

Programátorská specifikace ročníkového projektu
Patterns4Net

Štěpán Šindelář

29. srpna 2010

Obsah

1	Úvod	3
1.1	Účel dokumentu	3
1.2	Stručný popis	3
1.3	Související dokumenty	3
2	Celková architektura	4
2.1	Použitá platforma a vývojové prostředí	4
2.2	Rozdělení na sestavy	4
2.3	Obecné principy, které budou uplatněny v celém projektu	5
2.3.1	Automatizované testy	5
2.3.2	Rozšiřitelnost	5
2.3.3	Kontrakty	5
2.3.4	Dokumentace vzorů	5
2.3.5	Nástroje pro kontrolu konvencí a statickou analýzu kódu	6
2.3.6	Integrační skript	6
2.3.7	Knihovna pro zpracování CIL	6
3	Sestavy Core a Patterns	7
3.1	Definice vzoru	7
3.2	Systém vyhledávání vzorů	7
4	Pattern Enforcer	8
4.1	Návrh tříd	8
4.1.1	<code>IChecker<T></code>	8
4.1.2	<i>Builder</i> objektů typu <code>IChecker<T></code>	8
4.1.3	Spárování <code>IChecker<T></code> a vzoru	8
4.1.4	Třída <code>PatternEnforcer</code>	9
4.2	Uživatelské rozhraní	9
4.2.1	Příkazová řádka a MSBuild úkol (task)	9
4.2.2	API pro automatizované testy	9
5	Architecture Explorer	10
5.1	Použité technologie pro GUI	10
5.2	Model-View-ViewModel (MVVM)	10
5.3	Návrh tříd	10
5.3.1	Zjišťování vztahů mezi třídami v analyzovaném kódu	10
5.3.2	Pohledy a jejich modely	11
5.4	Generování diagramů	11
6	DocGenerator	13
6.1	Návrh tříd	13
6.2	Uživatelské rozhraní	13
7	Pattern Recognizer	14
7.1	Rozšiřitelnost <i>Architecture Exploreru</i>	14
7.2	Spolupráce s <i>Pattern Enforcere</i> m	14

1 Úvod

1.1 Účel dokumentu

Tento dokument slouží jako programátorská specifikace ročníkového projektu s názvem Patterns4Net.

1.2 Stručný popis

Patterns4Net bude sada nástrojů určená pro platformu .NET usnadňující návrh a vývoj aplikací pomocí vzorů. Pojmem vzory jsou zde myšleny klasické návrhové vzory popsané v [1] a vzory popsané v [2], dále rozvinuté v [3] jako součást postupu nazvaného *Domain-driven design*.

1.3 Související dokumenty

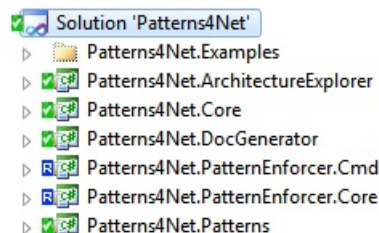
Souvisejícím dokumentem je Uživatelská specifikace projektu Patterns4Net [4], která specifikuje funkční požadavky projektu Patterns4Net. Dále se předpokládá, že čtenář je s funkčními požadavky již seznámen.

2 Celková architektura

2.1 Použitá platforma a vývojové prostředí

Sada nástrojů Patterns4Net bude vyvíjena na platformě .NET verze 4, v jazyce C# verze 4. Jako vývojové prostředí bude použito Microsoft Visual Studio 2010.

2.2 Rozdělení na sestavy



Obrázek 1: Rozložení projektů ve Visual Studiu.

Obrázek 1 ukazuje řešení (solution) Visual Studia, které bude použito pro vývoj Patterns4Net.

Ve složce Patterns4Net.Examples budou ukázkové projekty, které budou demonstrovat použití nástrojů Patterns4Net.

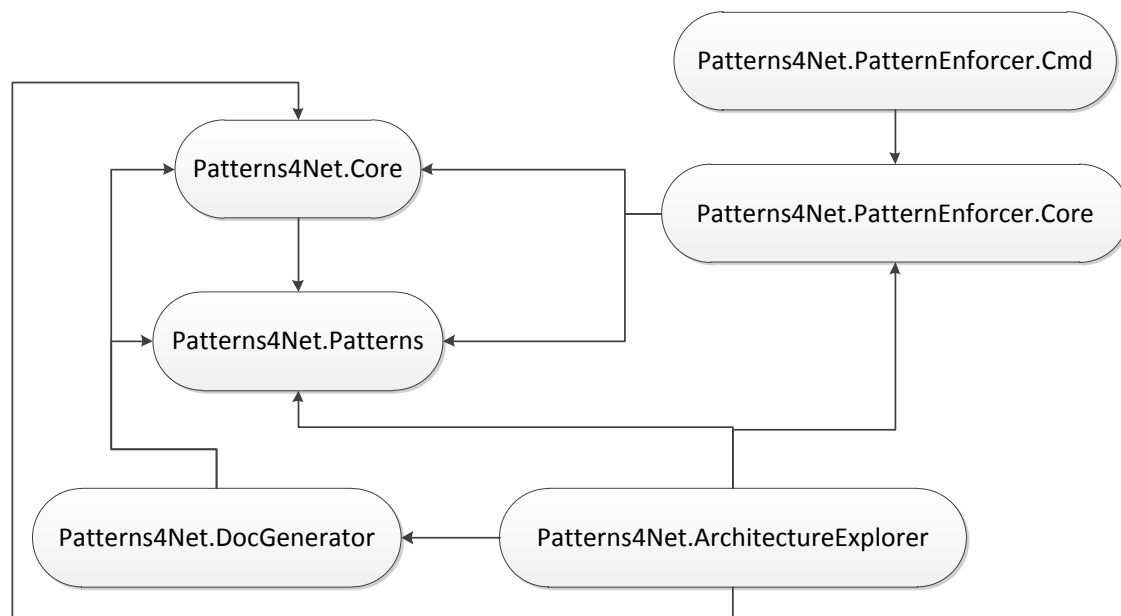
V projektu Patterns4Net.DocGenerator se bude nacházet implementace nástroje *DocGenerator*. Výstupem tohoto projektu bude program pro příkazovou řádku.

Sestavy začínající Patterns4Net.PatternEnforcer souvisejí s nástrojem *Pattern Enforcer*. Třídy zajišťující vlastní funkcionalitu *Pattern Enforceru* se budou nacházet v sestavě Patterns4Net.PatternEnforcer.Core. Tato část bude také obsahovat třídu úkolu (task) pro MSBuild a API pro volání *Pattern Enforceru* z automatizovaných testů (například NUnit testů). Důvodem je snaha nevytvářet zbytečně příliš mnoho projektů a tedy i výsledných sestav. Nicméně uživatelské rozhraní pro příkazovou řádku bude v samostatné sestavě Patterns4Net.PatternEnforcer.Cmd s příponou exe. Uživatelské rozhraní je odděleno, protože není obvyklé referencovat sestavu s příponou exe (v případě použití API pro automatizované testy) stejně jako načítat úkol (task) pro MSBuild ze souboru s touto příponou.

Projekt Patterns4Net.ArchitectureExplorer bude obsahovat zdrojový kód nástroje *Architecture Explorer* a *Pattern Recognizer*. Protože jak uživatelské rozhraní *Architecture Exploreru*, tak jeho jádro i *Pattern Recognizer* nebudou použity jinou sestavou, není třeba je rozdělovat do samostatných projektů, což ale neznamená, že tyto logické celky nebudou odděleny jinak.

Sestava Patterns4Net.Patterns bude obsahovat pouze definici vzorů a atributy pro dekoraci tříd v projektech, které budou používat nástroje Patterns4Net. Tato sestava bude obsahovat pouze minimum kódu a v případě, že si uživatel Patterns4Net vystačí pouze s předdefinovanými vzory, bude se jednat o jedinou sestavu, kterou bude muset ve svém projektu referencovat. Protože sestava Patterns4Net.Patterns bude obsahovat pouze jednoduché definice tříd, nebude třeba, aby využívala vlastnosti .NET verze 4, a proto bude vyvíjena tak, aby ji bylo možné přeložit i pro .NET verze 2, a díky tomu bude možné Patterns4Net použít i v projektech, které používají starší verze prostředí .NET.

Třídy, které budou mít na starosti identifikaci meta dat o vzorech, budou sídlit v sestavě Patterns4Net.Core. Meta data půjde získávat nejen z atributů, ale například i ze jmen tříd. Uživatel bude mít možnost implementovat vlastní logiku pro získání meta dat o vzorech.



Obrázek 2: Graf závislostí mezi sestavami.

2.3 Obecné principy, které budou uplatněny v celém projektu

Patterns4Net bude sada několika nástrojů a jak je vidět v předchozí sekci, vývoj bude probíhat na několika oddělených projektech (projekt ve smyslu Visual Studio), které spolu budou komunikovat přes veřejné rozhraní. Každý projekt bude řešit specifickou část problému, přesto bude skrz veškerý zdrojový kód Patterns4Net uplatněno několik obecných principů.

2.3.1 Automatizované testy

Každá třída bude pokryta automatizovanými testy, nejlépe jednotkovými testy, případně integračními testy. Pro testování bude použit framework NUnit [5]. Každý projekt (ve smyslu Visual Studio projekt) se jménem XY bude mít svůj XY.Tests projekt, který bude obsahovat testy. Pokud daný test bude integrační, bude označen atributem `[Category("Integration")]`.

2.3.2 Rozšiřitelnost

Pro lepší rozšiřitelnost a pro snazší uplatnění principu inverze závislostí bude Patterns4Net používat Managed Extensibility Framework (MEF) [6]. Uplatnění nalezne například pro přidávání další logiky pro získávání meta dat o vzorech.

2.3.3 Kontrakty

Většina tříd by měla definovat svůj kontrakt pomocí Code Contracts [7].

2.3.4 Dokumentace vzorů

Samotný projekt Patterns4Net půjde příkladem a při implementaci budou třídy opatřeny odpovídající dokumentací o implementovaných vzorech. Zdrojový kód Patterns4Net tak bude jedním velkým

ukázkovým příkladem využití nástrojů Patterns4Net.

2.3.5 Nástroje pro kontrolu konvencí a statickou analýzu kódu

Aby měly zdrojové kódy Patterns4Net jednotnou strukturu, budou vytvářeny tak, aby nástroj StyleCop¹ [8] při jejich analýze nenalezl žádnou chybu a nástroj FxCop² [9] žádnou kritickou chybu.

2.3.6 Integrovaný skript

Součástí celého projektu Patterns4Net by měl být i skript pro MSBuild který zkompile všechny sestavy, spustí automatizované testy a analýzu pomocí nástrojů FxCop a StyleCop.

2.3.7 Knihovna pro zpracování CIL

Pro načtení a parsování informací z .NET sestav bude použita knihovna Mono Cecil [10]. Toto rozhodnutí ovlivňuje kód celého projektu Patterns4Net, protože sestavy budou načteny pomocí Cecil do specifických datových struktur (jako například `Mono.Cecil.TypeDefinition`), se kterými potom bude Patterns4Net dále pracovat.

¹StyleCop provádí analýzu C# kódu a vyhledává provinění proti konvencím, například špatně umístěné složené závorky.

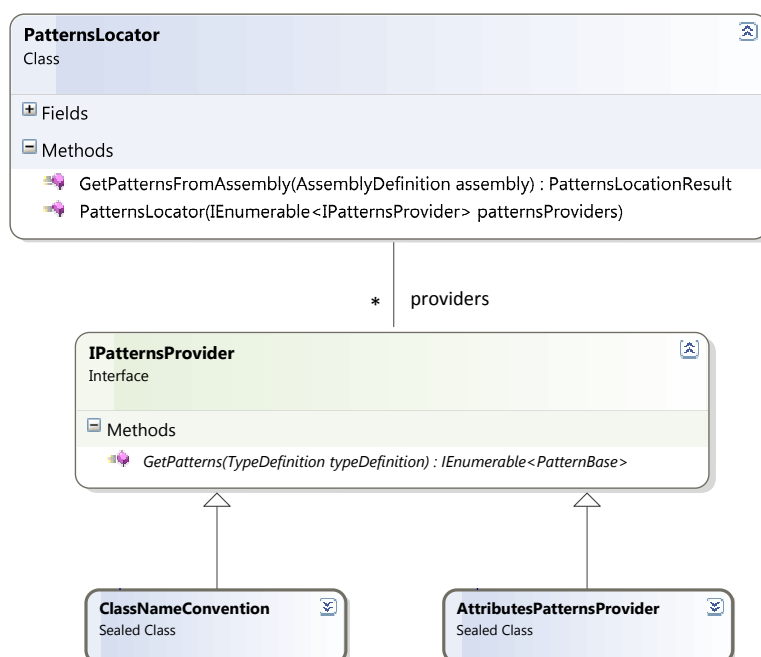
²FxCop provádí statickou analýzu CIL kódu a vyhledává potenciální problémy.

3 Sestavy Core a Patterns

3.1 Definice vzoru

Definice vzoru bude neměnná (immutable) třída, která bude dědit od třídy `PatternBase`, a bude obsahovat vlastnosti (properties), které budou obsahovat další informace o vzoru. Například vzor *Adapter* může mít jako nepovinnou informaci i typ adaptovaného rozhraní.

3.2 Systém vyhledávání vzorů



Obrázek 3: Diagram hlavních tříd v sestavě Patterns4Net.Core.

Na obrázku 3 je znázorněn diagram hlavních tříd v sestavě `Patterns4Net.Core`. Základní třídou je `PatternLocator`, která poskytuje metodu `GetPatternsFromAssembly`. Metoda `GetPatternsFromAssembly` vrací objekt typu `PatternsLocationResult`, což je pouze obálka na seznam dvojic typ a jeho vzory a metoda a její vzory. `PatternLocator` ve skutečnosti sám nevyhledává meta informace, ale tuto činnost deleguje na instance rozhraní (interface) `IPatternsProvider`. Seznam instancí `IPatternsProvider` získá `PatternsLocator` jako parametr konstruktoru.

Dvěma základními implementacemi `IPatternsProvider` budou třídy `AttributePatternsProvider`, která bude meta informace získávat z atributů prověřovaných tříd, a třída `ClassNameConvention`, která bude meta informace určovat podle jména třídy (například jméno třídy implementující vzor *Null Object* začíná na „Null“).

O „injektování“ instancí `IPatternsProvider` se postará `Managed Extensibility Framework`. Zmíněné třídy akorát musí být odekorenovány patřičnými atributy, pomocí kterých se `Managed Extensibility Framework` konfiguruje. Díky použití MEF bude umožněno přidat další implementace `IPatternsProvider` jakožto pluginy i pro samotné uživatele `Patterns4Net`.

4 Pattern Enforcer

4.1 Návrh tříd

4.1.1 IChecker<T>

Základním stavebním kamenem *Pattern Enforceru* je generické rozhraní (interface) `IChecker<T>` s jedinou metodou `Check(T)`. Konkrétní implementace budou mít jako generický parametr `T` typy `Mono.Cecil.TypeDefinition` a `Mono.Cecil.MethodDefinition`, protože „kontrola“ bude probíhat na třídách i jednotlivých metodách. Všechny implementace `IChecker<T>` budou neměnné (immutable). Následuje seznam nejdůležitějších implementací rozhraní `IChecker<T>` s krátkým popisem.

`CompositeChecker<T>`. Tato generická třída bude implementovat vzor *Composite* a svou práci dál delegovat na seznam objektů typu `IChecker<T>`, který získá jako parametr konstruktoru.

`ClassMethodsChecker : IChecker<TypeDefinition>` bude jako parametr konstruktoru brát predikát (lambda výraz, který vrací `bool`), podle kterého budou vybrány metody daného typu, které budou zkontrolovány objektem typu `IChecker<MethodDefinition>`, jež bude druhým argumentem konstruktoru.

`PredicateClassChecker` a `PredicateMethodChecker`. Tyto třídy budou umožňovat vytvořit instanci `IChecker<T>` z lambda výrazu. Parametrem tohoto lambda výrazu nebudou `Mono.Cecil.TypeDefinition`, ani `Mono.Cecil.MethodDefinition`, nýbrž vlastní typy s názvy `TypeDescription` a `MethodDescription`, které budou obsahovat pouze základní informace jako jméno, viditelnost, virtualitu, atp. Poskytnutí pouze základních informací má naznačit, že `PredicateClassChecker` a `PredicateMethodChecker` jsou vhodné pouze pro jednoduché případy. Navíc pokud bude chtít uživatel vytvořit vlastní vzor a k němu pouze jednoduchý „checker“, nebude muset referencovat knihovnu `Mono.Cecil`.

Další specifické implementace `IChecker<T>`. Součástí *Pattern Enforceru* budou i další implementace `IChecker<T>` zaměřené na konkrétní potřeby kontroly vzorů jako například třída s názvem `DoesNotChangeInternalState`, která bude kontrolovat, zda daná metoda nemění interní stav objektu. Tyto třídy budou plně využívat možnosti knihovny `Mono.Cecil`, především možnost získat seznam CIL instrukcí, které tvoří tělo metody.

4.1.2 Builder objektů typu IChecker<T>.

Protože vytváření komplexních objektů typu `IChecker<T>`, které budou komponovat několik dalších instancí `IChecker<T>`, nebude přehledné (díky jejich neměnnosti – všechny vlastnosti jsou parametrem konstruktoru), bude pro tento účel vytvořena třída `ClassCheckerBuilder`, která bude poskytovat tzv. fluent rozhraní jako v následující ukázce:

```
builder.For.AllMethods.DoesNotChangeInternalState();
builder.For.VirtualMethods.AreOverriden();
builder.Type.IsSealed();
var checker = builder.Build();
```

4.1.3 Spárování IChecker<T> a vzoru

O nalezení správného „checkeru“ k danému vzoru se bude starat objekt implementující rozhraní `IPatternCheckerProvider` s jedinou metodou `GetCheckerFor(PatternBase)`.

Defaultní implementace rozhraní `IPatternCheckerProvider`, třída `MEFPatternCheckerProvider`, bude spoléhat na Managed Extensibility Framework. Konkrétně MEF umožňuje k exportovaným typům přidat další meta-informace, což v tomto případě využijeme tak, že k třídám typu `IChecker<T>` určeným pro kontrolu nějakého vzoru přidáme speciální atribut jako v následující ukázce:

```
[ExportPatternChecker(typeof(Composite))]  
public class CompositeChecker : IChecker<TypeDefinition>  
{  
    // ...  
}
```

kromě toho pak bude uživatelům snadno umožněno překrýt defaultní kontrolu tak, že navíc ve své implementaci uvedou parametr `OverridesDefault`.

```
[ExportPatternChecker(typeof(Composite), OverridesDefault=true)]  
public class MyCompositeChecker : IChecker<TypeDefinition>  
{  
    // ...  
}
```

4.1.4 Třída `PatternEnforcer`

Hlavním bodem veřejného rozhraní *Pattern Enforceru* bude stejnojmenná třída `PatternEnforcer`. Tato třída bude poskytovat metody:

- `CheckAssembly(string)` – načte sestavu na zadané cestě a provede na ní kompletní kontrolu.
- `CheckTypeImplements<TPattern, T>()` – zkontroluje, zda typ `T` implementuje vzor `TPattern`.
- `LoadConfiguration(string)` – načte konfigurační xml soubor na zadané cestě a konfiguraci aplikuje.

4.2 Uživatelské rozhraní

4.2.1 Příkazová řádka a MSBuild úkol (task)

Tato uživatelská rozhraní budou plně využívat třídu `PatternEnforcer`. Program pro příkazovou řádku nebude používat žádnou knihovnu pro zpracování argumentů příkazové řádky a tato funkcionality v něm bude implementována přímo. Další knihovna by zbytečně zvýšila počet závislostí `Patterns4Net`, přitom zpracování argumentů *Pattern Enforceru* nebude tak složitý úkol.

4.2.2 API pro automatizované testy

API pro automatizované testy bude poskytovat metody pro kontrolu vzorů, které budou v případě chyby vyhazovat výjimky. Bude se jedna o následující metody:

```
PatternAssert.Implements<Immutable, MyClass>();  
PatternAssert.CheckAssemblyOf<MyClass>();
```

první z nich zkontroluje, zda druhý generický parametr implementuje vzor zadaný pomocí prvního generického parametru. Druhá metoda spustí kompletní kontrolu *Pattern Enforceru* podle meta dat tříd a metod v sestavě, ve které se nachází typ zadaný jako generický parametr.

5 Architecture Explorer

5.1 Použité technologie pro GUI

Pro grafické rozhraní *Architecture Exploreru* bude použita technologie WPF. Kromě standardních součástí WPF budou dále použity knihovny AvalonDock [11] pro vytváření dokovatelných panelů a Ribbon Control Suite [12] pro tvorbu menu ve stylu Office 2007.

5.2 Model-View-ViewModel (MVVM)

Základním principem, podle kterého bude vyvíjeno uživatelské rozhraní, je architektonický vzor Model-View-ViewModel. Jedná se o variantu vzoru Model-View-Presenter. Podrobný popis lze nalézt v článku [13].

Zjednodušeně řečeno, ViewModel je třída, která poskytuje data pro zobrazení v podobě vlastností (properties) a akce v podobě obyčejných metod. Díky data-bindingu se vlastnosti ViewModelu automaticky zobrazí ve View kdykoliv se změní a metody jsou naopak zavolány, kdykoliv je ve View spuštěna odpovídající akce jako například stisknutí tlačítka. Tím pádem je ViewModel jednoduchá C# (není vůbec svázána s prezentační technologií), kterou lze snadno otestovat, a přitom řídí celou logiku GUI.

Existuje několik Model-View-ViewModel frameworků pro WPF. Pro vývoj Patterns4Net bude použit framework Caliburn.Micro (viz [14]).

5.3 Návrh tříd

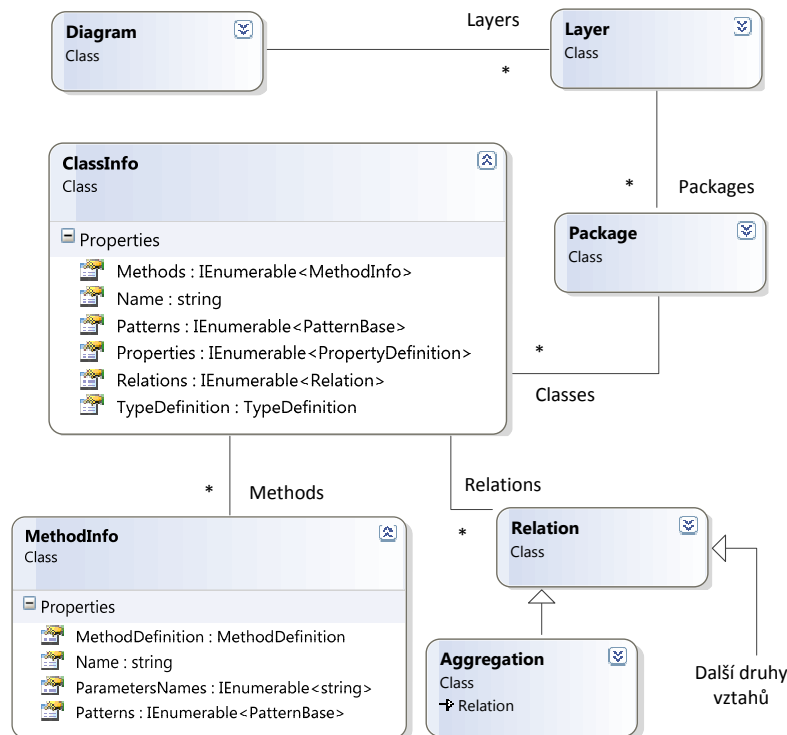
5.3.1 Zjišťování vztahů mezi třídami v analyzovaném kódu

Na obrázku 4 je znázorněna hierarchie tříd, která bude zachycovat diagram vrstev, balíčků a tříd. O načtení diagramu se bude starat defaultní implementace rozhraní `IDiagramLoader`, jejíž metoda `Load(string):Diagram` načte diagram pomocí Mono.Cecil.

Vztahy mezi třídami. Vztahy mezi třídami budou zachycovat instance potomků třídy `Relation`. Zavedení další hierarchie dědičnosti v tomto případě má smysl, protože WPF vybírá šablonu, pomocí které zobrazí instanci nějakého objektu, podle typu objektu a tím pádem bude snadno umožněno mít pro každý vztah jinou šablonu. Navíc některé specifické druhy vztahů mohou obsahovat další informace, které „jejich“ šablona může zobrazit.

Zjišťování vztahů. Vztahy nebude zjišťovat přímo `DiagramLoader`, ale tuto odpovědnost bude delegovat na seznam instancí rozhraní (interface) `IRelationsProvider` s jedinou metodou `GetRelations(TypeDefinition, IList<Relation>)`, která bude objevené vztahy ukládat do parametrem předaného seznamu. Tímto způsobem bude umožněno přepisovat vztahy, které již objevil nějaký méně specifický `IRelationsProvider`, jejich konkrétnější verzí. Tím pádem ale bude záležet na jejich pořadí. Správného pořadí bude docíleno pomocí meta dat Managed Extensibility Frameworku – každá třída implementující `IRelationsProvider` bude odekorována atributem uvádějícím prioritu dané třídy při exportu do objektu `DiagramLoader`.

Základními implementacemi `IRelationsProvider` budou třídy `AggregationProvider`, `CompositionProvider`, `UsesProvider` a `PatternRelationsProvider`. Poslední jmenovaná třída bude vztahy zjišťovat pomocí reflexe podle vlastností (properties) daného vzoru. (Pokud bude obsahovat vlastnost typu `System.Type`, pak bude brána jako vztah s tímto typem.)



Obrázek 4: Diagram tříd, které budou tvořit diagramy Architecture Exploreru.

5.3.2 Pohledy a jejich modely

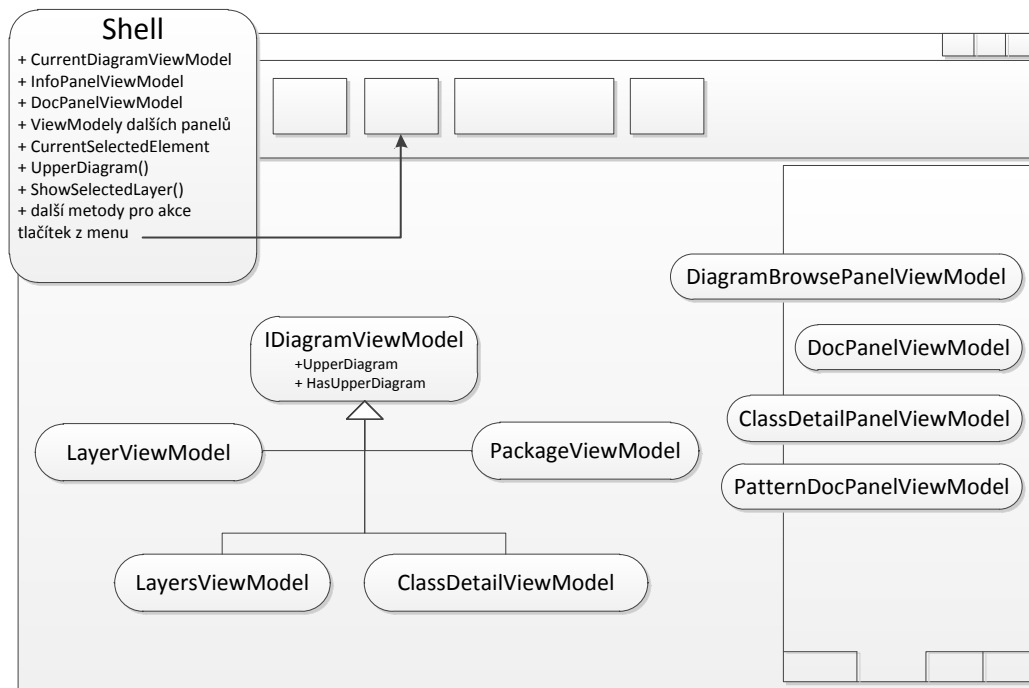
Obrázek 5 znázorňuje rozložení grafického uživatelského rozhraní a ViewModely umístěné zhruba tam, kde budou pomocí svého View zobrazeny. Hlavní třídou agregující celé uživatelské rozhraní je třída *Shell*.

Třída *Shell* bude obsahovat vlastnost (property) *CurrentDiagramViewModel*, která bude určovat aktuálně zobrazenou část diagramu. Framework Caliburn Micro se postará o to, aby při každé změně této vlastnosti došlo k překreslení hlavní části okna pomocí odpovídající šablony. Dále bude mít *Shell* veřejnou vlastnost (property) *CurrentElement*, která bude označovat aktuálně vybraný element z celého diagramu a především událost *OnCurrentElementChanged*. Logiku toho, jaký element je právě aktuální, bude řídit *Shell* tak, že si zaregistruje obsluhu událostí *OnSelectionChanged* jednotlivých ViewModelů, které budou umožňovat výběr elementu. Na druhou stranu panely, které budou zobrazovat nějakou informaci o aktuálním elementu, si zaregistrují zmíněnou událost *OnCurrentElementChanged* třídy *Shell*. V tomto smyslu lze *Shell* označit jako *Mediator*.

5.4 Generování diagramů

Diagramy budou generované, takže bude potřeba vygenerovat i jejich rozložení tak, aby vypadaly „hezky“.

O rozmístění WPF prvků se postará vlastní uživatelský WPF layout, ten zjistí na jaké pozici má prvky umístit od objektu typu *IDiagramLayoutService*.



Obrázek 5: Grafické uživatelské rozhraní a ViewModely.

Defaultní implementací rozhraní `IDiagramLayoutService` bude třída `QuickGraphLayoutService`, která za tímto účelem použije knihovny `QuickGraph` [15] a `GraphViz` [16].

6 DocGenerator

6.1 Návrh tříd

Hlavní třída ponese stejné jméno, tedy `DocGenerator`. `DocGenerator` načte xml soubor s dokumentačními komentáři do paměti, jednoduše použije funkcionality, kterou poskytuje třída `PatternsLocator`, nalezne vzor a k němu odpovídající element v xml souboru a obě dvě tyto instance předá objektu typu `IDocProvider`.

Rozhraní (interface) `IDocProvider` bude poskytovat metodu `InsertDoc(XmlElement, PatternBase)`, která by do zadaného xml elementu měla přidat dokumentaci zadaného vzoru, a metodu `GetDoc(PatternBase)`, kterou bude používat *Architecture Explorer* pro zobrazení nápovědy.

Defaultní implementací rozhraní `IDocProvider` bude třída `XmlDocProvider`, která bude nápovědu načítat z xml souboru. Obsah nápovědy bude získán z Wikipedia.org.

6.2 Uživatelské rozhraní

Stejně jako *Pattern Enforcer* nebude `DocGenerator` používat speciální knihovnu pro zpracování argumentů příkazové řádky.

7 Pattern Recognizer

7.1 Rozšiřitelnost *Architecture Exploreru*

Pattern Recognizer má pomocí *Pattern Enforceru* generovat odhad, že nějaká třída implementuje nějaký vzor, a potom jej zobrazit v *Architecture Exploreru*.

To znamená, že potřebujeme rozšířit třídu `ClassInfo` v hierarchii diagramu, tak aby mohla nést dodatečnou informaci, kterou pak její WFP šablona vykreslí. Rozšíření šablony nelze udělat příliš obecně, ale rozšíření všech elementů v hierarchii bude provedeno následovně.

Společný předek všech tříd tvořících hierarchii diagramu, `DiagramElement`, bude definovat metody `AddTag(object)` a `GetTag<TTag>()`. První metoda přidá zadaný objekt do slovníku tagů a druhá vrátí tag požadovaného typu. Tímto způsobem bude *Pattern Recognizer* přidávat k třídám vlastní data jako objekty typu `PatternRecognizerInfo`.

Samotný průchod třídami diagramu bude realizován klasickou kombinací vzorů *Composite* a *Visitor*, i když konkrétně *Pattern Recognizer* bude „navštěvovat“ pouze uzly typu `ClassInfo`, nicméně tato možnost rozšiřitelnosti by se mohla hodit do budoucna.

7.2 Spolupráce s *Pattern Enforcerem*

Výsledkem analýzy daného typu nebo metody pomocí objektu `IChecker<T>` nebude pouze `boolean`, ale počet prověřených instancí aspektů daného vzoru celkem a počet těch, které požadavky splňují. Celá tato informace bude zapouzdřena v třídě `CheckerResult`. Třída `PatternRecognizer`, která bude zároveň zmíněným *Visitem*, použije tuto informaci pro vytvoření svých výsledků.

Reference

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.
- [2] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.
- [3] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Longman, 2004.
- [4] Š. Šindelář, “Specifikace ročníkového projektu Patterns4Net.” 2010.
- [5] “Nunit - home.” <http://www.nunit.org/>, Aug. 2010.
- [6] “Managed extensibility framework.” <http://mef.codeplex.com/>, Aug. 2010.
- [7] “Contracts - microsoft research.” <http://research.microsoft.com/en-us/projects/contracts/>, Aug. 2010.
- [8] “Stylecop.” <http://stylecop.codeplex.com/>, Aug. 2010.
- [9] “Fxcop.” <http://msdn.microsoft.com/en-us/library/bb429476.aspx>, Aug. 2010.
- [10] “Mono - cecil.” <http://www.mono-project.com/Cecil>, Aug. 2010.
- [11] “Avalondock.” <http://avalondock.codeplex.com/>, Aug. 2010.
- [12] “Fluent ribbon control suite.” <http://fluent.codeplex.com/>, Aug. 2010.
- [13] J. Smith, “Wpf apps with the model-view-viewmodel design pattern.” <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>, Aug. 2010.
- [14] “Caliburn micro: A micro-framework for wpf, silverlight and wp7.” <http://caliburnmicro.codeplex.com/>, Aug. 2010.
- [15] “Quickgraph, graph data structures and algorithms for .net.” <http://quickgraph.codeplex.com/>, Aug. 2010.
- [16] “Graphviz - graph visualization software.” <http://www.graphviz.org/>, Aug. 2010.