Tato metoda je volána především při deserializaci

[11] K balení (boxing) hodnotového typu dochází automaticky např. při konverzi na typ object. Ze své podstaty je tato operace poměrně „drahá“, proto ji je potřeba minimalizovat. Více informací o této problematice naleznete v [3].

hodnoty datového pole. Druhá slouží ke konverzi z textového řetězce, což využívají především komponenty uživatelského rozhraní. Obě abstraktní třídy `BaseDataField` i `BaseTypedDataField<T>` jsou veřejné a je možné je použít i k vlastní implementaci dalších typů datových polí.

Kromě vlastnosti `Value` definuje rozhraní `IDataField` další dvě vlastnosti pro změnu hodnoty datového pole. První z nich je `EmptyValue` typu `bool`. Hodnota `true` znamená, že datové pole neuchovává žádnou platnou hodnotu. To nastává v případě, kdy hodnota datového pole ještě nebyla zadána. Přiřazením hodnoty `true`, do této vlastnosti dojde k zapomenutí původní platné hodnoty. Vlastnost je rovně využívána při kontrole vyplnění povinných polí. Druhou ze zmíněných vlastností je `OldValue`, která slouží především pro sledování změn v hodnotě datového pole. Přiřazením nové hodnoty do vlastnosti `Value` dojde v datovém poli k zapamatování původní hodnoty a ta je přístupna prostřednictvím vlastnosti `OldValue`. Jakmile dojde k potvrzení hodnoty např. uložením celé entity je `OldValue` nastaveno na stejnou hodnotu jako `Value`.

Použití datových polí v entitách

Entity uchovávají hodnoty v datových polích, což jsou výše popsané speciální objekty. Instance datových polí jsou v entitě definovány jako prvky třídy. Aby byl dodržen správný návrh, měly by být všechny prvky třídy neviditelné zvenčí a pro přístup k nim vytvořeny vlastnosti třídy. Tento princip je využit i při definici datových polí entity. Příklad definice datového pole ve třídě entity vypadá takto:

```
[DataField]
[Required]
[Display(Name = "Název")]
[StringLength(200)]
public string Name
{
    get { return _Name.Value; }
    set { _Name.Value = value; }
}
private DFString _Name = new DFString();
```

Z ukázky je patrné, že samotné datové pole je definováno jako privátní a rovnou je vytvořena jeho instance. Veřejná vlastnost je již pouze typu, který datové pole interně uchovává, a v setteru a getteru probíhá nastavení resp. čtení vlastnosti `Value` datového pole. Rovněž atributy pro metadata (kapitola 4.4) jsou uvedeny u této veřejné vlastnosti, nikoliv u privátní instance datového pole. Důležitá je však jmenná konvence mezi názvem veřejné vlastnosti a privátním prvkem třídy s instancí datového pole.

5.3 Zdroje dat

Framework obsahuje dvě různé implementace zdrojů dat. Prvním a nejdůležitějším je `SqlDataSource`, který zpřístupňuje tabulku SQL databáze. Druhým je `MemoryDataSource` sloužící k dočasné serializaci entit do paměti.

SqlDataSource

Stěžejní činností zdroje dat `SqlDataSource` je sestavování SQL dotazů na základě poskytnutých metadat entity, která obsahují potřebné informace o sloupcích tabulky, primárním klíči a názvu tabulky. Výsledný SQL dotaz dodržuje základní syntaxi, která je nutná pro správnou funkci na různých databázových serverech. Pro účely sestavování SQL dotazů byl vytvořen pomocný nástroj v podobě třídy `QueryBuilder` a jejich konkrétních variant (`SelectQueryBuilder`, `InsertQueryBuilder`, `UpdateQueryBuilder`, `DeleteQueryBuilder`), jejichž metody definují snadno použitelné rozhraní pro definici výsledného dotazu. Ten je následně po patřičných úpravách např. v podobě uvození identifikátorů znakem odpovídajícím použitému databázovému serveru vrácen jako textový řetězec.

Celý framework a především SQL zdroj dat využívá pro přístup k databázi pouze základních prostředků technologie ADO.NET [27] bez žádných pozdějších nadstaveb, které by v navrženém schématu frameworku nepřinesly žádný užitek. Jedná se rovněž o nejefektivnější přístup k datům z databázového serveru. Pro připojení ke konkrétnímu databázovému serveru je vyžadován ADO.NET provider výrobce databázového serveru pro zajištění správné funkce generování SQL dotazů.

Automatické vytváření SQL tabulek

`SqlDataSource` disponuje doplňkovou funkcí automatického vytváření tabulek. Jako zdroj informací pro vytváření tabulek slouží opět metadata entit. Bylo však nutné řešit problém jednak různých názvů datových typů v různých databázových serverech, ale také problém rozdílné syntaxe příkazu `CREATE TABLE`. Z toho důvodu vzniklo rozhraní `IDbDriver` definující metody tzv. ovladače pro konkrétní typ databáze. Samotné vytváření příkazu `CREATE TABLE` je tak delegováno na objekt implementující toto rozhraní, které je opět veřejné a správnou instanci ovladače lze nastavit během inicializace frameworku. Tím je umožněno další rozšíření podporovaných databázových serverů.

Součástí frameworku jsou tři implementace – `DbDriverMSSQL`, `DbDriverMySQL` a `DbDriverPqSql`. Odpovídající databázové systémy (MS SQL Server, MySQL a PostgreSQL) jsou také standardně podporovány celým frameworkem.

Stránkování čtených dat

Zamýšlené využití frameworku předpokládá práci s velkým počtem záznamů v jednotlivých tabulkách. `SqlDataSource` tedy nemůže pracovat způsobem, kdy by najednou přečetl všechny řádky tabulky a následně je dále zpracovával a předával vyšším vrstvám. To by bylo v lepším případě neefektivní, protože jen málokdy je nutné pracovat ve vyšších vrstvách se všemi řádky tabulky (entitami) najednou. V horším případě, kdy by již počet řádků tabulky přesáhl únosnou mez, by tento přístup nebyl vůbec použitelný. Proto je nutné číst obsah tabulky po částech, což se často označuje pojmem *stránkování*. `SqlDataSource` čte vždy pouze maximálně 50 řádků z tabulky. Teprve v případě potřeby dalších záznamů spouští další SELECT dotaz a přečte následujících 50 řádků. Výhodou tohoto řešení je, že se značně omezí

počet zbytečně přečtených řádků tabulky, ale zároveň při enumeraci celé tabulky je toto průběžné načítání řádků zcela transparentní pro vyšší vrstvy frameworku.

Při implementaci stránkování bylo nutné řešit opět jeden rozdíl v syntaxi SQL dotazů u různých databázových systémů. Pro efektivní implementaci stránkování je nutné sdělit databázovému serveru přímo v SELECT dotazu, že je požadován jen omezený počet navracených řádků. Bohužel syntaxe tohoto omezení je rozdílná. První ukázka je pro MS SQL Server, druhá pro MySQL:

```
SELECT TOP 50 ... FROM ...

SELECT ... FROM ... LIMIT 50
```

K vyřešení tohoto rozdílu bylo opět využito ovladače pro konkrétní databázový systém (`IDbDriver`) popsaný v předchozí podkapitole.

5.4 Správce entit

Správce entit je jedna z nejkomplexnějších částí frameworku. Jeho implementace vyžadovala poměrně velké úsilí z důvodu množství funkcionality. Správce entit je implementován jako generická třída `EnityManager<TEntity>`. Umožňuje tedy spravovat jeden typ entity.

Základní práce s entitami

Správce entit funguje jako speciální typ kolekce. V prvé řadě implementuje rozhraní `IEnumerable<TEntity>`, které umožňuje jednoduché procházení všech entit. Během enumerace kolekce jsou postupně načítána data ze zdroje dat a deserializována do entit. Správce entit také obsahuje jednoduchou cache (vyrovnávací paměť), do níž jsou vkládány vytvořené instance entit. Tato paměť má význam především pro začátek enumerace od konkrétní entity, kdy je paměť naplněna již při načítání konkrétní entity. Velikost paměti je omezena na dvojnásobek počtu načítaných řádků v jednom bloku z SQL databáze. Tento počet entit v paměti se experimentálně ukázal jako optimální.

Správce entit se od běžných kolekcí liší především v tom, že kromě obousměrné enumerace přes všechny entity implementuje jistou obdobu návrhového vzoru *active record*. `EntityManager` si interně pamatuje poslední entitu, kterou zpracoval. Okolí ji zpřístupňuje pomocí vlastnosti `Current`. Jedná se tedy o jakýsi kurzor ukazující vždy na jednu entitu z celé kolekce. Dále poskytuje správce entit metody `MoveNext` a `MovePrevious`, které provedou jeden krok enumerace od aktuální entity odpovídajícím směrem. Dalšími metodami pro navigaci v kolekci jsou `SetFirst` a `SetLast`, které nastaví kurzor před první resp. za poslední entitu v kolekci. Tento způsob procházení kolekce umožňuje například různě měnit směr procházení, v případě potřeby se vrátit o krok nebo více zpět na předchozí entitu apod. Navíc v kombinaci s vyrovnávací pamětí entit bez potřeby spouštění dalších SQL dotazů. Hlavní výhodu ale poskytuje při implementaci uživatelského rozhraní a napojování komponent. Komponenta DataGrid může při přesunutí aktivního řádku rovnou provést změnu aktuální entity. Formulář,

který je napojený na stejnou instanci správce entit pak jednoduše zobrazí entitu z vlastnosti `Current`.

Poslední možností je přístup ke konkrétní entitě na základě hodnot jejich datových polí. K tomu slouží metoda `GetEntityByKey`. Pokud entita s odpovídajícími hodnotami existuje, je načtena, vyrovnávací paměť je naplněna okolními entitami a nakonec je nastavena ve správci entit jako aktuální. Pro účely nastavení klíčových hodnot obsahuje správce entit další vlastnost `Primary`, která vrací pomocnou instanci daného typu entity. V této pomocné entitě je možné nastavit hodnoty klíčových polí. Metoda `GetEntityByKey` pak tyto hodnoty použije a předá je zdroji dat a ten dále formou SQL dotazu databázovému serveru.

Paměťové editace podřízených modulů

Podřízené moduly entit jsou implementovány jako vlastnost entity typu správce entit. To ale znamená, že správce entit musí pracovat ve dvou režimech. První odpovídá doposud popsanému a slouží pro práci s kolekcí běžných (nadřízených) entit. V případě podřízeného modulu se správce entit musí chovat poněkud odlišně především během editace entit v něm obsažených. V kapitole 4.3 bylo vysvětleno chování entit v podřízených modulech a způsob jejich ukládání. Pro správce entit podřízeného modulu to znamená, že veškeré změněné entity si musí po uložení pouze zapamatovat v paměti a fyzicky je předat k uložení do zdroje dat až v okamžiku uložení nadřízené entity. Toto platí pro změnu entity, vytvoření nové a odstranění existující entity.

Základem implementace tohoto chování je úzká spolupráce mezi nadřízenou entitou a podřízeným správcem entit. Nadřízená entita musí sdělit všem svým podřízeným modulům, že došlo k jejímu fyzickému uložení a podřízené moduly musí na tento signál provést uložení všech změn v entitách. K tomuto účelu přibyla do veřejného rozhraní správce entit metoda `SaveInMemoryEdits`. Interně je pamatování změn řešeno třemi pomocnými kolekcemi uchovávající změněné entity, odstraněné entity a nové entity. Po fyzickém uložení změn ve správci entit jsou kolekce vyprázdněny.

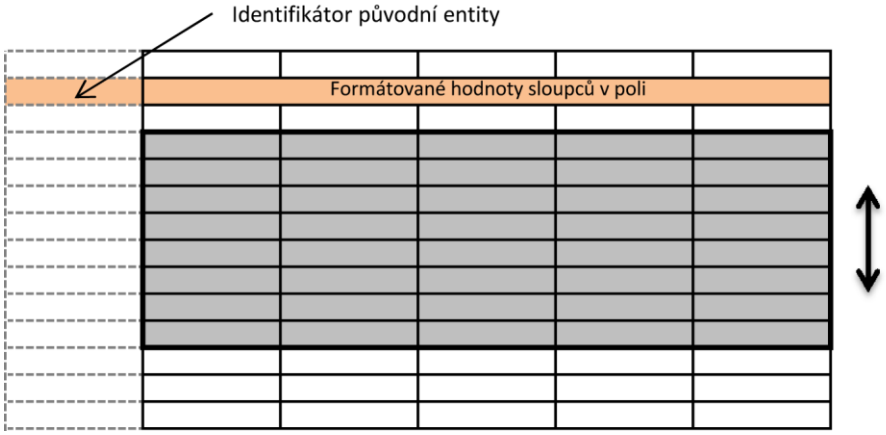
Druhá nutná úprava musela být provedena v implementaci enumerace entit. Pokud je nějaká entita změněna, je uložena do pomocné interní kolekce. Při enumeraci je ale získána původní uložená verze entity ze zdroje dat. Během deserializace entity je tak pokaždé nejprve prohledána pomocná kolekce se změněnými a odstraněnými entitami. Pokud je v těchto kolekcích stejná entita nalezena, je vrácena tato instance místo nově deserializované s původními hodnotami. Nakonec enumerace jsou doplněny nově vložené entity.

5.5 Komponenta DataGrid

Z implementace uživatelského rozhraní určitě stojí za zmínění komponenta `DataGridISF` pro zobrazení entit ze správce entit v tabulce. Komponenta vychází ze standardně dodávané komponenty `DataGridView`, která je přímo součástí platformy .NET. Tato komponenta má velmi široké možnosti použití, včetně editace zobrazených hodnot, možnosti přemísťování sloupců apod. Podrobný popis použití komponenty je k nalezení v [21] nebo [16]. Dokonce je možné využít jednoduchou vazbu dat na kolekci, kdy komponenta automaticky zobrazí všechny objekty z kolekce.

Tento přístup má ale jeden zásadní problém. Komponenta totiž před vykreslením projde celou předanou kolekci, aby zjistila kolik v ní je položek, a pro každou položku vytvoří svou interní reprezentaci jednotlivých řádků i buněk. Ve výsledku tak komponenta vytváří pro každý zobrazený řádek instance interních objektů, kterých je dohromady minimálně „počet sloupců“ + 1. Takový přístup již ale není možný v případě, že je entit velké množství (v řádu minimálně tisíců, což v podnikových IS není problém dosáhnout). Jednak by průchod všemi entitami zabral nepřípustnou dobu, ale hlavně by enormně rostly paměťové nároky komponenty. Naštěstí tvůrci komponenty `DataGridView` s touto možností počítali a implementovali v ní i tzv. *virtuální režim*. Zjednodušeně jde o to, že ve virtuálním režimu volá komponenta speciální událost pro každou zobrazenou buňku a implementace této události je zodpovědná za vrácení odpovídající hodnoty, která se v dané buňce zobrazí. Výhodou tohoto režimu tak je, že komponenta virtuálně zobrazuje velké množství řádku, ve skutečnosti ale potřebuje znát hodnoty pouze viditelných řádků. Až v případě, že dojde k posunu na další dříve neviditelné řádky, komponenta spustí událost pro každou buňku nově zobrazeného řádku. Implementace virtuálního režimu vychází z příkladu uvedeného v [28].

Největší problém, který při implementaci virtuálního režimu bylo nutno řešit, je mapování souřadnic buněk datagridu na konkrétní datové pole konkrétní entity z kolekce. Vzhledem k faktu, že událost pro získání hodnoty buňky (`CellValueNeeded`) je komponentou `DataGridView` volána poměrně často při každém překreslení komponenty, musí být navíc mapování velmi rychlé. Mapování čísla sloupce na datové pole entity není až tak náročná operace. Komponenta musí pouze znát pořadí zobrazených sloupců a název datového pole v každém sloupci. Následně, pokud již zná instanci entity pro daný řádek, pouze nalezne v entitě podle názvu datové pole a získá jeho hodnotu. Nicméně ani toto není zcela triviální operace. Neustálé vyhledání datového pole podle jeho názvu v instanci entity a také formátování jeho aktuální hodnoty jako textového řetězce může být znatelné. Přitom komponentě stačí ona konečná hodnota. Proto vznikla pomocná datová struktura pro uložení získaných hodnot, která pro každý řádek obsahuje pole se získanými hodnotami jednotlivých zobrazených sloupců a pak unikátní identifikátor entity, z níž byly hodnoty získány. Pole bylo zvoleno z důvodu vysoké efektivity přístupu ke složkám podle indexu (který odpovídá indexu sloupce v komponentě). Tato datová struktura (označena oranžově) je vytvořena pro každý viditelný řádek a několik nejbližších řádků okolo, jak ilustruje obrázek 5.1, a umístěna opět do obyčejného pole. Jedná se



Obrázek 5.1: Schéma maticové vyrovnávací paměti s hodnotami sloupců

tedy o jakousi jednoduchou cache v podobě matice již konkrétních naformátovaných hodnot určených pro zobrazení v komponentě. Pokud po jejím naplnění nedojde k většímu posunu viditelných řádků nebo zvětšení komponenty, jsou při překreslování potřebné hodnoty čteny přímo z této matice. K překreslování komponenty nebo jejích částí přitom dochází poměrně často – např. i při pohybu myší nad komponentou nebo při překrytí komponenty jiným oknem.

Matice hodnot je při prvotním zobrazení komponenty naplněna odpovídajícím počtem hodnot. V této situace je šedě zobrazené okno viditelných řádků zarovnáno se začátkem matice. Při plynulém posunu viditelných řádků dochází nejprve k myšlenému posuvu okna přes matici. Jakmile se okno přiblíží k opačnému konci matice, dojde k načtení následujících několika entit, naformátování jejich hodnot a vytvoření nových řádků matice. Velikost matice ale zůstává konstantní, takže je nutné původně načtené hodnoty posunout směrem nahoru. Vzhledem k tomu, že celá operace probíhá nad obyčejným polem hodnotových typů, je velmi rychlá. Během této operace se myšlené okno přesune o načtený počet nových řádků výše. Další posuv probíhá opět pouze oknem po matici až do chvíle, kdy se okno znovu přiblíží ke spodnímu okraji a celý postup se pak opakuje.

Pokud nějaká událost vyvolá velký posuv (např. prudké přetažení posuvníku), je pak celá matice naplněna znovu. V takovém případě je však nutné zjistit, která entita má být na prvním zobrazeném řádku viditelná. Proto byla doplněna do správce entit další metoda, která provádí nalezení entity podle jejího pořadí v kolekci. Jakmile je odpovídající entita nalezena, je matice naplněna stejným způsobem jako na začátku od prvního řádku.

Další vylepšení komponenty DataGridView

Komponenta je přímo vázána na správce entit. Její inicializace se provede přiřazením instance správce entit do vlastnosti `EntityManager`. Komponenta rovněž využívá metadat entit k automatickému určení zobrazovaných sloupců. Standardně komponenta vyhledá datová pole označená atributem `ImplicitColumnAttribute` (jeho podrobný popis je uveden v příloze A). Pokud žádné datové pole není označeno tímto atributem, zobrazí komponenta všechny sloupce.

Dále byly doplněny některé nové události. Jedna z nich je `DefineColumns` a slouží k explicitní definici zobrazených sloupců nebo jejich parametrů. Další užitečná událost je `RowActivate`, která sdružuje existující události do jedné. Je volána po dvojkliku na řádek nebo stisku klávesy Enter s vybraným některým řádkem. Před spuštěním události je ještě provedena synchronizace vybraného řádku se zdrojovým správcem entit. Korespondující entita je v něm nastavena jako aktuální (vlastnost `Current`, více v kapitole 5.4). Díky tomu je událost vhodná pro zpracování zobrazení formuláře s detailními informacemi o entitě.

5.6 Testování

Průběžné testování funkčnosti frameworku je velmi důležité. Má-li framework značně usnadňovat práci při vývoji konkrétních IS, je nutné, aby byl řádně odladěn a obsahoval čím jak nejméně chyb.

Průběžné testování implementovaných částí probíhalo na několika pomocných aplikacích, které v prvních fázích simulovaly chování doposud neimplementovaných částí frameworku. Zároveň s dokončením datového jádra frameworku bylo vytvořeno několik vzorových entit a odpovídajících dat, na nichž probíhal vývoj především dalších vrstev (např. uživatelského rozhraní). Hlavním prověřením frameworku bylo vytvoření ukázkových aplikací, jejichž popis je uveden v následující kapitole. Tyto aplikace představují reálné příklady použití framworku.

Dále bylo využito testovací tabulky obsahující cca. 600 000 řádků pro ověření a ladění výkonnosti nejprve datového jádra a následně především komponenty `DataGridISF`. Data sloužila rovněž k experimentálnímu stanovení vhodných velikostí vyrovnávacích pamětí ve správci entit apod. Dosaženým výsledkem je bezproblémová možnost zobrazit a procházet všechna data z tabulky v komponentě `DataGridISF`. V případě seřazení dat podle optimalizovaného databázového indexu (tzv. clusterovaného v terminologii MS SQL Serveru) je procházení řádků při pomalém i rychlejším pohybu posuvníkem téměř zcela plynulé. Většina jiných technologií nebo frameworků již má s takovým množstvím dat znatelné problémy.

K zajištění průběžné kontroly funkčnosti a odhalení potenciálních problémů byly využity nástroje vývojového prostředí MS Visual Studio 2010. Ověřování funkčnosti zajišťuje základní sada tzv. *unit testů*. Dále byl využit nástroj „*Code Analysis*“, který provádí statickou analýzu kódu a vyhledává v kódu potenciálně nebezpečné vzory, které by mohly způsobit problémy.

5.7 Ukázkové aplikace

Celkem byly vytvořeny tři ukázkové aplikace, přičemž každá z nich demonstruje některé odlišné prvky frameworku.

LoginDemo

Jedná se o nejjednodušší ukázkovou aplikaci ve Windows Forms demonstrující především základní postupy konfigurace a inicializace celého frameworku. Jediným modulem v aplikaci je jednoduchá správa uživatelů a jejich uživatelských oprávnění. Implementace modulu je umístěna v základní knihovně dodávané s frameworkem ve jmenném prostoru `PD.ISFramework.UI.WinForms.BaseLib`. Modul pro správu uživatelů je veden jako vestavěná součást frameworku, protože prakticky každý informační systém musí správu uživatelů a jejich oprávnění řešit. Dále pak příklad ukazuje možnost snadného vytvoření vlastního přihlašovacího formuláře, který systém využije pro přihlášení uživatele.

PartnersDemo

Demonstruje, jak by mohla vypadat reálná aplikace založená na frameworku. Jde o správu obchodních partnerů a kontaktů na ně. Aplikace se skládá z více modulů různé komplexnosti od jednoduchých číselníků pro definici partnerských států a pracovních pozic až po komplexní modul partnerů, který obsahuje panel s různými filtry a možnostmi seřazení, dodatečné uživatelské akce v hlavním panelu nástrojů a ručně navržený editační a zobrazovací formulář s vloženým datagridem pro editaci kontaktů k danému partnerovi. Příklad rovněž obsahuje hned několik různých entit s demonstrací obou typů vazeb mezi entitami. Posledním ukázkovým

prvkem je zcela odlišný domovský modul, který místo standardního datagridu vykresluje vlastní obsah do plochy pro obsah modulu.

TasksDemo

Je prototypová ukázka možnosti vytvoření třívrstvé architektury a sdílení aplikační logiky mezi různými prezentačními technologiemi. Jedná se o aplikaci pro zadávání a sledování plnění úkolů. Umožňuje po přihlášení zadat někomu úkol. Poté, co plnitel označí úkol jako splněný, musí zadavatel úkol schválit nebo jej vrátit zpět plniteli.

Celá ukázka se skládá ze 4 projektů. Projekt TasksDemo.Data obsahuje definici entit. Další projekt TasksDemo.WinForms obsahuje jednoduchou klientskou aplikaci v technologii Windows Forms, která kromě správy úkolu obsahuje i správu uživatelů systému. Zajímavostí této ukázky je demonstrace možnosti ovlivnění průběhu generování formulářů.

Zbylé dva projekty obsahují klientskou aplikaci technologie Silverlight a webovou aplikaci se službou. Silverlight aplikace umožňuje provádět s úkoly všechny operace stejně jako aplikace pro Windows Forms. Navíc je definice entit a základní aplikační logika mezi oběma typy klientů (a architektur) sdílena. Klientské aplikace tak opravdu plní pouze prezentační úlohu.

6 Možnosti dalšího vývoje

Již samotná povaha této práce – univerzální framework pro podporu vytváření celé řady informačních systémů – ji předurčuje k širokým možnostem a směrům budoucího vývoje. Tento fakt je dále umocněn propracovaným návrhem, který dovoluje přizpůsobit prakticky každou stěžejní část frameworku pro specifické potřeby řešeného IS. Následuje seznam některých možných směrů dalšího vývoje a vylepšení.

- Doplnění a odladění podpory dalších databázových systémů (např. Oracle).

- Implementace více specializovaných datových polí například pro další číselné typy, ukládání obrázků nebo binárních dat, případně nějakých komplexnějších struktur, jako je adresa.

- Vylepšení správce entit, aby umožňoval řazení a filtrování entit i podle hodnot počítaných sloupců. Nyní je toto umožněno pouze na základě hodnot z datového zdroje.

- Dalším velice užitečným vylepšením správce entit by byla alespoň částečná implementace rozhraní `IQueryable`, které umožňuje optimalizované zpracovávání řazení a filtrování entit ve výrazech integrovaného dotazovacího jazyka *LINQ*. Současná implementace využívá vlastní rozhraní pro definici filtrovacích podmínek, které jsou předány až na úroveň zdroje dat. `SqlDataSource` následně tyto filtrovací podmínky deleguje až do SQL dotazu. Implementace rozhraní `IQueryable` je poměrně náročná, ale výhody, které přináší, stojí za toto úsilí.

- Vysoce deklarativní způsob základní definice entit resp. jejich datových polí, kdy veškeré parametry pole jsou nastaveny pomocí atributů, nabízí možnost vytvoření nástroje umožňujícího automatické generování základního kódu entit podle sestaveného datového modelu aplikace. Nástroj by mohl poskytovat jednak uživatelské rozhraní pro vizuální návrh datového modelu, ale také by mohl umožnit vygenerovat kód entit podle schématu existující databáze.

- Výchozí uživatelské rozhraní technologie Windows Forms by mohlo být rozšířeno o další druhy komponent. Stávající komponenty vylepšeny (např. zlepšení signalizace zadání chybné hodnoty apod.). Velkých vylepšení by se také mohla dočkat komponenta `DataGridISF`. Může umožnit výběr a případně i ukládání nastavení sloupců za běhu aplikace nebo možnost snadného seřazení podle jednoho nebo více sloupců kliknutím na jeho záhlaví apod.

- Rozsáhlých vylepšení by také měla doznat naznačená cesta podpory vícevrstvé architektury a technologie Silverlight. I když Silverlight a některé jeho rozšiřující knihovny nabízejí značné množství komponent, včetně obdoby generovaných formulářů implementovaných ve frameworku pro Windows Forms, jejich napojení na framework

není zcela bezproblémové. Pro reálné využití by bylo vhodné podpůrnou knihovnu pro Silverlight rozšířit o vlastní komponenty korektně implementující data binding na entity a jejich datová pole a připravit vhodnou organizaci uživatelského rozhraní do modulů jako je tomu u Windows Forms. Možnosti pro zlepšení jsou i ve spolupráci mezi frameworkem a technologií WCF RIA Services. Mimo jiné by zde našla velké uplatnění již výše zmíněná podpora rozhraní `IQueryable` u správce entit, protože klientská strana WCF RIA Services dokáže LINQ dotazy přenést v požadavku na server a teprve tam dotaz vyhodnotit.

- Podstatné zvýšení možností využití frameworku přinese zavedení podpory dalších prezentačních technologií. Ať už pouze transformace stávající prezentační vrstvy na modernější desktopovou technologii WPF, ale také zcela nové technologie. Velice vhodné by bylo zavést propracovanou podporu pro webové aplikace – tedy technologii ASP.NET. Jedná se v podstatě o alternativu k realizované třívrstvé architektuře s využitím technologie Silverlight. Webová aplikace umožňuje přístup k IS ze širokého spektra různých platforem, včetně moderních mobilních zařízení.

- Pro použití v reálných aplikacích by dále bylo vhodné rozšířit základní knihovnu entit o další obecně použitelné typy – např. různé jednoduché číselníky se státy, měnami, kurzovní lístek apod.

- Zavedení podpory pro snadnou lokalizaci výsledné aplikace a množství dalších drobných úprav.

7 Závěr

Účelem první části této práce bylo analyzovat a stanovit požadavky na nástroj zjednodušující tvorbu podnikových informačních systémů. Tyto požadavky jsou i se stručným zdůvodněním formulovány v kapitole 2. Na jejich základě bylo provedeno otestování, zhodnocení a vzájemné porovnání několika běžně dostupných nástrojů a frameworků, které si kladou obdobný cíl. I když byly zkoumány velmi kvalitní nástroje, ukázalo se, že ani jeden z nich nesplňuje všechny stanovené požadavky.

Proto se druhá část práce věnuje návrhu vlastního řešení, které odstraňuje uvedené nedostatky dostupných nástrojů. Podrobný návrh řešení byl uveden ve 4. kapitole. Další kapitola se věnuje popisu některých postupů použitých při implementaci a 6. kapitola uvádí přehled některých možných směrů dalšího vývoje.

Výsledný produkt – framework – splňuje všechny stanovené požadavky a odstraňuje nalezené nedostatky existujících nástrojů. Mezi hlavní výhody navrženého a implementovaného produktu patří podpora různých typů zdrojů dat (různé relační databáze jsou jen jedním z typů zdrojů dat), snadná definice datového modelu aplikace přímo v kódu s využitím atributů pro konfiguraci parametrů, přímá podpora vazeb mezi entitami typu master-detail i odkazových vazeb, optimalizace pro práci s velkým množstvím dat, zcela automatizované vytváření prezentační vrstvy, ale především velmi vysoká míra přizpůsobitelnosti a rozšiřitelnosti frameworku pro potřeby konkrétní implementované aplikace.

Další významnou vlastností je možnost využití takto navrženého frameworku ve vícevrstvé architektuře. Podpora obou variant architektur (dvouvrstvá a třívrstvá) na úrovni frameworku poskytuje výhodu v podobě možnosti vytvoření uceleného řešení se sdílenou aplikační logikou mezi plnou verzí IS pro využití v interní firemní síti a odlehčenou variantou v podobě klientské aplikace technologie Silverlight pro přístup zvenčí přes internet. Vytvoření webové služby zpřístupňující operace s aplikační logikou navíc otevírá možnost pro další propojení např. s dnes stále rostoucím odvětvím mobilních aplikací.

Výsledek je použitelný pro reálné aplikace, významně přispívá k zefektivnění vývoje a poskytuje základ pro další rozšiřování možností. Stanovené stěžejní cíle této práce tak byly zcela naplněny.

Literatura

[1] Basl, Josef. *Podnikové informační systémy*: *podnik v informační společnosti*. 2., výrazně přepracované a rozšířené vyd. Praha: Grada, 2008, 283 s. ISBN 978-802-4722-795.

[2] *Tech-FAQ* [online]. c2011 [cit. 2011-12-28]. ERP Systems. Dostupné z: <http://www.tech-faq.com/erp-systems.html>.

[3] Nash, Tray. *C# 2010*: *rychlý průvodce novinkami a nejlepšími postupy*. Vyd. 1. Brno: Computer Press, 2010. 624 s. ISBN 978-802-5130-346.

[4] ECMA-335. *Common Language Infrastructure (CLI)*. 5th Edition / December 2010. Geneva: ECMA International, 2010. Dostupné z: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.

[5] Stair, Ralph M a George Walter Reynolds. *Principles of information systems*. 10th ed. United States: Course Technology Cengage Learning, c2011, 676 s. ISBN 05-384-7829-2.

[6] *MSDN Library* [online]. c2011 [cit. 2011-12-29]. ADO.NET Entity Framework. Dostupné z: < http://msdn.microsoft.com/en-us/library/bb399572.aspx>.

[7] *Online Documentation – Developer Express Inc.* [online]. c2011 [cit. 2011-12-30]. eXpressApp Framework Overview. Dostupné z: <http://documentation.devexpress.com/#Xaf/CustomDocument2558>.

[8] Brust, Andrew. *Visual Studio LightSwitch Technical White Paper: What is LightSwitch?*. [online]. c2011[cit. 2012-01-01]. Dostupné z: <https://www.microsoftvirtualacademy.com/tracks/building-business-apps-with-visual-studio-lightswitch>.

[9] Lacko, Luboslav. *Silverlight: Výukový průvodce tvorbou interaktivních aplikací*. Vyd. 1. Brno: Computer Press, 2010. 464 s. ISBN 978-80-251-2716-2.

[10] *Online Documentation – Developer Express Inc.* [online]. c2011 [cit. 2011-12-30]. eXpress Persistent Objects. Dostupné z: <http://documentation.devexpress.com/#XPO/CustomDocument1998>.

[11] *Cuberry* [online]. c2012 [cit. 2012-01-02]. Cuberry Overview. Dostupné z: <http://www.cuberry.net/Overview/tabid/55/Default.aspx>.

[12] *Oak Leaf Enterprises* [online]. 08/28/2011 [cit. 2012-01-02]. MM .NET Application Framework. Dostupné z: <http://www.oakleafsd.com/pgProducts_mmnet.htm>.

[13] *MM .NET 2010 Developer's Guide*. 04/21/10. Charlottesville: Oak Leaf Enterprises Solution Design, Inc., 2010. Dostupné z: <http://www.oakleafsd.com/pgMMNetDevGuide.htm>.

[14] Sharp, John. *Microsoft Visual C# 2010*: *krok za krokem*. Vyd. 1. Brno: Computer Press, 2010, 696 s. ISBN 978-802-5131-473.

[15] Froehlich, Garry, Wendy Liew, H. James Hoover a Paul G Sorenson. *Application Framework Issues when Evolving Business Applications for Electronic Commerce*. Edmonton, 1999. Dostupné z: <http://webdocs.cs.ualberta.ca/~softeng/papers/ecom.pdf>. University of Alberta.

[16] Nagel, Christian, Bill Evjen, Jay Glynn, Karli Watson a Morgan Skinner. *C# 2008*: *programujeme profesionálně*. Vyd. 1. Brno: Computer Press, 2009, 1126 s. ISBN 978-802-5124-017.

[17] Nagel, Christian, Bill Evjen, Jay Glynn, Karli Watson a Morgan Skinner. *Professional C# 4 and .Net 4*. Indianapolis, IN: Wiley Publishing, Inc., c2010, 1474 s. ISBN 978-0-470-50225-9.

[18] Bishop, Judith. C# návrhové vzory. Vyd. 1. Brno : Zoner Press, 2010. 328 s. ISBN 978-80-7413-076-2

[19] *MSDN Library* [online]. c2012 [cit. 2012-04-10]. AssociationAttribute Class. Dostupné z: <http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.associationattribute%28VS.95%29.aspx>.

[20] Anderson, Chris. *Pro Business Applications With Silverlight 5*. Apress, 2012, 670 s. 2nd Edition. ISBN 978-143-0235-002.

[21] Petzold, Charles. *Programování Microsoft Windows Forms v jazyce C#*. Vyd. 1. Překlad Karel Voráček. Brno: Computer Press, 2006, 356 s. ISBN 80-251-1058-3.

[22] Watson, Ben. *C# 4.0*: *řešení praktických programátorských úloh*. Vyd. 1. Brno: Zoner Press, 2010, 656 s. Encyklopedie Zoner Press. ISBN 978-80-7413-094-6.

[23] *MSDN Library* [online]. c2012 [cit. 2012-04-17]. DisplayAttribute Class. Dostupné z: <http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayattribute.aspx>.

[24] *MSDN Library* [online]. c2012 [cit. 2012-04-29]. System.Collections.Concurrent Namespace. Dostupné z: <http://msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx>.

[25] Meints, Willem. Using T4 to change the way RIA services work. In: *The Info Support blogcommunity* [online]. 8 February 2011 [cit. 2012-04-29]. Dostupné z: <http://blogs.infosupport.com/using-t4-to-change-the-way-ria-services-work/>

[26] Cruysberghs, Stefan. .NET - WCF RIA Services code generatie naar je hand zetten. In: SCIP.be [online]. 2011-06-12 [cit. 2012-05-05]. Dostupné z: <http://scip.be/index.php?Page=Welcome&Lang=NL>

[27] Price, Jason. *C#: programování databází*. Praha: Grada, 2005, 623 s. ISBN 80-247-0982-1.

[28] *MSDN Library* [online]. c2012 [cit. 2012-04-29]. Walkthrough: Implementing Virtual Mode in the Windows Forms DataGridView Control. Dostupné z: <http://msdn.microsoft.com/en-us/library/15a31akc.aspx>.

Seznam příloh

Příloha A

Přehled podporovaných atributů prvků entit

Tato příloha obsahuje kompletní výčet atributů a jejich parametrů, které framework zpracovává.

1 Atributy datových polí

DataFieldAttribute

Umístění: `PD.ISFramework.Data.Metadata`
Význam: Označuje klasické datové pole.
Parametry:

- *Caption* – zobrazovaný název datového pole.
- *FieldType* – interní reprezentace hodnoty datového pole (výčtový typ `DataFieldType` popsaný v kapitole 4.4).
- *IsPrimary* – datové pole je součástí primárního klíče entity.
- *Length* – délka pro datové pole. V případě textového řetězce určuje maximální počet znaků, pro reálná čísla určuje počet desetinných míst, v případě časových údajů určuje přesnost zobrazení.
- *LookupLinkId* – Identifikátor vazby, která vede z tohoto datového pole. Má význam pro dynamicky generované formuláře, kdy je pole s nastaveným platným identifikátorem vazby v tomto parametru zobrazeno jako lookup.
- *Multiline* – Má význam pouze u textových datových polí a označuje, že text může být víceřádkový. Ovlivňuje vlastnosti připojené komponenty `TextBoxISF`.
- *NoInsert* – Hodnota datového pole není serializována při ukládání nové entity
- *NoUpdate* – Hodnota datového pole není serializována při ukládání změn v existující entitě.
- *ReadOnly* – Datové pole je ve výchozím stavu jen pro čtení.
- *Required* – Zadání hodnoty datového pole je ve výchozím stavu povinné.
- *Source* – určuje zdroj hodnoty datového pole. Výchozí hodnota je `DataFieldSource.Storage`.

CalculatedDataFieldAttribute

Umístění: `PD.ISFramework.Data.Metadata`
Význam: Označuje počítané datové pole.
Parametry: Shodné s parametry atributu `DataFieldAttribute`, pouze vlastnost Source má implicitní hodnotu `DataFieldSource.Calculation`.

Atributy odvozené od ValidationAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Atributy provádějící validaci hodnoty datového pole.
Poznámka: Atributu je k validaci předána pouze nová hodnota datového pole.

RequiredAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Zadání hodnoty datového pole je ve výchozím stavu povinné.
Parametry: Popis parametrů je uveden MSDN dokumentaci - http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.requiredattribute.aspx
Poznámka: Atribut je ekvivalentem k nastavení vlastnosti `Required` atributu `DataFieldAttribute` na hodnotu `true`.

KeyAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Datové pole je součástí primárního klíče entity.
Parametry: Popis parametrů je uveden MSDN dokumentaci - http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.keyattribute.aspx
Poznámka: Atribut je ekvivalentem k nastavení vlastnosti `IsPrimary` atributu `DataFieldAttribute` na hodnotu `true`.

StringLengthAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Omezení délky zadaného textového řetězce.
Parametry: Popis parametrů je uveden MSDN dokumentaci - http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.stringlengthattribute.aspx
Poznámka: Hodnota parametru `MaximumLength` je ekvivalentem k nastavení vlastnosti `Length` atributu `DataFieldAttribute`.

EditableAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Určení výchozího stavu možnosti změny hodnoty datového pole.
Parametry: Popis parametrů je uveden MSDN dokumentaci - http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.editableattribute%28VS.95%29.aspx
Poznámka: Hodnota parametru `AllowEdit` je opakem hodnoty vlastnosti `ReadOnly` atributu `DataFieldAttribute`.

DisplayAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Určuje, zda má být datové pole zobrazeno v automaticky generovaných formulářích.
Parametry: Popis parametrů je uveden MSDN dokumentaci - http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayattribute.aspx
Poznámka: Implementovány jsou vlastností `AutoGenerateField`, `GroupName`, `Name`, `Order`, `ShortName`. Hodnoty ostatních vlastností jsou ignorovány.

ImplicitColumnAttribute

Umístění: `PD.ISFramework.Data.Metadata`
Význam: Označuje datové pole, které má být implicitně zobrazováno jako sloupec v komponentě `DataGridISF`.
Parametry:

- *Order* – pořadí sloupce.
- *WidthWeight* – poměrná hodnota vztažená k ostatním datovým polím s tímto atributem. Určuje poměr šířek jednotlivých sloupců. Pokud hodnota není zadána (je nula), pak je šířka sloupce určena automaticky podle zbývajícího místa a šířek ostatních sloupců.

2 Atributy podřízených modulů

DetailModuleAttribute

Umístění: `PD.ISFramework.Data.Metadata`
Význam: Označuje vlastnost entity, která zpřístupňuje spravovaný podřízený modul. Je nutný pro správnou činnost správy podřízených modulů.

AssociationAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Definuje mapování datových polí entit pro realizaci vztahu podřízeného a nadřízeného modulu. Je nutný pro správnou činnost správy podřízených modulů.
Parametry: Popis parametrů je uveden MSDN dokumentaci - http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.associationattribute%28VS.95%29.aspx
Poznámka: V případě, že vazba je realizována více datovými poli, jsou jejich názvy odděleny čárkou.

DisplayAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Určuje, zda má být podřízený modul v podobě datagridu zobrazen v automaticky generovaných formulářích.
Parametry: Popis parametrů je uveden MSDN dokumentaci - http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayattribute.aspx
Poznámka: Implementovány jsou vlastností `AutoGenerateField`, `GroupName`, `Name`, `Order`, `ShortName`. Hodnoty ostatních vlastností jsou ignorovány.

3 Atributy entity

EntityHeaderAttribute

Umístění: `PD.ISFramework.Data.Metadata`
Význam: Definuje doplňující informace o entitě.
Parametry:

- *Name* – zobrazovaný název entity.
- *ReadableKeyField* – název datového pole, jehož hodnota může být využita k vizuální identifikaci entity. Toto pole je například automaticky zobrazováno v lookupech. Pokud není vlastnost nastavena, je pro tyto účely použita hodnota primárního klíče entity.
- *TableName* – alternativní název tabulky pro ukládání entit daného typu. Má význam pouze při použití zdroje dat `SqlDataSource`.

Atributy odvozené od ValidationAttribute

Umístění: `System.ComponentModel.DataAnnotations`
Význam: Atributy provádějící validaci hodnot celé entity.
Poznámka: Atributu je k validaci předána celá instance entity, takže může dojít k validaci závislostí mezi hodnotami datových polí entity.