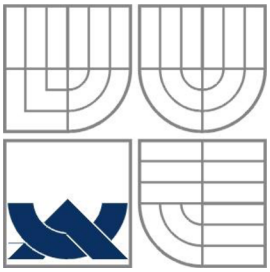
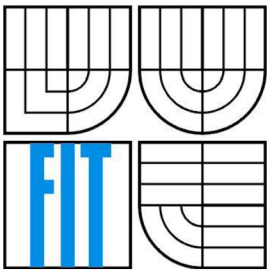


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ ZDROJOVÝCH KÓDŮ APLIKACÍ POMOCÍ NÁVRHOVÝCH VZORŮ

CODE GENERATION USING DESIGN PATTERNS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. FRANTIŠEK HANÁK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. PETER JURNEČKA

BRNO 2012

Abstrakt

Tato práce se zabývá generováním zdrojových kódů aplikací pomocí návrhových vzorů. Práce popisuje problematiku specifikace návrhových vzorů a jejich užití při generování kódů aplikací. Podstatná část práce se věnuje popisu návrhových vzorů, jejich dělení, účelu použití, ale i způsobům specifikace vzorů. Detailně popisuje nejčastěji používané formální specifikace návrhových vzorů, jejich možnosti využití při generování kódu a návrh algoritmu pro vyhledání podobných struktur vzorů ve zdrojovém kódu.

Abstract

This thesis describes code generation using design patterns. It deals with questions of specification of design patterns and their usage in code generation. The main part of thesis follows describtions of design patterns, their categorization, usage purpose and main ways of design patterns definitions. It describes the most often used formal design patterns specifications, their possible usage in code generation and design of algorithm for searching similar structures of patterns in source code in detail.

Klíčová slova

návrhové vzory, formální specifikace, SPINE, BPSL, GEBNF, generování kódu, CodeDom, .NET, refaktorizace

Keywords

design patterns, formal definition, SPINE, BPSL, GEBNF, code generation, CodeDom, .NET, refactoring

Citace

Hanák František: Generování zdrojových kódů aplikací pomocí návrhových vzorů, diplomová práce, Brno, FIT VUT v Brně, 2012

Generování zdrojových kódů aplikací pomocí návrhových vzorů

Prohlášení

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

……
František Hanák
21.5.2012

Poděkování

Děkuji svému vedoucímu panu Ing. Peterovi Jurnečkovi za odbornou pomoc a časovou dostupnost. Dále bych rád poděkoval své rodině a přítelkyni za psychickou podporu při zpracování diplomové práce.

Obsah

1 Úvod

Již od vynalezení počítačů se lidé snaží co nejvíce využít jejich potenciál. Tím, jak se mění doba a informační technologie jsou stále více vyspělejší, stávají se běžnou součástí našeho života. Důsledkem je, že aplikace jsou komplexnější a jsou na ně kladeny čím dál větší nároky. Vytvoření aplikace už není jen programování, je to celý proces, který zahrnuje více aspektů. Tento proces se nazývá softwarové inženýrství.

Na softwarové inženýrství je možné pohlížet jako na proces, který se skládá z dalších menších, specifických podprocesů, např. management projektu, proces tvorby softwaru (analýza, návrh, testování, programování), poučení se z chyb apod. Lze je chápat jako činnost tvorby softwarových systémů, které jsou tak velké a komplexní, že jsou vytvářeny v týmu. V posledních desetiletích se v softwarovém inženýrství rozšířil zejména objektově orientovaný přístup k modelování systému pomocí jazyka UML (Unified Modelling Language). Čím dál větší důraz je kladen na efektivnost tvorby systému a jeho znovupoužitelnost, protože v době, kdy se požadavky na systém mění velmi často, lze bez tohoto přístupu efektivity dosáhnout velmi těžko. Při vývoji se se hodně používají návrhové vzory, díky kterým se tak aplikace stávají flexibilnějšími a jednotlivé komponenty je možné jednoduše znovu použít v jiných systémech.

Zpočátku byly návrhové vzory užívány k řešení problémů samostatně, nezávisle na ostatních. Později se ale ukázalo, že je výhodnější používat návrhové vzory jako mikro-architektury, které jsou různě vhodným způsobem kombinovány a společně tvoří komponentu aplikace nebo celý systém jako takový. V dnešní době jsou vzory většinou definovány neformálně, tj. slovním popisem, UML diagramem nebo ukázkou zdrojového kódu. Počet vzorů se ale neustále zvyšuje a tím přibývají problémy s porozuměním definic vzorů. Ukázalo se, že neformální definice nejsou vždy jednoznačné a vedou k chybám způsobených špatným nebo odlišným pochopením vzoru, a proto lze pozorovat nový trend, formální definice vzorů.

Tato práce se zabývá problematikou jednoznačnosti definice vzorů, vysvětluje možnosti formální specifikace návrhových vzorů a porovnává jednotlivé způsoby reprezentace vzorů z hlediska využitelnosti při generování kódu aplikací. Cílem je vytvoření aplikace, která načte definici vzoru, v již existujícím projektu vyhledá vhodné třídy pro aplikaci vzoru, nabídne je uživateli a následně provede refaktorizaci projektu aplikací daného vzoru.

První kapitola obsahuje uvedení do problematiky a popis kapitol. Druhá kapitola stručně vysvětluje návrhové vzory, jejich možnosti dělení a způsob řešení problémů. Třetí kapitola pojednává o problematice specifikace návrhových vzorů. Čtvrtá kapitola popisuje principy generování kódu v jazyce C#. Pátá kapitola obsahuje popis návrhu aplikace. Šestá kapitola vysvětluje implementaci aplikace a sedmá kapitola popisuje testování. Osmá kapitola shrnuje dosažené poznatky, které vyhodnocuje, a pojednává také o možnosti budoucího rozšíření.

2 Návrhové vzory

Návrhové vzory (Design Patterns) jsou doporučené postupy a osvědčená řešení často nebo opakovaně se vyskytujících problémů v objektově orientovaném (OO) návrhu. Tato řešení jsou obecného charakteru, nezávislá na architektuře či programovacím jazyku a zajišťují společně s OO návrhem znovu použitelnost komponent systému. Vzory jsou obvykle popsány strukturovaně a tvoří tzv. katalog. Návrhový vzor se skládá ze čtyř základních elementů:

- Jméno vzoru,
- popis problému, který řeší,
- popis řešení,
- důsledky aplikace vzoru.

Jméno vzoru se většinou skládá z jednoho či dvou slov a popisuje problém návrhu nebo jeho řešení. Díky jménu je možné vzory uložit v katalogu a také podle něj vyhledávat. Pojmenování vzoru poskytuje určitou úroveň abstrakce, díky které je možné se na vzory jednoduše odkazovat, např.: při komunikaci s jinými lidmi nebo v dokumentaci. Jména vzorů se většinou nepřekládají do jiných jazyků, protože zde hrozí riziko nepřesnosti překladu, a tak se používají i v jiných jazycích originální názvy (v angličtině). Příkladem názvu vzoru je třeba Singleton.

Druhou základní části návrhového vzoru je popis problému. Popis je nejčastěji formou vysvětlení problému, jeho kontextu situace, kdy je vhodné daný vzor aplikovat. Popisem problému může být specifický problém návrhu, např. jak reprezentovat algoritmus jako skupinu objektů, anebo problémem může být také i seznam podmínek, které musí být splněny, aby mělo smysl vzor aplikovat. Aplikace vzoru na nesprávný problém může mít fatální důsledky na aplikaci (ať na její stabilitu, srozumitelnost či pozdější rozšiřitelnost) či být zcela nemožná. Proto je důležité, aby popis problému byl jasný, srozumitelný a jednoznačný.

Popis řešení je další významnou částí návrhového vzoru. Řešení obvykle specifikuje prvky, ze kterých se návrh skládá, jejich zodpovědnosti, vztahy a spolupráce mezi nimi. Řešení ale není konkrétní pro danou architekturu či programovací jazyk, lze je spíše chápat jako šablonu, která má obecný charakter a nevyužívá specifik daného jazyka.

Posledním základním prvkem návrhového vzoru jsou důsledky. Důsledky můžeme chápat jako následky aplikace vzoru, které mohou mít dopad na flexibilitu, přenositelnost nebo rozšiřitelnost systému. Tak jako každé řešení má své výhody i nevýhody, i aplikace návrhového vzoru má své konkrétní důsledky a je na vývojářích, aby tyto vlastnosti zvážili. Více lze nalézt v [1][2][3][4].

2.1 Cíl návrhových vzorů

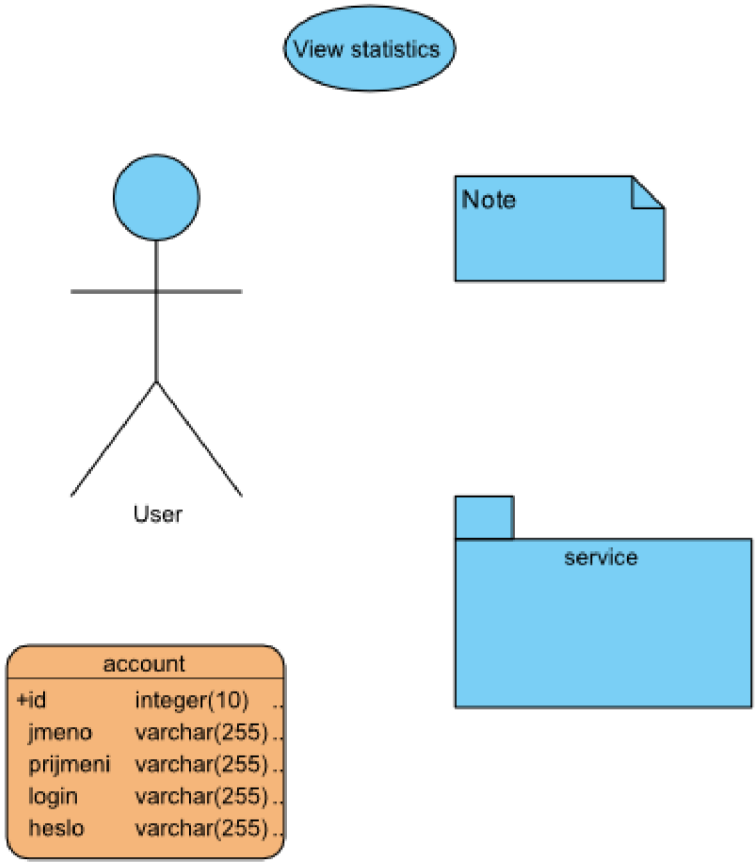
Jak již bylo řečeno, návrhové vzory poskytují osvědčená řešení častých problémů v objektově orientovaném návrhu. Cílem vzorů je poskytnout obecné řešení s minimálními důsledky na flexibilitu, srozumitelnost či rozšiřitelnost systému a naopak co nejvíce podpořit a umožnit znovu použitelnost jednotlivých komponent systému. Je tedy zřejmé, že návrhové vzory úzce souvisí s objektově orientovaným přístupem k modelování.

2.2 Popis návrhových vzorů

Jak bylo popsáno výše, návrhové vzory se skládají ze čtyř hlavních prvků: jména, popisu problému, popisu řešení a důsledků aplikace vzoru. Vzory jsou obvykle organizovány do katalogů. V dnešní době se používají dva hlavní způsoby popisu vzoru: pomocí grafické notace (nejčastěji UML diagramy) a textový popis.

2.2.1 Jazyk UML

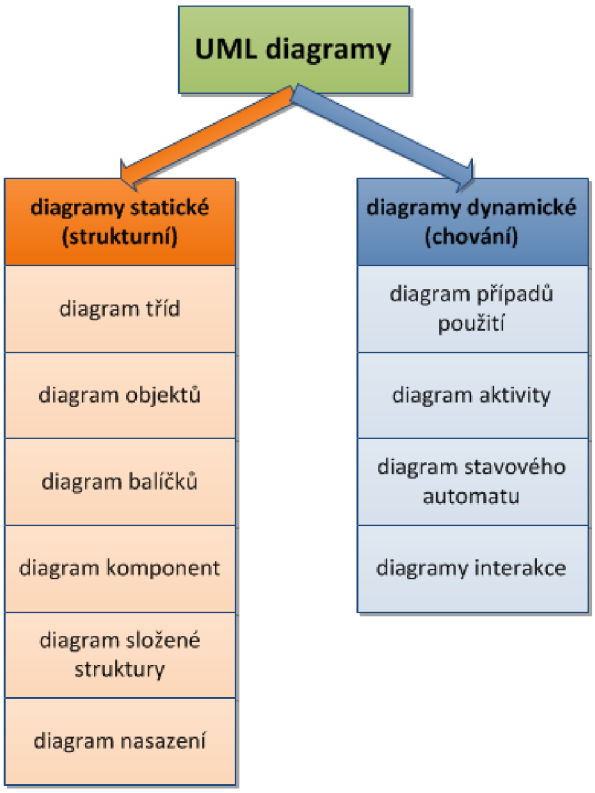
UML (Unified Modeling Language) je jazyk využívaný v softwarovém inženýrství pro vizualizaci, specifikaci, návrh a dokumentaci softwarových systémů. Umožňuje modelovat nejen data, ale i procesy. UML podporuje objektově orientovaný přístup. V současnosti se používá UML verze 2, která oproti verzi předchozí definuje nové prvky a typy diagramů. Jazyk UML lze chápat jako grafický. Popisy mohu být pomocí speciálních symbolů. Více informací lze nalézt v [5][16][17].



Obrázek 1: Příklad UML symbolů.

2.2.2 Typy diagramů

Diagramy jsou nejpoužívanější částí UML. Jazyk UML ve verzi 2 definuje diagramy uvedené níže na obrázku.



Obrázek 2: Přehled diagramů jazyka UML.

2.2.2.1 Diagram tříd

Někdy je také nazýván jako class diagram. Popisuje statickou strukturu systému, zobrazuje systémové třídy, jejich atributy a vztahy mezi nimi. Tento diagram nemusí přesně kopírovat strukturu databáze, některé třídy nejsou ve skutečnosti ani v databázi implementovány (např. abstraktní třídy). Společně s diagramem užití vytváří tzv. konceptuální model.

2.2.2.2 Diagram objektů

Poskytuje úplný nebo částečný pohled na strukturu modelovaného systému v určitém čase. V podstatě se jedná o „časový snímek“ diagramu tříd.

2.2.2.3 Diagram balíčků

Popisuje vnitřní strukturu systému na logické úrovni pomocí logických seskupení (balíčků) a závislostí mezi nimi.

2.2.2.4 Diagram komponent

Popisuje strukturu systému na fyzické úrovni pomocí fyzických částí systému (komponent) a závislostí mezi nimi.

2.2.2.5 Diagram složené struktury

Popisuje vnitřní strukturu třídy a ostatních prvků, které se podílejí na vzniku této struktury.

2.2.2.6 Diagram nasazení

Umožňuje popsat reálné nasazení systému, respektive jeho jednotlivých artefaktů a jejich propojení.

2.2.2.7 Diagram případů použití

Diagram představuje vnější pohled na systém z pohledu uživatele. Umožňuje zobrazit závislosti mezi nejčastěji prováděnými akcemi se systémem (tzv. případy užití).

2.2.2.8 Diagram aktivity

Umožňuje modelovat chování jednotlivých procesů, datových toků apod. Každý proces je v diagramu reprezentován posloupností kroků (akcí), které jsou dále nedělitelné, nebo vnořenými aktivitami.

2.2.2.9 Diagram stavového automatu

Diagram graficky popisuje chování objektu pomocí stavů, aktivit v těchto stavech a přechodů mezi nimi. Objekt může na základě definovaných událostí změnit svůj stav, který je reprezentován hodnotami atributů a vazbami na jiné objekty.

2.2.2.10 Diagramy interakce

Diagramy interakce jsou souhrnným názvem pro sekvenční diagram, diagram komunikace, přehledový diagram interakcí a diagram časování.

Sekvenční diagram zobrazuje, jak spolu jednotlivé objekty komunikují v čase pomocí zasílání zpráv. Každý objekt má svoji tzv. časovou linii (osu), v rámci níž komunikuje s jinými objekty.

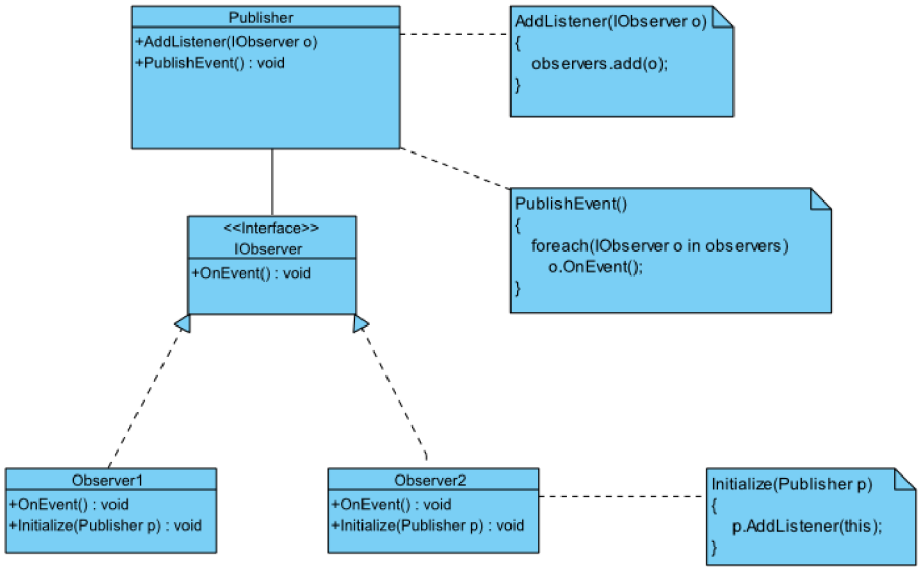
Diagram komunikace také zobrazuje komunikaci mezi objekty pomocí zasílání zpráv, ale na rozdíl od diagramu sekvence je tento diagram zaměřen více na zobrazení statické struktury.

Diagram časování je svým způsobem typem diagramu sekvence. Používá se k modelování chování objektů v reálném čase.

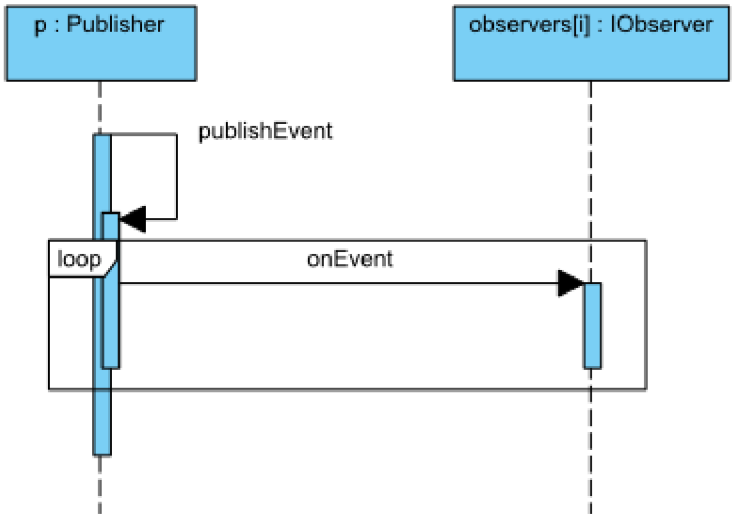
Přehledový diagram interakcí je určitým typem diagramu aktivity, kde každý uzel grafu reprezentuje jiný diagram interakce. Používá se k modelování toku řízení.

2.2.3 Grafický popis návrhových vzorů

Návrhové vzory jsou nejčastěji popsány grafickou notací. Ve většině případů se používá diagram tříd, případně sekvenční diagram. Tyto diagramy obvykle stačí k plnému vyjádření účelu daného vzoru, pro jasnost ale bývají ještě doplněny textovým popisem. Pokud se v návrhovém vzoru, problému, který řeší, nebo řešení vyskytuje nějaká nejasnost, je možné ještě tyto popisy doplnit ukázkovým kódem. Níže je zobrazen příklad grafického popisu pro návrhový vzor Observer. Detailnější informace je možné nalézt v [5].



Obrázek 3: Příklad diagramu tříd pro návrhový vzor Observer.



Obrázek 4: Příklad sekvenčního diagramu pro návrhový vzor Observer.

2.2.4 Textový popis návrhových vzorů

Textový popis návrhových vzorů je neméně důležitou částí. Bývá uváděn často společně s grafickým popisem. Jak již bylo řečeno, vzory jsou obvykle organizovány do katalogu, ve kterém mohou být tříděny a vyhledávány dle různých atributů. Kromě čtyř hlavních elementů vzoru, které byly popsány výše, se u vzorů ještě uvádějí následující atributy:

- Klasifikace,
- motivace,
- struktura,
- účastníci,
- spolupráce,
- implementace,

- vzorový kód,
- související vzory,
- způsob řešení problému.

Vzory jsou někdy klasifikovány podle svého účelu nebo podle oblasti, kde jsou primárně aplikovány. Následující tabulka zobrazuje možné dělení vzorů.

		Účel		
		Tvořivý	*Strukturální*	*Chování*
Oblast	*Třídy*	Factory Method	Adapter (class)	Interpreter Template Method
	Objekty	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Facade Proxy	Chain of Responsibility Command Iterator Observer State Strategy Visitor

Tabulka 1: Příklad dělení návrhových vzorů.

Motivace vzoru popisuje vzorový příklad, ve kterém je specifikován problém a aplikace vzoru se zaměřením na to, jak objekty a třídy řeší daný problém. Cílem je, aby vývojář lépe a rychleji pochopil, kdy je vhodné použít daný vzor.

Struktura vzoru je obvykle znázorněna graficky pomocí diagramů UML. Nejčastěji se používají diagram tříd a diagramy interakce. Účastníky vzoru se rozumí třídy nebo objekty, ze kterých se vzor skládá. Jsou zde také popsány jejich odpovědnosti. Spolupráce popisuje, jak jednotliví účastníci vzoru spolu komunikují, aby realizovali svoji odpovědnost. Implementace vzoru určuje, které techniky a případná specifika pro programovací jazyk bychom si měli uvědomit při implementaci. Vzorovým kódem se většinou rozumí jen krátké části kódu (ne celá implementace vzoru), které se vztahují k záludnějším místům návrhového vzoru. Obvykle bývají doplňkem grafického popisu vzoru. Související vzory pak uvádějí další vzory, které jsou blízké danému vzoru, nebo by bylo vhodné je použít společně s daným vzorem.

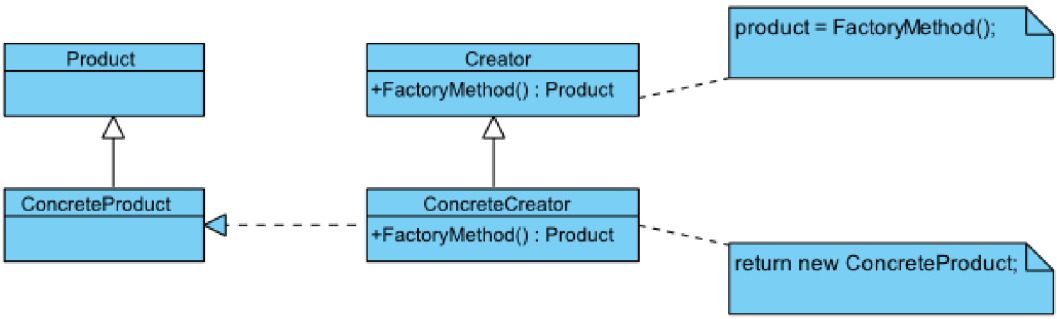
Způsoby, jakými návrhové vzory řeší problém, lze obecně rozdělit do několika kategorií. Jedním ze způsobů je nalezení příslušných objektů. Při objektově orientovaném návrhu se často stane, že obsahuje třídy, které nemají svůj protějšek v reálném světě. Návrhové vzory (např. Composite) umožňuje pracovat s objekty uniformním způsobem a přináší určitou míru abstrakce. Návrhový vzor Strategy naopak umožňuje pracovat s objekty, které se podílejí na realizaci procesu nebo algoritmu, který ale nemusí nutně mít svůj fyzický protějšek.

Dalším způsobem, kterým návrhové vzory řeší problémy, je určení granularity objektů. Objekty se v systému velmi liší v počtu a velikosti. Mohou reprezentovat rozličné elementy, a tak jednou z nejtěžších částí objektově orientovaného návrhu je určení, co bude objekt. Příkladem mohou být vzory Facade, který popisuje, jak reprezentovat část systému jako jeden objekt, nebo vzor Abstract Factory, který určuje, jaký objekt bude zodpovědný za vytváření jiných.

Dalšími možnými způsoby, jak návrhové vzory řeší problémy, je, že definují či jiným způsobem ovlivňují rozhraní objektů, jejich implementaci, snaží se usnadnit pozdější změny v kódu či zavádějí mechanismy do návrhu aplikace, které umožní znovu použití komponent. Více informací je v [5].

2.3 Factory Method

Návrhový vzor Factory Method (tovární metoda), dále již jen Factory, poskytuje způsob, jak vytvářet objekty, ale podtřídy rozhodují o tom, které třídy budou instanciovány. Výhodou je, že metoda eliminuje nutnost vázat se při tvorbě objektů na aplikačně specifické třídy. Tento vzor je výhodné použít, když chceme zachovat flexibilitu nebo existují konkrétní důvody pro výběr třídy, jež bude vyvářet instance, za běhu. Příklad struktury je zobrazen níže.

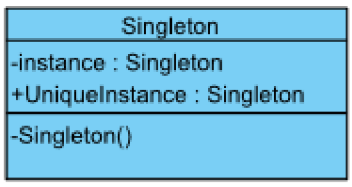


Obrázek 5: Příklad struktury návrhového vzoru Factory Method.

Product definuje rozhraní objektů, které jsou vytvářeny tovární metodou. Creator je abstraktní třída, která obsahuje virtuální metodu *FactoryMethod*. ConcreteProduct jsou konkrétní, specifické produkty, které jsou vytvářeny konkrétním creatorem pomocí tovární metody. *ConcreteCreator* je aplikačně specifický creator, který překrývá původní creator a přepisuje virtuální tovární metodu. Klient, který potřebuje vytvořit produkt, volá virtuální tovární metodu třídy *Creator*. Tato metoda je díky dědičnosti zavolána v aplikačně specifickém creatoru, a tak není nutné vázat Creatora, respektive tovární metodu *FactoryMethod,* na specifika jednotlivých aplikací nebo platforem.

2.4 Singleton

Návrhový vzor Singleton (jedináček) zajišťuje, aby existovala pouze jedna instance třídy a k této třídě se přistupovalo přes jediný globální přístupový bod. Vzor tedy zajišťuje, že instance je vytvořena jen jednou a přistupuje se k ní jednotným způsobem. Výhodou je, že existuje kontrolovaný přístup k instanci, jednoduché rozšíření na více různých instancí a jednotná správa tvorby instancí. Příklad struktury vzoru je zobrazen níže.

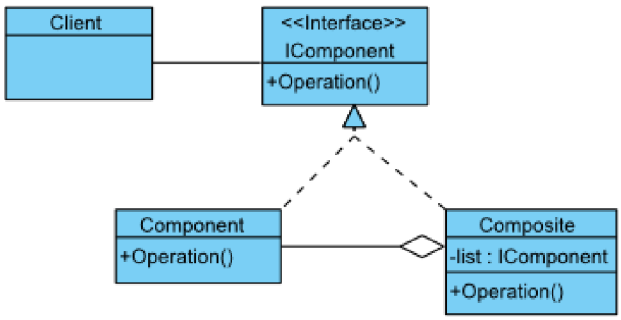


Obrázek 6: Příklad struktury návrhového vzoru Singleton.

Třída *Singleton* obsahuje privátní konstruktor, který vytváří požadované instance. Dále obsahuje privátní statický objekt *instance*, který je vytvořený pomocí privátního konstruktoru. Samotná instance je pak zpřístupněna pomocí veřejné statické vlastnosti *UniqueInstance* (tzv. globální přístupový bod).

2.5 Composite

Návrhový vzor Composite (složenina) umožňuje jednotně zacházet s jednotlivými objekty i skladbami objektů. Lze tak modelovat hierarchii typu část - celek. Klienti pak zacházejí s objekty ve složené struktuře jednotným způsobem. Příklad struktury vzoru je zobrazen na následujícím obrázku.

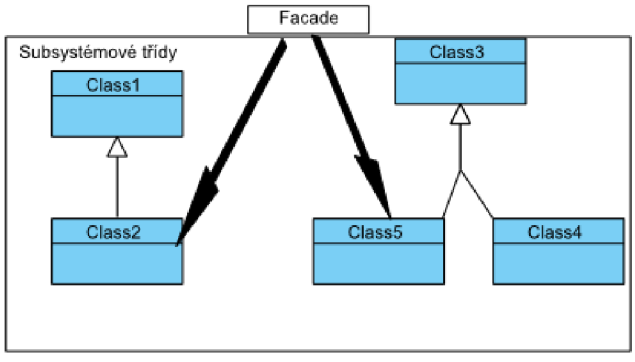


Obrázek 7: Příklad struktury návrhového vzoru Composite.

Vzor se skládá ze dvou hlavních typů objektů: *Component* a *Composite*. Oba typy realizují společné rozhraní *IComponent*, čímž je zajištěno, že klient může pracovat jak s jednotlivými objekty, tak i se složeninami objektů uniformním způsobem. Objekty typu *Composite* se skládají z objektů typu *Component*, přičemž operace volané na objektu *Composite* jsou většinou implementovány tak, že se zavolají ekvivalentní operace na objektech *Component*, ze kterých se *Composite* skládá.

2.6 Facade

Návrhový vzor Facade (fasáda) poskytuje vysokoúrovňový pohled na subsystémy, jejichž detaily jsou schovány, a to přes jednotné rozhraní, které je definováno na vyšší úrovni a usnadňuje použití subsystému. Fasáda v podstatě představuje vstupní bod do subsystému a významně snižuje závislost mezi klienty a třídami subsystému. Níže je zobrazena struktura návrhového vzoru Facade.



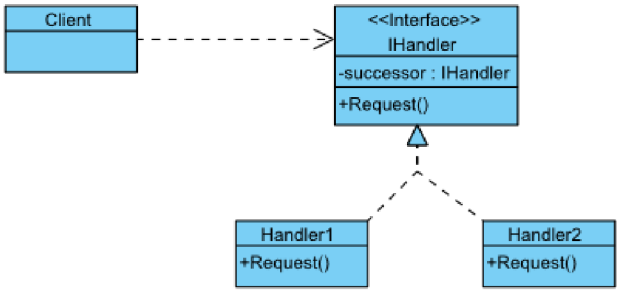
Obrázek 8: Struktura návrhového vzoru Facade.

Facade deleguje jednotlivé žádosti klientů subsystémovým třídám, protože ví, která třída je za co odpovědná. Subsystémové třídy pak implementují funkce, které jsou poskytovány subsystémem. Vykonávají práci, kterou jim přidělí *Facade*, a o fasádě nemají ponětí. Výhodou tohoto vzoru je, že

umožňuje redukovat vnitřní složitost systému, zajišťuje snazší dekompozici systému do vrstev a podporuje nezávislost a přenositelnost subsystému jako takového.

2.7 Chain of Responsibility

Návrhový vzor Chain of Responsibility (řetěz odpovědnosti) zabraňuje spojení odesílatele zprávy a příjemce tím, že umožní, aby byla žádost zpracována více než jedním objektem. Toho docílí tak, že objekty příjemců zřetězí (každý objekt obsahuje odkaz na následníka). Objekty příjemců jsou schopny zpracovat jen určité požadavky. Odesílatel pošle požadavek prvnímu příjemci a ten, pokud nemůže požadavek zpracovat, ho přepošle následníkovi v řetězci. Na konci řetězce obvykle bývá definována výchozí obsluha požadavku. Následující obrázek ukazuje strukturu návrhového vzoru Chain of Responsibility.

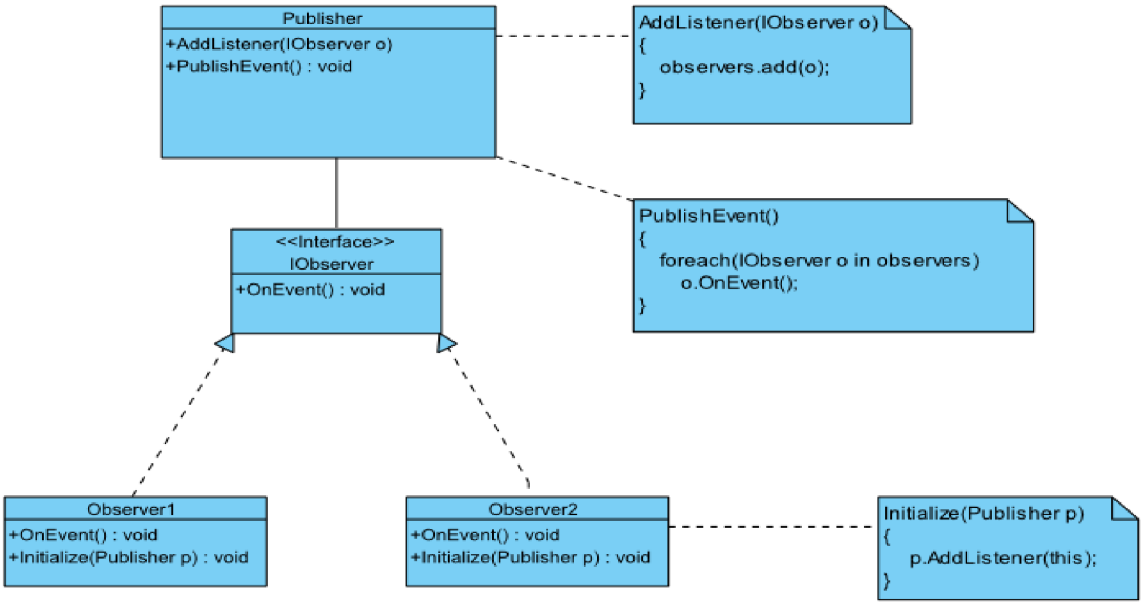


Obrázek 9: Příklad struktury návrhového vzoru Chain of Responsibility.

Client je třída, která zasílá požadavek voláním metody *Request*. *IHandler* je společné rozhraní pro jednotlivé objekty zpracovávající požadavky. Tento vzor je vhodné použít, pokud existuje více různých objektů, které mohou zpracovat žádost, a zpracovatel není předem znám, nebo chceme vystavit žádost více příjemcům, aniž bychom určili, kdo ji obslouží.

2.8 Observer

Návrhový vzor Observer (pozorovatel), někdy také označován jako Publisher-Subscriber, definuje závislost typu *1:n* mezi objekty tím způsobem, že změní-li jeden objekt svůj stav, ostatní objekty jsou o této změně informovány. Obvykle platí, že existuje jen jeden pozorovaný objekt, který mění svůj stav, a více pozorovatelů, kteří si přejí být informováni o změně. Typické použití vzoru je v situacích, kdy chceme, aby objekt informoval jiné objekty, aniž by mezi nimi bylo „těsné“ spojení, např. objekty aplikační logiky při změně stavu informují objekty GUI. Níže je zobrazena struktura návrhového vzoru Observer.

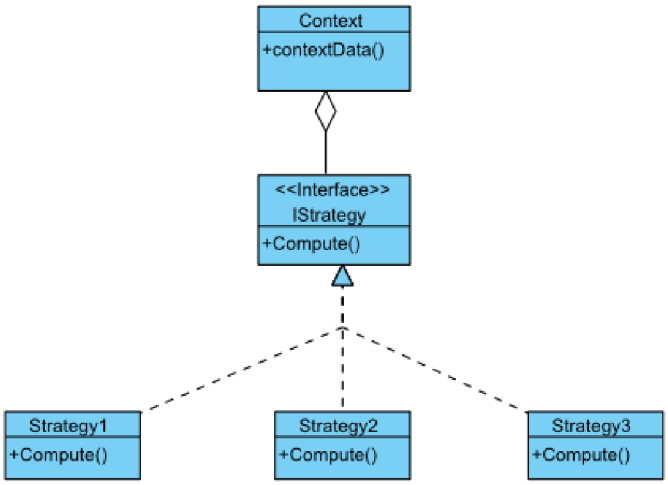


Obrázek 10: Příklad struktury návrhového vzoru Observer.

Subject zná všechny objekty, které chtějí být informovány o jeho stavu. Poskytuje rozhraní pro zaregistrování nebo odregistrování objektu (metody *addListener* a *removeListener*). Při změně stavu informuje všechny objekty přes rozhraní *IObserver*, které tak definuje, jakým způsobem mají být o změnách informovány (metoda *publishEvent*). Třídy implementující toto rozhraní pak obsahují referenci na objekt *Subject* a u něj se případně registrují pro informování o změně jeho stavu. Tento vzor je vhodné použít, když změna jednoho objektu vyžaduje i změnu objektů jiných, ale jejich počet dopředu neznáme, nebo když chceme, aby objekt informoval jiné, aniž by byl s nimi těsně „vázán“. Proto je tento vzor nejčastěji užit pro informování objektů GUI o změně objektu v aplikační logice.

2.9 Strategy

Návrhový vzor Strategy (strategie) umožňuje definovat řadu algoritmů, které spolu nějak souvisí (např. řeší stejný problém různými způsoby), tak, že jsou odděleny od hostitelské třídy (klienta) a jsou vloženy do samostatné třídy. Důsledkem je, že kód v klientovi je čistší a přehlednější. Vzor umožňuje klientovi vybrat příslušný algoritmus na základě kontextu takovým způsobem, že třídy implementující algoritmy mohou být vyjádřeny nezávisle na datech, která používají. Struktura návrhového vzoru Strategy je zobrazena na následujícím obrázku.



Obrázek 11: Příklad struktury návrhového vzoru Strategy.

Klient komunikuje výhradně přes objekt *Context*, kterému předá data. Výběr vhodného algoritmu pak provádí buď klient (a tento výběr specifikuje při komunikaci s *Context*), nebo výběr provede *Context* sám. V případě druhé možnosti *Context* vybere vhodný algoritmus na základě dat, která obdržel od klienta. *IStrategy* je rozhraní, které je společné pro všechny algoritmy, a *Context* ho využívá při volání algoritmu. Třídy implementující toto rozhraní pak reprezentují jednotlivé algoritmy. V důsledku klient komunikuje s algoritmy pouze prostřednictvím třídy *Context*. Tento vzor je vhodné použít, když existuje skupina příbuzných tříd, které se odlišují jen chováním, nebo když existuje skupina algoritmů, které řeší stejný problém různým způsobem. Výhodou je, že klient nezná datové struktury algoritmů a jeho kód je tak přehlednější (není nutné programovat algoritmy přímo do třídy klienta). Další návrhové vzory a jejich detailní popisy je možné nalézt v [1].

3 Specifikace návrhových vzorů

Návrhové vzory poskytují řešení typických problémů v objektově orientovaném přístupu k návrhu aplikací. Vývojáři jejich užitím zvyšují flexibilitu a znovupoužitelnost systému. Vzory tak významně usnadňují návrh aplikace a jsou úzce vázány se softwarovým inženýrstvím. Vzory jsou většinou organizovány do katalogu, ve kterém se často využívá kombinace textového a grafického popisu vzoru (více viz kapitola Popis návrhových vzorů). Z formálního hlediska lze popis (specifikaci) návrhových vzorů rozdělit na formální a neformální.

3.1 Neformální specifikace návrhových vzorů

Za neformální specifikace návrhových vzorů lze označit v současnosti používané popisy v katalozích. Tyto specifikace definují jak strukturu vzoru, tak i chování. Jejich výhodou je, že jsou jednoduché na pochopení a na první pohled srozumitelné. Stejně tak jsou vhodné k použití při běžné komunikaci mezi vývojáři. Zásadní nevýhodou je, že neformální specifikace mohou být nejednoznačné, dvojsmyslné a důsledkem je špatné pochopení ze strany vývojářů, což může mít fatální následky při vývoji systému. Návrhové vzory často mají různé specifické atributy a vlastnosti, které ho jednoznačně charakterizují, a při použití vzoru je nutné tato specifika dodržet. Neformální popis ale neumožňuje tyto aspekty automatizovaně kontrolovat. Díky těmto vlastnostem je neformální specifikace nevhodná pro automatizované generování kódů.

3.2 Formální specifikace návrhových vzorů

Současný výzkum ve specifikacích návrhových vzorů se zaměřuje na formální popis vzorů. Cílem formální specifikace návrhových vzorů je odstranit nevýhody neformálních popisů, zajistit jejich precizní a jednoznačný popis a umožnit jejich automatizovanou verifikaci. Formální specifikace nenahrazují neformální, pouze je doplňují, pomáhají zlepšit pochopení sémantiky, mohou pomoci vývojářům při rozhodování, který vzor použít. Více informací o formálních specifikacích návrhových vzorů je v [8][10][11].

V současnosti se možnosti formální specifikace návrhových vzorů dělí do dvou kategorií. Do první kategorie patří speciálně vyvinuté formální jazyky nebo polo-formální grafické modelovací jazyky. Tyto jazyky jsou zcela nové a vyvinuté konkrétně pro potřeby formálních zápisů vzorů. V případě grafických modelovacích jazyků se většinou jedná o formální doplnění jazyka UML. Do druhé kategorie pak patří způsoby, které využívají již existující formální systémy, jež upravují a vhodně kombinují tak, aby je bylo možné použít pro formální specifikace návrhových vzorů.

Hlavní výhodou první kategorie je, že je jednoduché je použít, protože tyto formální jazyky jsou přímo koncipovány pro formální specifikace vzorů. Není je tedy nutné jakkoliv upravovat. Nevýhodou je, že nejsou příliš rozšířené, a tudíž i podpora např. nástrojů pro verifikaci vzoru je velmi nízká. Do této kategorie lze zařadit např. GEBNF (Graphical Extended Bacus Normal Form).

Výhodou druhé kategorie je, že existující formální systémy většina vývojářů zná. Tyto systémy již existují dlouho a díky tomu je i podpora nástrojů pro automatizovanou verifikaci na vysoké úrovni. Zásadní nevýhodou ale je, že neumožňuje úplnou definici vzoru, tj. jak definici chování, tak i struktury. Důvodem je, že tyto formální systémy nebyly primárně koncipovány pro popis vzorů, ale nyní jsou na tento problém adaptovány. Toto se nejčastěji řeší kombinací dvou formálních systémů:

predikátové logiky 1. řádu, pomocí které se definuje struktura vzoru, a temporální logiky akce, která definuje chování.

3.2.1 Predikátová logika 1. řádu

Za predikátovou logiku je v matematice a logice většinou označován formální odvozovací systém používaný k popisu matematických teorií. Je rozšířením výrokové logiky. Výroková logika zkoumá způsob tvorby složených výroků z výroků jednoduchých a závislost pravdivosti složeného výroku na pravdivosti výroků, z nichž je složen. Jestliže P je neprázdná množina symbolů, které nazýváme prvotní formule. Složené výroky se pak vytvářejí pomocí logických spojek negace, konjunkce, disjunkce, implikace a ekvivalence. Jazyk výrokové logiky L nad množinou P značí se L_P, jsou prvky množiny P, logické spojky a závorky (,). Jednotlivé výrokové formule jazyka jsou pak definovány následovně:

1. Každá prvotní formule $p \in P$ je výrokovou formulí.
2. Jsou-li A, B výrokové formule, pak i $(\neg A), (A\&B), (AVB), (A \rightarrow B), (A \equiv B)$ jsou výrokové formule.
3. Každá výroková formule vznikne konečným počtem použití předchozích pravidel.

Formální axiomatický systém výrokové logiky je tedy tvořen abecedou (množina P), formulemi, jazykem, axiomy (výroky, jejichž platnost se nedokazuje, ale předpokládá) a odvozovacím pravidlem *modus ponens*.

Predikátová logika je pak rozšířením výrokové logiky. Kromě prvků definovaných výše predikátová logika zavádí ještě proměnné (označují objekty daného oboru), konstanty (nulární funkční symboly), funkční symboly (definují operace, které je možné s objekty provádět), predikáty (vyjadřují vztahy mezi objekty) a univerzální kvantifikátor (vyjadřuje platnost pro všechny prvky z daného oboru) a existenční kvantifikátor (vyjadřuje existenci daného prvku v oboru). Kvantifikátory umožňují kvantifikaci proměnných. U funkčních symbolů a predikátů se ještě uvádí přirozené číslo, které udává jejich četnost, tj. počet argumentů. Pokud je četnost rovna n, říká se, že predikát je n-ární (např. pro n=2 je binární). Jazyk predikátové logiky 1. řádu je pak tvořen logickými symboly (proměnné, logické spojky, kvantifikátory, závorky) a speciálními symboly (funkční symboly a predikáty). V predikátové logice 1. řádu se za term označuje:

1. Každá proměnná.
2. Je-li f funkční symbol četnosti n a jsou-li $t_1 \ldots t_n$ termy, pak také $f(t_1 \ldots t_n)$ je term.
3. Každý term vznikne konečným počtem použití předchozích pravidel.

Jestliže p je predikát o četnosti n a $t_1 \ldots t_n$ jsou termy, pak i $p(t_1 \ldots t_n)$ je atomická formule. Formule, které tvoří jazyk predikátové logiky 1. řádu, jsou pak tvořeny následujícím způsobem:

1. Každá atomická formule je formule.
2. Jsou-li a, b formule, pak i $(\neg a), (a\&b), (avb), (a \rightarrow b), (a \equiv b)$ jsou formule.
3. Je-li x proměnná a b formule, pak i $(\forall xb), (\exists xb)$ jsou formule.
4. Každá formule vznikne konečným počtem použití předchozích pravidel.

Predikátová logika 1. řádu se používá ke specifikaci struktury návrhových vzorů. Nejčastěji se pomocí ní specifikují vztahy mezi účastníky vzoru pomocí predikátů, dále se často využívají proměnné, které reprezentují třídy, jejich atributy a metody. Další informace o predikátové logice a její použití při specifikace vzorů je možné nalézt v [7][9][18][20].

3.2.2 Temporální logika akce

Temporální logika akce je logika vyvinutá pro popis chování systémů. Je založena na temporální logice a logice akce. Temporální logika je formální systém, který umožňuje popis systémů, ve kterém

jsou jeho pravidla a tvrzení podmíněny v rámci časových intervalů. Temporální logiku můžeme chápat jako rozšíření predikátové logiky o notaci umožňující specifikaci časově podmíněných tvrzení (např. „Stále mám hlad“), které není možné vyjádřit pomocí prvků predikátové logiky. Čas je definován diskrétně a pokračuje do nekonečna (lze ho chápat jako nekonečnou množinu diskrétních okamžiků).

Temporální logika definuje tři hlavní unární operátory, které jsou nejčastěji reprezentovány grafickou notací takto: □, ○, ◊. Operátor □ (někdy označován jako *G*) vyjadřuje, že od daného okamžiku je tvrzení platné (nabývá logické hodnoty *true*), ○ (někdy označován jako *N*) vyjadřuje, že tvrzení je platné (nabývá logické hodnoty *true*) od následujícího časového okamžiku, ◊ (někdy označován jako *F*) vyjadřuje, že tvrzení někdy v budoucnosti bude platné (nabude logické hodnoty *true*). Dále definuje dva binární operátory: *U* a *P*, které jsou definovány následovně: *aUb* znamená, že tvrzení a je platné, dokud je platné tvrzení b, *aPb* znamená, že tvrzení a předchází tvrzení b (jeho platnost nastane v dřívější diskrétní časový okamžik). Temporální logiku lze dále rozdělit na dvě hlavní oblasti: lineární a větvící se. V lineární temporální logice se používá lineární čas, který se z pohledu budoucnosti vyskytuje jen v jediné instanci (existuje jen jedna budoucnost). V rozdělující se logice dochází k dělení času do jednotlivých částí (tzv. větví), tj. čas se z pohledu budoucnosti dělí do více instancí (existuje více budoucností).

Temporální logika akce staví z větší části na temporální logice. Mimo jiné se ale také skládá z logiky akce. Umožňuje popsat chování systémů jako sekvencí stavů, ve kterých se systém nachází, a akcí, které způsobují změnu stavu. Stav je jednoznačně charakterizován hodnotami atributů objektů, ze kterých se systém skládá. Akce jsou atomické, tj. nemůže být přerušena jinou akcí.

Tvrzení v temporální logice akce je možné zapsat ve tvaru $[A]_x$, kde *A* je akce a *x* je množina proměnných, které se v akci vyskytují. *A* pak může být zapsána např. výrazem: $a + a' * b = b'$. Proměnné v tomto výrazu jsou značeny jako *a*, *b*, znamenají hodnotu proměnné v aktuálním stavu. Jejich ekvivalenty s apostrofem pak znamenají hodnotu proměnné v následujícím stavu.

Temporální logika akce se nejčastěji používá k formální definici chování návrhových vzorů. Pomocí ní lze definovat, jak spolu účastníci vzoru kooperují, nebo i například asociace objektů přes dočasné vztahy. Více informací o temporální logice akce je v [9][19].

3.2.3 SPINE

SPINE je formální jazyk založený na programovacím jazyku Prolog, který umožňuje formální specifikaci návrhového vzoru pomocí vestavěných funkcí a predikátů. Jazyk podporuje jednoduché existenční kvantifikátory a umožňuje iterace přes struktury tříd (metody, rozhraní apod.) a implementace metod. Díky tomu umožňuje specifikaci struktury i chování vzoru.

Termy v jazyce SPINE jsou proměnné, seznamy nebo složené termy (predikáty). Pravidla definující složené termy jsou odděleny čárkami a jejich vyhodnocování probíhá zleva doprava. SPINE definuje speciální kvantifikátory *forAll* a *exists*, které operují nad seznamem termů a jsou ekvivalentní logické operaci *AND* respektive *OR*. Níže je uveden příklad specifikace vzoru Singleton v jazyce SPINE.

```
realises('PublicSingleton', [C]) :-
  exists(constructorsOf(C), true),
  forAll(constructorsOf(C), Cn.isPrivate(Cn)),
  exists(fieldsOf(C), F.and([
    isStatic(F), isPublic(F), isFinal(F), typeOf(F, C), nonNull(F)
                              ])
)
```

Kód 1: Příklad specifikace vzoru Singleton v jazyce SPINE.

Predikát *realises* specifikuje definici vzoru, (v tomto případě *PublicSingleton* bude porovnán vůči jakékoliv třídě *C*). Predikáty *constructorsOf* a *fieldsOf* vrací seznam konstruktorů dané třídy, respektive její atributy. Predikáty *exists* a *forAll* poté tyto seznamy zpracovávají. Výše uvedená definice tedy znamená, že libovolná třída *C* je Singleton, pokud:

- obsahuje alespoň jeden konstruktor,
- všechny konstruktory této třídy jsou privátní,
- všechny atributy třídy jsou statické, veřejné, *final*, nejsou typu *null* a jsou typu, jako je sama třída.

Tomuto formálnímu zápisu vzoru odpovídá následující kód v programovacím jazyce Java.

```
public class PublicSingleton {
      public static final PublicSingleton instance = new
PublicSingleton();
}
```

Kód 2: Příklad návrhového vzoru Singleton v jazyce Java.

Je tedy zřejmé, že díky formálnímu zápisu návrhového vzoru je pak jednoduché tento vzor automatizovaně verifikovat. Jazyk SPINE se nejčastěji používá k formálním zápisům vzorů pro programovací jazyk Java a následně je verifikován pomocí nástroje HEDGEHOG. Detailnější informace o jazyku SPINE je možné nalézt v [10].

3.2.4 GEBNF

EBNF (Extended Backus Naur Form) je notace, pomocí které se formálně popisuje syntax programovacích jazyků. Syntax je popsána pomocí pravidel ve tvaru: *<číslo se znaménkem> ::= [znaménko] , <číslo>;* . Pravá strana pravidla určuje prvek, pro který se definuje syntax, levá strana pak obsahuje samotnou syntax. Výše uvedené pravidlo tedy znamená, že číslo se znaménkem se skládá z čísla a případného znaménka (ve většině programovacích jazyků se pro kladná čísla znaménko neuvádí). Rozšířená Bacus-Naurova forma (EBNF) zavádí právě do definice pravidel volitelnost výskytu prvků, která se značí pomocí hranatých závorek (na rozdíl od Bacus-Naurovi formy, která toto neumožňuje).

GEBNF je pak grafické varianta EBNF. Je to tedy notace, která umožňuje popsat syntax grafických modelovacích jazyků. Ve smyslu formální specifikace návrhových vzorů se používá jako doplněk modelovacího jazyka UML.

GEBNF definuje modelovací jazyk jako n-tici *(R,N,T,S)*, kde *N* je konečná množina neterminálních symbolů, *T* je konečná množina terminálních symbolů, *R∈N*, označován jako počáteční symbol a *S* je množina konečná množina pravidel zapsaných ve tvaru: *Y ::= Exp* , kde

Y ∈*N* a *Exp* je ve tvaru: L_1:X1L2:X2...Ln:Xn nebo také $X_1|X_2|...|X_n$, $L_1...L_n$ označují názvy položek a $X_1...X_n$ jsou položky, které mohou být ve tvaru *Y*, Y^*, Y^+, *[Y]*, $\underline{Y}$. Význam této notace je uveden v následující tabulce.

Notace	Význam	Příklad užití
$X_1\|...\|X_n$	výběr z X_1, X_2 ... X_n	ActorNode | UseCaseNode znamená, že entita je aktivním účastníkem události nebo případem užití
$L_1 : X_1$ $L_2 : X_2$: $L_n : X_n$	Uspořádaná posloupnost položek typu X_1, X_2, ... X_n , které jsou zpřístupněny přes jména L_1, L_2, ... L_n	ClassName : Text, Attributes : Attribute*, Methods : Method* znamená, že se entita skládá ze tří částí nazvaných ClassName, Attributes a Methods
X^*	X^i, i >= 0	Diagrams* znamená, že se entita skládá z n diagramů, kde n >= 0
X^+	X^i, i >= 1	Diagrams+ znamená, že se entita skládá z n diagramů, kde n >= 1
[X]	X je volitelné	[User]: prvek user je volitelný
$\underline{X}$	Odkaz na existující prvek typu X	ClassNode: je odkaz na existující prvek typu třídy

Tabulka 2: Příklad významu notace v GEBNF.

Díky GEBNF je pak možné formálně definovat diagramy tříd, sekvence apod., díky tomu se pak společně s jazykem UML stává vhodným prostředkem pro formální popis návrhových vzorů. Příklad uvedený níže ukazuje možnou definici diagramu tříd v GEBNF.

```
ClassDiagram::=classes:Class+,assocs:Rel*,inherits:Rel*,compaq:Rel*
Class::=name:String,[attrs:Property*],[oper:Operation*]
Operation::=name:String,[params:Parameter*],[isAbstract:Bool],[isStatic:Bo
ol]
Parameter::=[name:String],[type:Type],[direction:ParDir]
ParDir::="in"|"out"|"return"
Property::=name:String,type:Type,[isStatic:Bool]
Rel::=[name:String],source:End,destination:End
End::=node:Class,[name:String]
```

Kód 3: Příklad formální definice diagramu tříd v GEBNF.

Výše uvedený zápis ve formátu GEBNF definuje, že diagram tříd se skládá z několika tříd (minimálně však jedné), mezi kterými existují vztahy asociace, dědičnosti a relace generalizace/specializace (*compaq*). Třída pak obsahuje povinně název, volitelně atributy a operace (metody). Operace se povinně skládá z názvu a volitelných parametrů, dále je definováno, zda je metoda statická a abstraktní. Parametr metody se skládá z názvu, typu a směru. Vlastnost (typ atributu třídy) je určena názvem, typem a příznakem, zda se jedná o atribut třídy nebo instance třídy. Relace je definována mezi dvěma koncovými body a je pojmenovaná. Koncové body relace jsou třídy.

Obdobným způsobem lze formálně definovat i ostatní diagramy UML, které lze použít k popisu návrhového vzoru. Nejčastěji se ještě používá diagram sekvence k definování chování vzoru.

3.2.5 Formální specifikace pomocí Prologu

Další možností, jak formálně popsat návrhové vzory, je programovací jazyk Prolog. Návrhové vzory jsou reprezentovány pomocí pravidel Prologu a jsou uloženy v jeho databázi. Výhodou tohoto přístupu je, že návrhové vzory mohou být znovu použity pouhým instanciováním patřičných pravidel. Typické vlastnosti a omezení vzoru jsou také zapsány pomocí pravidel a díky tomu je možné je jednoduše verifikovat. Přidávání nebo odebírání jednotlivých prvků struktury vzoru je realizováno pomocí *assert* a *retract* klauzulí, a převod takového zápisu vzoru do kódu programovacího jazyka je velmi jednoduchý, na to existuje hodně nástrojů (např. DRACO-PUC).

Návrhové vzory jsou v prologu zapsány jako termy reprezentující základní prvky objektově orientovaného návrhu (tj. třídy, rozhraní, metody apod.) ve formě predikátů. Příloha 2 popisuje polymorfismus v objektově orientovaném přístupu pomocí pravidel jazyka Prolog. Pravidlo *polymorph* reprezentuje strukturu polymorfismu. Argumenty tohoto predikátu jsou rozhraní, třídy, reference, metody, atributy apod. *Interface* a *Imp* jsou abstraktní třídy. *ImpOperation* je konkrétní metoda, kterou mají implementovat třídy účastnící se polymorfismu, a *Operation* je metoda rozhraní. V predikátu *polymorph* nejdříve uložíme do databáze fakt, že *Imp* je abstraktní třída (pomocí predikátu *abstractClass*) a že *ImpOperation* je metoda třídy *Imp* a její přístupový modifikátor je *public* (pomocí vložení predikátu *method* do databáze). Následně pomocí predikátu *forAll*, který reprezentuje obecný kvantifikátor v predikátové logice, pro každou konkrétní třídu, která implementuje dané rozhraní, do databáze uložíme informaci, že daná třída dědí z třídy *Imp* (pomocí predikátu *inherit*), že se jedná o třídu (pomocí predikátu *class*) a že obsahuje metodu *ImpOperation*, která je *public* (pomocí predikátu *method*).

Změna účastníků polymorfismu (tj. tříd, které implementují nějakou společnou metodu, ale každá jiným způsobem) je realizována pomocí predikátů *extend_polymorph* pro přidání tříd a *retract_polymorph* pro odebrání. Jejich definice je pak velmi podobná definici predikátu *polymorph*, tj. opět se projde seznam všech tříd implementujících rozhraní a společnou metodu a pomocí predikátů *assert* nebo *retract* se příslušně změní databáze Prologu.

Výše zmiňovaný polymorfismus pak může být použit při definic vzorů Bridge, State nebo Strategy. Vzor Bridge je vhodné použít, když existuje více implementací jedné abstrakce. Cílem vzoru je zachovat nezávislost mezi implementací a abstrakcí. Tento vzor je vhodné použít například tehdy, když byla vydána nová verze komponenty systému, ale vývojáři chtějí zachovat i starší verze bez nutnosti modifikovat klienta, který tyto verze používá. Příklad možné formální definice vzoru Bridge je uveden v příloze 2.

Pravidlo *bridge*, které popisuje návrhový vzor Bridge, využívá ve své definici již výše definovaný predikát *polymorph*. Vzor Bridge lze tedy charakterizovat tak, že nejprve vytvoříme polymorfismus mezi třídami *Abstraction*, která reprezentuje abstrakci, s níž komunikují klienti, *Implementor*, které reprezentuje rozhraní, který imeplementují již konkrétní třídy, *Imp* je abstraktní třída, kterou implementuje *Implementor*, *ConcreteImplementorSet*, který reprezentuje seznam tříd, která implementují různá řešeni, *ImpOperation*, což je operace, kterou jednotlivé implementující třídy poskytují, a *Operation*, což je operace, kterou volá klient ve třídě *Abstraction*. Následně se už jen pomocí predikátu *forAll* uloží do databáze všechny abstrakce, které dědí základní třídu *Abstraction*, a stejně tak se o nich uloží informace, že se jedná o konkrétní třídy (ne abstraktní). Případně rozšíření vzoru Bridge o další implementátory je velmi jednoduché, rozšíří se jen polymorfismus. Stejně tak tomu je při odstranění jednoho či více implementátorů. Také je velmi jednoduché a efektivní změnit třídy, které implementují abstrakci.

3.2.6 BPSL

BPSL (Balanced Pattern Specification Language) je formální jazyk, který umožňuje specifikaci návrhových vzorů. Tento formální jazyk se řadí do kategorie formálních systémů, které vhodným způsobem kombinují již existující formální mechanismy a uzpůsobují je tak, aby je bylo možné použít k popisu návrhových vzorů. K definici struktury používá predikátovou logiku 1. řádu (více viz kapitola 3.2.1), k definici chování temporální logiku akce (více viz kapitola 3.2.2).

BPSL využívá ke specifikaci struktury vzoru podmnožinu predikátové logiky 1. řádu (zejména proměnné a predikáty), protože vztahy mezi účastníky vzoru mohou být snadno vyjádřeny jako predikáty. Temporální relace mezi účastníky vzoru a jejich chování umožňuje definovat pomocí podmnožiny temporální logiky akce (používá zejména akce a proměnné popisující stav). BPSL tak vhodným způsobem kombinuje dva formální systémy a díky tomu můžeme jedním jazykem definovat jak chování, tak i strukturu vzoru. Proto je jazyk vyvážený.

BPSL uvažuje jako hlavní stavební prvky, které odrážejí entity a relace:

1. Třídy, atributy, metody, objekty, netypované hodnoty, které nazývá primární entity,
2. trvalé a dočasné relace,
3. akce,
4. každé další prvky musí být odvozeny z trvalých relací nebo primárních entit.

Primární entity jazyk chápe jako základní prvky struktury návrhových vzorů, přičemž uvažuje možnost objektů a netypovaných hodnot vyskytovat se jako parametry metod. Trvalé relace definuje jazyk jako relace, které se po vytvoření již nemohou měnit, zatímco dočasné relace se mohou měnit v průběhu chování vzoru. Akce jazyk popisuje jako atomické jednotky činnosti, kde sestavení více takovýchto jednotek definuje chování vzoru.

BPSL využívá k definici struktury prvky predikátové logiky 1. řádu, zejména pak proměnné, logické spojky, existenční kvantifikátor a predikáty. Proměnné reprezentují primární entity, zatímco predikáty reprezentují trvalé relace mezi nimi. Nechť třídy jsou označeny proměnnou *C*, atributy *A*, metody *M*, objekty *O* a netypované hodnoty *V*, poté následující tabulka popisuje možné trvalé relace mezi účastníky vztahu realizované jako predikáty.

Název	Doména	Význam
Defined-in	M×C	Metoda je definována v konkrétní třídě
	A×C	Atribut je definován v konkrétní třídě
Reference-to-one (many)	C×C	Třída obsahuje referenci na jednu (více) instancí třídy
Inheritance	C×C	První třída dědí z druhé
Creation	M×C	Metoda vytváří instance třídy
	C×C	Jedna z metod první třídy vytváří instance druhé třídy
Invocation	M×M	První metoda volá druhou metodu
Argument	C×M	Argumentem metody je reference na třídu
	V×M	Argumentem metody je hodnota
Instance	O×C	Objekt je instancí dané třídy

Tabulka 3: Příklad definice trvalých relací návrhových vzorů v jazyce BPSL.

U některých návrhových vzorů nestačí pouhá definice jejich struktury, ale je také velmi důležitá definice jejich chování, tj. popis, jak účastníci vzoru spolu spolupracují. BPSL popisuje chování vzoru pomocí podmnožiny temporální logiky akce. Jazyk popisuje chování vzoru jako nekonečnou sekvenci stavů, kterými entity vzoru procházejí. Každý stav lze chápat jako kolekci hodnot stavových proměnných (např. hodnoty atributů tříd) a dočasných vztahů mezi objekty.

Dvojice po sobě jdoucích stavů se nazývá přechod. Systém se na počátku nachází v počátečním stavu a postupně, jak se provádějí jednotlivé akce, prochází jednotlivými stavy. Akce jsou vybírány nedeterministicky a každá má stanovenu vstupní podmínku, která musí být splněna, aby se akce provedla. BPSL navíc oproti definici temporální logiky akce rozšiřuje sémantiku akcí. V BPSL se během vykonávání akcí nejen mění hodnoty stavových proměnných, ale i dočasné vztahy mezi objekty (mohou vznikat nové, zanikat stávající).

Dočasné relace (TR) BPSL definuje jako dvojice *TR(C1[cardinality1],C2[cardinality2])*, kde *TR* je název relace, *C1, C2* jsou třídy a kardinality reprezentují počet instancí daných tříd, které jsou spolu ve vztahu. Kardinalitou může být konkrétní rozsah nezáporných celých čísel (zapsán ve tvaru [m...n]) nebo [*], což znamená jakýkoliv počet. V akcích se pak mohou vyskytovat zápisy ve tvaru např. *TR(o1,o2)*, což znamená, že objekt *o1* je ve vztahu s objektem *o2*, $\neg TR(o1,o2)$, což znamená, že objekt *o1* není již dále ve vztahu s objektem *o2*, *TR(o1,C2)* znamená, že objekt *o1* je ve vztahu se všemi instancemi třídy *C2*.

Akce v BPSL jsou definovány jako seznam vstupních parametrů, vstupní podmínky a těla (jak se má změnit stav systému vykonáním akce). Akce *A* může být např. definována následovně: $A(o1,o2,p)$: $TR(o1,o2) \wedge o1.x \neq p \rightarrow \neg TR(o1,o2) \wedge o1.x'=p$, kde *o1* je objekt třídy C1, *o2* je objekt třídy C2, *p* je vstupní parametr akce a *x* je atribut třídy C1. Je tedy zřejmé, že výraz $TR(o1,o2) \wedge o1.x \neq p$ je vstupní podmínka akce (pokud není splněna, akce se neprovede) a zbytek výrazu je samotné tělo akce, x' znamená hodnotu atributu *x* po provedení akce. Jak již bylo řečeno výše, objekty a hodnoty, které se akce účastní, jsou vybrány nedeterministicky. Proto by se akce *A* provedla pro všechny objekty tříd C1 a C2, které jsou spolu v relaci.

BPSL nepoužívá vzájemně disjunktní množiny proměnných v trvalých a temporálních relacích a akcích. Naopak kombinuje tyto množiny dohromady, tj. některé proměnné, které se vyskytují v definici trvalých relací, se vyskytují i v definici dočasných relací a akcí. BPSL tedy pohlíží na návrhový vzor jako na kolekci entit (tříd, atributů, metod, objektů, hodnot), relací (dočasných a trvalých) a akcí mezi nimi. Z formálního hlediska BPSL definuje návrhový vzor jako model $M = \langle E, R \rangle$, kde *E* je univerzum entit a *R* je množina relací (temporálních/permanentních a akcí). Níže je uveden příklad formální specifikace návrhového vzoru Observer (definice vzoru viz kapitola 2.8).

```
∃ Subject, IObserver, Observer ∈ C;
  subject-state, observer-state ∈ A;
  addListener, removeListener, publishEvent, onEvent ∈ M;
  o, s ∈ O;
  d ∈ V;
```

```
  Defined-in(subject-state, Subject) ∧
  Defined-in(observer-state, Observer) ∧
  Defined-in(addListener, Subject) ∧
  Defined-in(removeListener ,Subject) ∧
  Defined-in(publishEvent, Subject) ∧
  Defined-in(onEvent, Observer) ∧
  Reference-to-one(Observer, Subject) ∧
  Reference-to-many(Subject, IObserver) ∧
  Inheritance(Observer, IObserver) ∧
  Invocation(publishEvent, onEvent) ∧
  Argument(IObserver, addListener) ∧
  Argument(IObserver, removeListener) ∧
  Instance(s, Subject) ∧
  Instance(o, Observer).
```

Attached(Subject[0..1],Observer[*]).
Initially: ¬Attached(s, Observer). addListener(s,o): ¬Attached(s,o) → Attached(s,o) ∨ removeListener(s,o): Attached(s,o) → ¬Attached(s,o) ∨ publishEvent(s,o,d): Attached(s,o) → s.subject-state' = d.

Tabulka 4: Příklad specifikace návrhového vzoru Observer v jazyce BPSL.

Detailnější informace o jazyce BPSL včetně dalších příkladů užití je možné nalézt v [6].

4 Generování kódu

Generování kódu je proces tvorby zdrojového kódu v cílovém programovacím jazyce z jiného popisu. Tuto činnost vykonává komponenta nazývaná generátor, jejímž vstupem je popis kódu a výstupem je cílový kód. Zdrojovým popisem může být například jiný programovací jazyk, formální popis nebo obecná šablona. Hlavním cílem generování kódu je usnadnit a urychlit proces tvorby programů tím způsobem, že programátoři pouze zadají nebo vytvoří vstupní popis cílového kódu a generátor za ně provede samotné vygenerování kódu do cílového jazyka. Díky tomu je generování snazší, neboť programátor nemusí znát detailně (či vůbec) výstupní jazyk (popis se obvykle zadává ve vyšší míře abstrakce, než je výstup). Z hlediska automatizace lze generování kódu rozdělit do dvou skupin: generování kódu manuální a automatizované generování kódu.

4.1 Manuální generování kódu

Manuální generování kódu lze chápat jako proces generování zdrojového kódu, při kterém se na vygenerovaný kód pohlíží jako na řetězec textu (a tak se s ním i zachází), který je ve výsledku zapsán do výsledného souboru. Uživatel výsledný kód přímo sestavuje z jednotlivých řetězců. Jako vstup se nepoužívají žádné obecné popisy. Výsledek tedy není reprezentován žádnými abstraktními uniformovanými strukturami. Níže následuje příklad manuálního způsobu generování kódu v jazyce C#.

```
using System.IO;
using System.Text;
static void Main(string[] args) {
  StringBuilder code = new StringBuilder();
  code.Append("using System;");
  code.Append("public class Person {");
  code.Append("private string firstName; private string lastName;");
  code.Append("public Person(string fName, string lName) {
              this.firstName = fName; this.lastName = lName;}");
  code.Append("}");
  TextWriter tw = new StreamWriter("code.cs");
  tw.WriteLine(code.ToString());
  tw.Close();
}
```

Kód 4: Příklad manuálního generování kódu.

```
using System;
public class Person {
private string firstName; private string lastName;
public Person(string fName, string lName) {
this.firstName = fName; this.lastName = lName;
}
}
```

Kód 5: Výsledek generovaného kódu v jazyce C#.

Tento způsob generování kódu je vhodný pro jednoduché úlohy (typicky s malým počtem řádků vygenerovaného kódu). K jeho hlavním výhodám patří jednoduchost, časová a výpočetní nenáročnost a to, že neklade nároky na uživatele. Hlavními nevýhodami tohoto přístupu ke generování kódu je velká složitost a nepřehlednost při generování rozsáhlých kódů, nízká míra přenositelnosti, vysoká míra závislosti na dané platformě a nemožnost reprezentovat kód v uniformních strukturách.

4.2 Automatizované generování kódu

Automatizované generování kódu lze chápat jako proces generování zdrojového kódu, při kterém je výsledný kód obvykle reprezentován uniformní strukturou. Vygenerovaný kód je nejčastěji reprezentován stromovou strukturou, kde každý uzel stromu je určitého typu, kde daný typ reprezentuje konkrétní entitu kódu (tj. třídy, metody, parametry, proměnné apod. mohou být reprezentovány svým typem). Uživatel pak jen sestavuje strom výsledného kódu pomocí těchto typů. Vstup generátoru je v podstatě abstraktní popis výsledného kódu.

K hlavním výhodám tohoto přístupu patří velká rozšiřitelnost, funkcionalita a mnoho možností, jak specifikovat generovaný kód. Díky tomu, že vstup generátoru je popsán pomocí abstraktních struktur, je dosaženo většího stupně přenositelnosti než v případě manuálního generování kódu. Výhodou je také snadná udržovatelnost kódu a není pro uživatele nutné znát detailně všechna specifika výsledného kódu.

Nevýhodou automatizovaného generování kódu je vyšší složitost zápisu oproti manuálnímu generování, která ovšem u rozsáhlých kódů je spíše výhodou, nutnost detailní znalosti jednotlivých struktur, které reprezentují entity vygenerovaného kódu, a obecně vyšší časová a výpočetní náročnost procesu.

4.2.1 Generování kódu pomocí CodeDom

CodeDom je prostředek v .NET frameworku, který umožňuje automatizované generování zdrojových kódů pro jazyky C#, JScript a Visual Basic. CodeDom (Code Document Object Model) reprezentuje generovaný kód jako model objektů a umožňuje tak generovat zdrojový kód za běhu jiného programu. Objekty modelu jsou určitého typu, který reprezentuje danou entitu zdrojového kódu (např. třídu, metodu, konstruktor apod.), jsou spojeny dohromady pomocí referencí a utvářejí tak dohromady strukturu zvanou CodeDom graf, který modeluje strukturu zdrojového kódu.

Tato část .NET frameworku obsahuje i generátor kódu, jehož vstupem je popis struktury kódu právě v takovémto formátu. Hlavní výhodou této komponenty je, že umožňuje sestavit logickou strukturu zdrojového kódu, která je nezávislá na konkrétním programovacím jazyku. Díky rozšiřitelnosti podpory generátoru kódu o další programovací jazyky je tak možné z jednoho popisu

generovat zdrojový kód v mnoha jiných jazycích. V příloze 3 je uveden příklad objektového modelu zdrojového kódu v CodeDom pro jazyk C#.

Z výše uvedeného příkladu plyne, že CodeDom poskytuje pro každou základní entitu generovaného kódu vlastní třídu. Pro definici třídy se používá třída *CodeTypeDeclaration*, pro definici atributů třídy *CodeMemberField*, pro vytvoření konstruktoru *CodeConstructor* atd. Takto vytvořené instance tříd se spojí dohromady pomocí svých vlastních referencí a utvoří tzv. CodeDom graf, který jednoznačně reprezentuje logickou strukturu výsledného kódu. Třída *CSharpCodeProvider* pak poskytuje metody pro generování kódu do jazyka C#. Až do tohoto řádku v příkladu byl popis generovaného kódu zcela nezávislý na výsledném jazyce. Z výše uvedeného příkladu je tedy zřejmé, že pro generování jednoduchých zdrojových kódů malého rozsahu není CodeDom moc vhodný, neboť zápis je mnohem složitější. Pokud ale nároky na generovaný kód jsou vyšší, stejně tak pokud existují požadavky na automatizované generování kódu a na popis struktury kódu nezávislého na programovacím jazyce, CodeDom je vhodným nástrojem. Další informace o CodeDom je možné nalézt v [14].

```
public class Person {
      private string firstName;
      private string lastName;

      public Person(string fName, string lName) {
            this.firstName = fName;
            this.lastName = lName;
      }
}
```

Kód 6: Výsledek generovaného kódu v jazyce C#.

5 Návrh aplikace

Cílem této práce je vytvoření aplikace, která načte definice vzoru a již existující zdrojové kódy v jazyce C#, v kódu vyhledá vhodné entity (třídy, metody, atributy, rozhraní apod.), které jsou podobné entitám v definici vzoru, seřadí je dle relevantnosti a nabídne je uživateli. Uživatel si pro sebe vybere nejvhodnější prvky a aplikace poté provede refaktorizaci zvolených částí kódu dle vzoru (tj. provede aplikaci vzoru na již existující kód).

5.1 Požadavky na aplikaci

Je zřejmé, že se bude aplikace skládat z několika hlavních částí: generátoru kódu, části zpracovávající popis vzorů, parseru již existujícího kódu a části vyhledávající struktury vzoru v kódu a provádějící jejich vyhodnocování. Z tohoto plynou i hlavní požadavky na aplikaci:

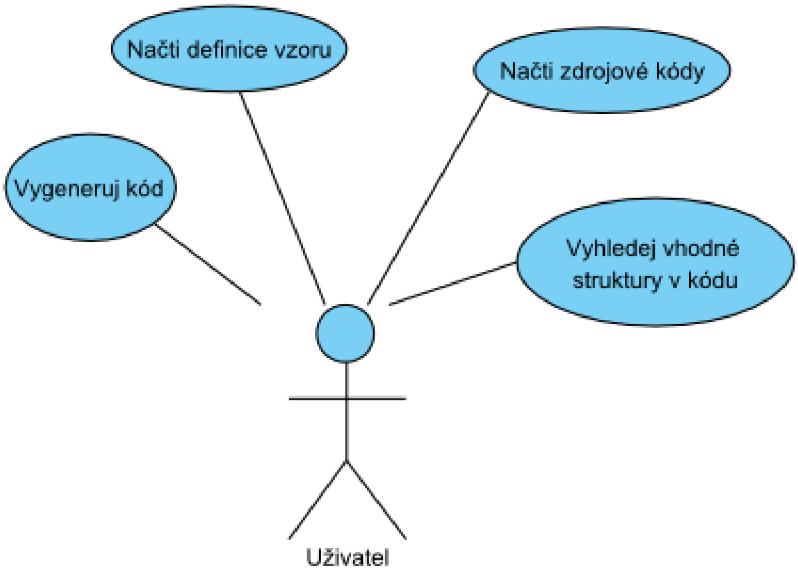
- Podpora definice vzoru v obecné formě nezávislé na platformě,
- validace definice vzorů,
- snadná rozšiřitelnost podpory dalších vzorů,
- reprezentace vzorů uniformní datovou strukturou,
- načítání (parsování) kódů v jazyce C# a jejich reprezentace uniformní datovou strukturou,
- validace načteného kódu,
- vyhledávání podobných struktur vzoru v kódu v abstraktní rovině,
- možnost ohodnotit nalezené entity v kódu relevantností (hodnotou určující vhodnost entity) a jejich porovnání,
- refaktorizace kódu nebo jeho částí dle výběru uživatele,
- přehlednost uživatelského rozhraní (zejména při zobrazení relevantních entit).

Jelikož hlavní náplní aplikace je provádět aplikaci návrhového vzoru na existující kód, je nutné zavést vhodnou reprezentaci vzoru, která by byla platformě nezávislá, ale dostatečně obecná, aby bylo pomocí ní možné vyjádřit co nejvíce vzorů. Zároveň je třeba zajistit, aby takovýto způsob popisu vzorů byl srozumitelný uživatelům. Důležitá je také možnost validace popisu, neboť popisy pro další vzory bude vytvářet uživatel, a tak je nutné zajistit, aby všechny definice vzoru splňovaly určitá omezení. Forma popisu vzoru by také měla být dostatečně uniformní, aby umožňovala přenositelnost mezi aplikacemi.

Dalším významným požadavkem je parsování zdrojových kódů a jejich reprezentace v uniformní struktuře. Jelikož aplikace má provádět refaktorizace kódů jazyka C#, je nutné, aby disponovala kvalitním parserem jazyka C#, který by i umožňoval validaci načteného kódu, neboť nemá smysl aplikovat vzor na kód, který není z lexikálního nebo syntaktického hlediska validní. Důležité je také reprezentovat takto načtený kód pomocí takové struktury, aby bylo možné v ní efektivně a jednoduše vyhledávat podobné struktury vzoru (zejména třídy, rozhraní, metody, atributy apod.)

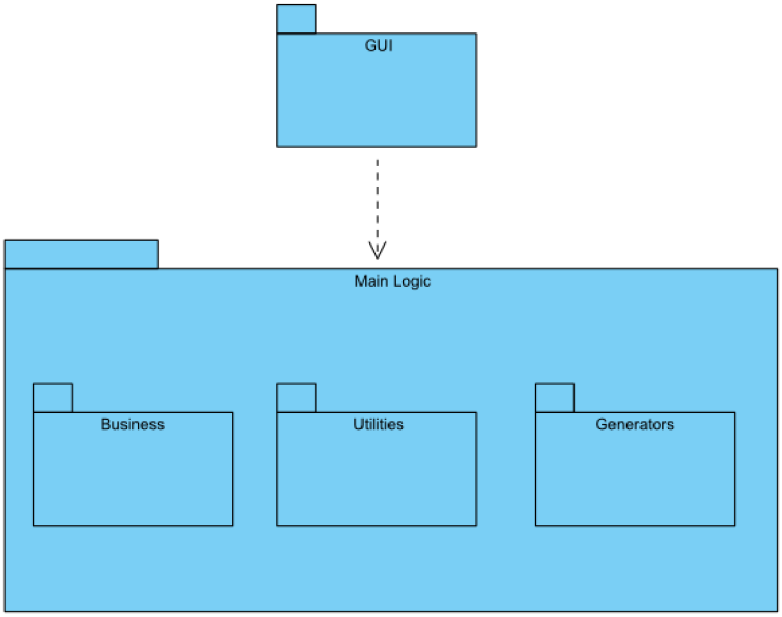
Aplikace by také měla umožňovat ohodnocení nalezených entit v kódu dle vhodnosti (relevantnosti), aby bylo možné uživateli nabídnout nalezené entity od nejvhodnějších po nejméně vhodné, což je velmi efektivní, neboť v případě nalezení velkého množství entit nebude muset uživatel procházet velké množství struktur, sám je vyhodnocovat a hledat tu nejvhodnější.

Z výše uvedených požadavků na aplikaci lze odvodit čtyři hlavní případy užití, které jsou zobrazeny na následujícím diagramu případů užití.



Obrázek 13: Diagram případů užití.

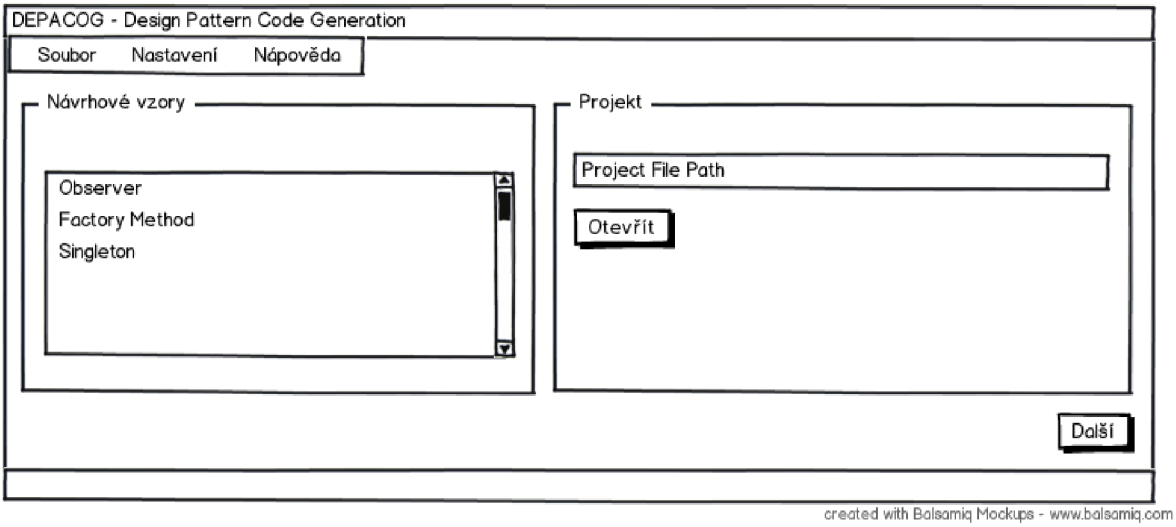
Dle výše uvedených požadavků jsem se rozhodl, že se systém bude skládat ze dvou hlavních částí: prezentační vrstvy a logiky aplikace (business vrstva). Vrstva logiky aplikace bude zodpovědná za provádění systémových operací takovým způsobem, aby realizovala jednotlivé případy použití. Dále bude poskytovat své služby vrstvě nadřazené (prezentační). Dohromady ji tvoří balíčky poskytující obecné služby logiky (Business), doplňkové služby (Utilities) a služby pro generování zdrojových kódů (Generators). Prezentační vrstva bude zobrazovat data uživateli a zachytávat vstupní data od něj, která bude předávat vrstvě nižší (vrstva logiky aplikace). Toto logické dělení aplikace zobrazuje následující diagram balíčků.



Obrázek 14: Diagram balíčků znázorňující logickou architekturu aplikace.

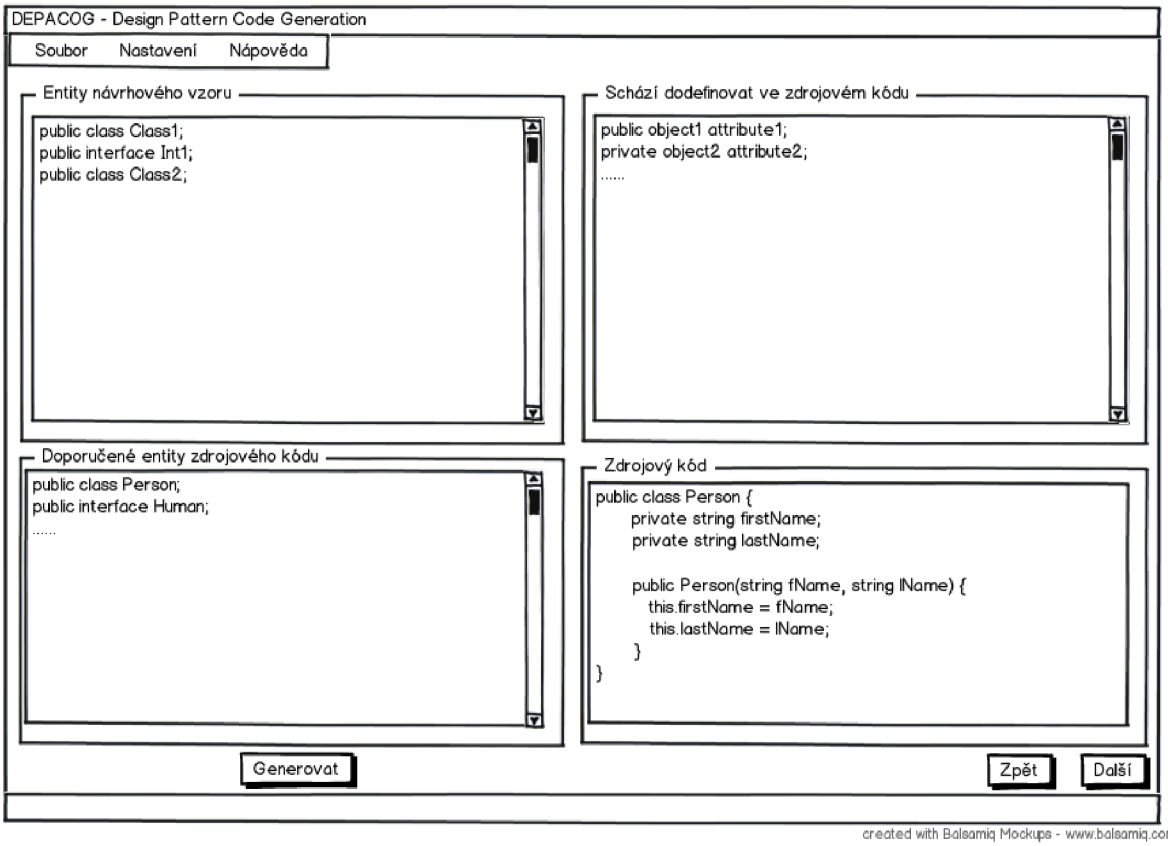
5.2 Prezentační vrstva

Hlavním cílem prezentační vrstvy je interakce s uživatelem, tj. reprezentace dat a příjem vstupu od uživatele. Téměř vždy je tato vrstva realizována jako grafické uživatelské rozhraní. Vzhledem k hlavním požadavkům na aplikaci jsem se rozhodl realizovat prezentační vrstvu jako desktopovou aplikaci. Uživatelské rozhraní je tvořeno dvěma hlavními okny, jejichž návrh je zobrazen na obrázcích níže.



Obrázek 15: Návrh grafického uživatelského rozhraní.

První hlavní okno (viz Obrázek 15: Návrh grafického uživatelského rozhraní.) se zobrazí uživateli po spuštění aplikace. Dělí se na dvě hlavní části: výběr vzoru a výběr projektu. V levé části formuláře jsou zobrazeny dostupné návrhové vzory ve formě seznamu, k jejichž popisům má aplikace přístup a je tak možné je použít pro refaktorizaci. Uživatel může vybrat vždy jen jednu položku. Pravá část formuláře pak slouží k výběru souborů s již existujícím kódem, na který chce uživatel aplikovat vybraný vzor. Po kliknutí na tlačítko *Otevřít* se zobrazí dialogové okno pro výběr souborů. Po kliknutí na tlačítko *Další* aplikace naparsuje kód, validuje ho a pokusí se vyhledat podobné struktury vybraného vzoru v kódu. Následně se zobrazí druhé hlavní okno (viz Obrázek 16: Návrh grafického uživatelského rozhraní.)



Obrázek 16: Návrh grafického uživatelského rozhraní.

Druhé okno slouží k zobrazení výsledků hledání podobných struktur vzoru v kódu. Dělí se na čtyři hlavní části: entity vzoru, doporučené entity v kódu, zdrojový kód a část obsahující kód, který schází dodefinovat. První část zobrazuje ve stromové struktuře nalezené entity návrhového vzoru (třídy, rozhraní a v nich atributy, metody, vlastnosti apod.) Druhá část zobrazuje ve stromové struktuře entity nalezené ve zdrojovém kódu, které jsou doporučené ke zvolené entitě vzoru z první části. Třetí část zobrazuje ty prvky, které zbývají dodefinovat v kódu, aby vybraná entita vzoru byla v kódu zcela reprezentována. Čtvrtá část zobrazuje zdrojový kód nalezené entity v kódu (odpovídá aktuálně vybranému prvku v druhé části).

Po kliknutí na tlačítko *Další* dojde k finální refaktorizaci zdrojových kódů, které jsou následně uloženy. Po kliknutí na tlačítko *Zpět* dojde k návratu do prvního okna. Po kliknutí na tlačítko *Generovat* dojde k modifikaci vybraného prvku zdrojového kódu z druhé části tak, aby plně realizoval prvek vzoru vybraného v první části.

5.3 Logika aplikace

Vrstva logiky (také nazývaná business vrstva) přijímá požadavky od prezentační vrstvy, zpracovává je, provádí příslušné výpočty a případně vrací výsledek výpočtů prezentační vrstvě, která ho zobrazí. Dohromady ji tvoří balíčky poskytující obecné služby logiky (Business), doplňkové služby (Utilities) a služby pro generování zdrojových kódů (Generators).

Balíček *Utilities* obsahuje pomocné služby pro přístup a práci s XML soubory a generování a práci s výjimkami.

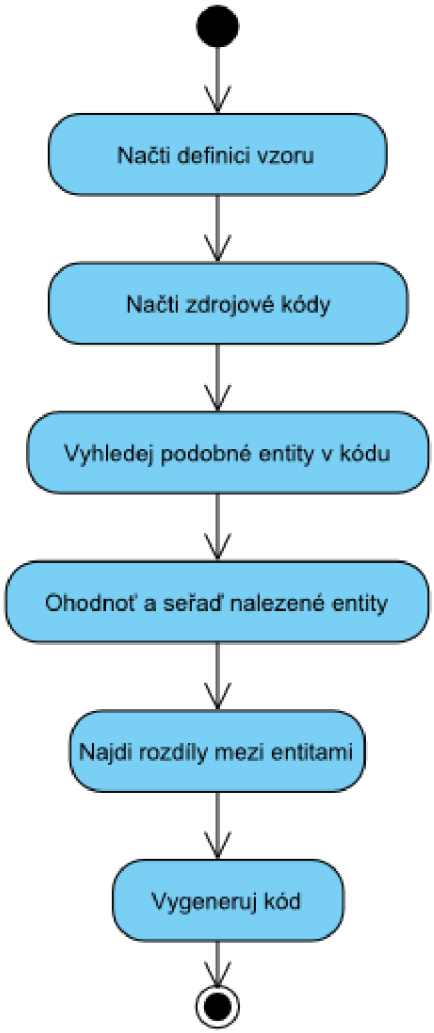
Balíček *Generators* obsahuje služby pro generování zdrojových kódů a vyhledávání podobných struktur ve zdrojovém kódu.

Balíček *Business* obsahuje hlavní logiku aplikace. Tato logika se skládá z několika částí: logika pro práci s definicemi návrhových vzorů, logika pro práci se zdrojovým kódem, parser jazyka C#, logika pro vyhledávání podobných entit vzoru v kódu, validátory, abstraktní datové struktury, pomocí kterých jsou reprezentovány popisy vzorů, ale i zdrojové kódy, a i generátor kódu v jazyce C#.

Logika aplikace se na nejvyšší úrovni abstrakce skládá z těchto částí:

1. Načtení definice vzoru a zdrojových kódů,
2. Vyhledání podobných entit v kódu,
3. Ohodnocení a seřazení nalezených entit,
4. Nalezení rozdílů mezi entitami,
5. Vygenerování kódu.

Na následujícím obrázku je logika aplikace znázorněna pomocí diagramu aktivity.



Obrázek 17: Diagram aktivity logiky aplikace.

Aplikace nejprve načte definici návrhového vzoru, poté zdrojové kódy. Následně v kódech vyhledá podobné struktury, které ohodnotí a seřadí dle relevantnosti. Pak jednotlivé entity vzoru a kódu porovná a najde mezi nimi rozdíly. Poslední činností aplikace je vygenerování kódu takovým způsobem, aby výsledný kód realizoval daný vzor. Jednotlivé kroky algoritmu jsou detailněji popsány v následujících kapitolách.

5.3.1 Reprezentace vzoru

Jedním z hlavních vstupů aplikace je popis vzoru. Tento popis by měl být nezávislý na platformě, snadno rozšiřitelný, přenositelný mezi aplikacemi, srozumitelný uživatelům, protože budou v této formě tvořit definice dalších vzorů, ale také snadno validovatelný, aby aplikace mohla zpracovávat jen správné definice. Proto by bylo vhodné zapisovat definice návrhových vzorů ve formátu XML, který splňuje všechny výše uvedené požadavky. Validita XML dokumentů je zajištěna tak, že se vytvoří XML schéma (např. XSD), pomocí kterého bude aplikace kontrolovat, zda je definice vzoru validní, či nikoliv. Níže následuje příklad definice návrhového vzoru *Singleton* ve formátu XML.

```
<?xml version="1.0" encoding="windows-1250"?>
<pattern name="Singleton" xmlns="http://www.w3.org">
  <entities>
    <class modifier="sealed">
      <name>Class1</name>
      <attributes>
        <attribute>
          <modifiers>
            <modifier>private</modifier>
            <modifier>static</modifier>
            <modifier>readonly</modifier>
          </modifiers>
          <name>attribute1</name>
          <type>Class1</type>
        </attribute>
        <attribute>
          <modifiers>
            <modifier>public</modifier>
            <modifier>static</modifier>
          </modifiers>
          <name>attribute2</name>
          <type>Class1</type>
        </attribute>
      </attributes>
      <methods />
    </class>
  </entities>
  <relations />
</pattern>
```

Kód 7: Příklad definice návrhového vzoru Singleton.

Popis vzoru se skládá ze dvou hlavní částí: entit (uvozeny pomocí *<entities>*) a vztahů mezi nimi (uvozeny pomocí *<relations>*).

Část *<entities>* obsahuje definici entit, které se vyskytují v návrhových vzorech. Jsou zde zejména uloženy informace o atributech, metodách, vlastnostech, konstruktorech. U každého lze definovat více modifikátorů. Část *<relations>* obsahuje definice relací mezi entitami. Jsou zde zejména uvedeny informace o asociacích, dědičnosti, realizaci apod. Každá tato relace „spojuje“ entity, které jsou popsány v předcházející části, a umožňuje určit jejich kardinality.

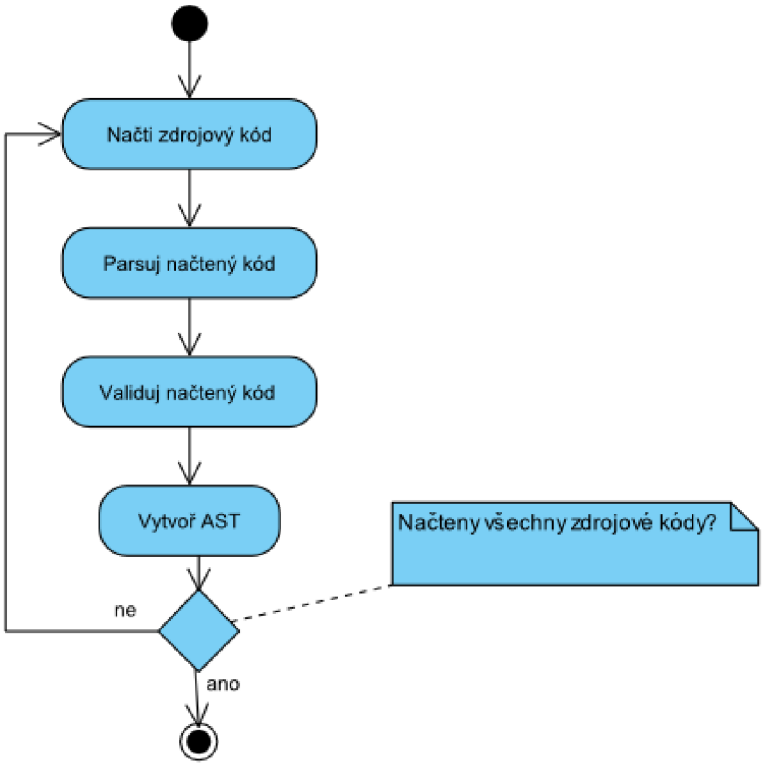
Aby popis vzoru splňoval požadavky na aplikaci popsanou výše, je nutné, aby byl dostatečně abstraktní. Jen díky tomu je pak možné vyhledat podobné struktury vzoru v kódu, které nejsou zcela

stejné se vzorem, ale zároveň dostatečně relevantní. Toho je možné dosáhnout tak, že v popisu vzoru jsou názvy entit (tříd, rozhraní, abstraktních tříd apod.) a datové typy metod, atributů, vlastností uvedeny pouze v abstraktním pojetí.

Datové typy jsou pak uvedeny obecným označením např. *typ1*. Jsou navíc očíslovány, aby bylo možné uvést více typů v rámci entity (např. návratové typy metod v třídě), ale zároveň při zachování možnosti definovat shodnost vlastnosti (atributu nebo metody) k třídě. Algoritmus nalezení vhodných entit (popsán níže) pak nebere v úvahu konkrétní „typ" třídy nebo metody, ale jeho abstraktní variantu. Algoritmus je toto schopen zohlednit při vyhledávání podobných entit ve zdrojovém kódu. Proto jsou na příkladu výše (viz Kód 7: Příklad definice návrhového vzoru Singleton.) třídy uvedeny jako *Class1*, její atributy *attribute1* apod. Díky tomu je popis návrhového vzoru dostatečně obecný, ale zároveň umožňuje plně specifikovat všechny vlastnosti vzoru.

5.3.2 Algoritmus načtení zdrojových kódů

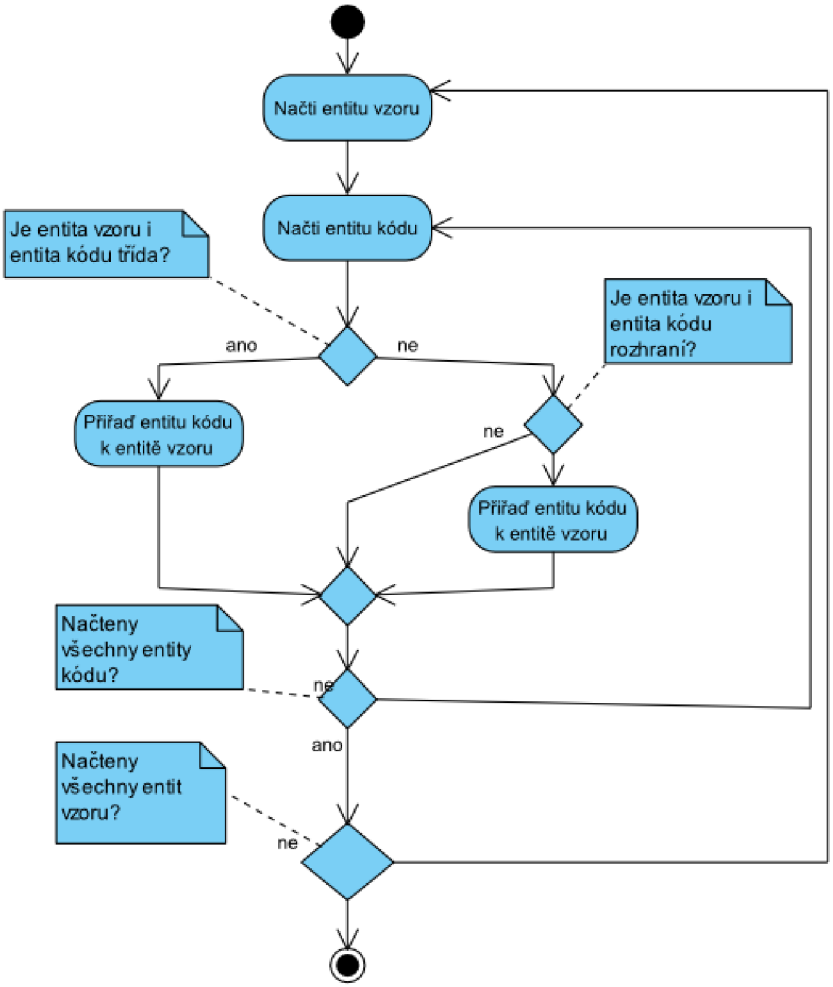
Aby bylo možné efektivně a správně vyhledávat podobné entity vzoru ve zdrojových kódech, porovnávat je a hodnotit je z hlediska relevantnosti, je nutné, aby zdrojové kódy i definice návrhových vzorů byly v rámci algoritmu reprezentovány stejnou uniformní strukturou. Zdrojové kódy jsou načteny pomocí parseru daného programovacího jazyka, jehož výstupem je abstraktní syntaktický strom (AST) daného kódu. Parsování kódů má i další výhodu. Současně s načítáním kódu parser provádí i validaci, neboť pokud načítaný kód není validní, nelze sestavit korektně AST pro daný jazyk. Díky tomu se po korektním načtení zdrojových kódů (a tedy i po úspěšném sestavení AST) převede již načtená definice vzoru taktéž na AST, který je reprezentován shodnou datovou strukturou jako AST zdrojových kódů. Činnost algoritmu pro načtení zdrojových kódů je ilustrována následujícím diagramem aktivity.



Obrázek 18: Diagram aktivity algoritmu pro načtení zdrojových kódů.

5.3.3 Algoritmus nalezení vhodných entit

Další významnou částí logiky aplikace je algoritmus nalezení vhodných entit. Jeho vstupem je popis vzoru a zdrojový kód, oba jsou reprezentovány stejnou strukturou. Cílem je pak vyhledat pro každou entitu vzoru odpovídající entitu ve zdrojovém kódu a její ohodnocení (více viz kapitola Algoritmus ohodnocení nalezených entit). Výsledkem algoritmu je seznam podobných struktur zdrojového kódu k jednotlivým entitám vzoru. Níže následuje diagram aktivity ilustrující činnost algoritmu.



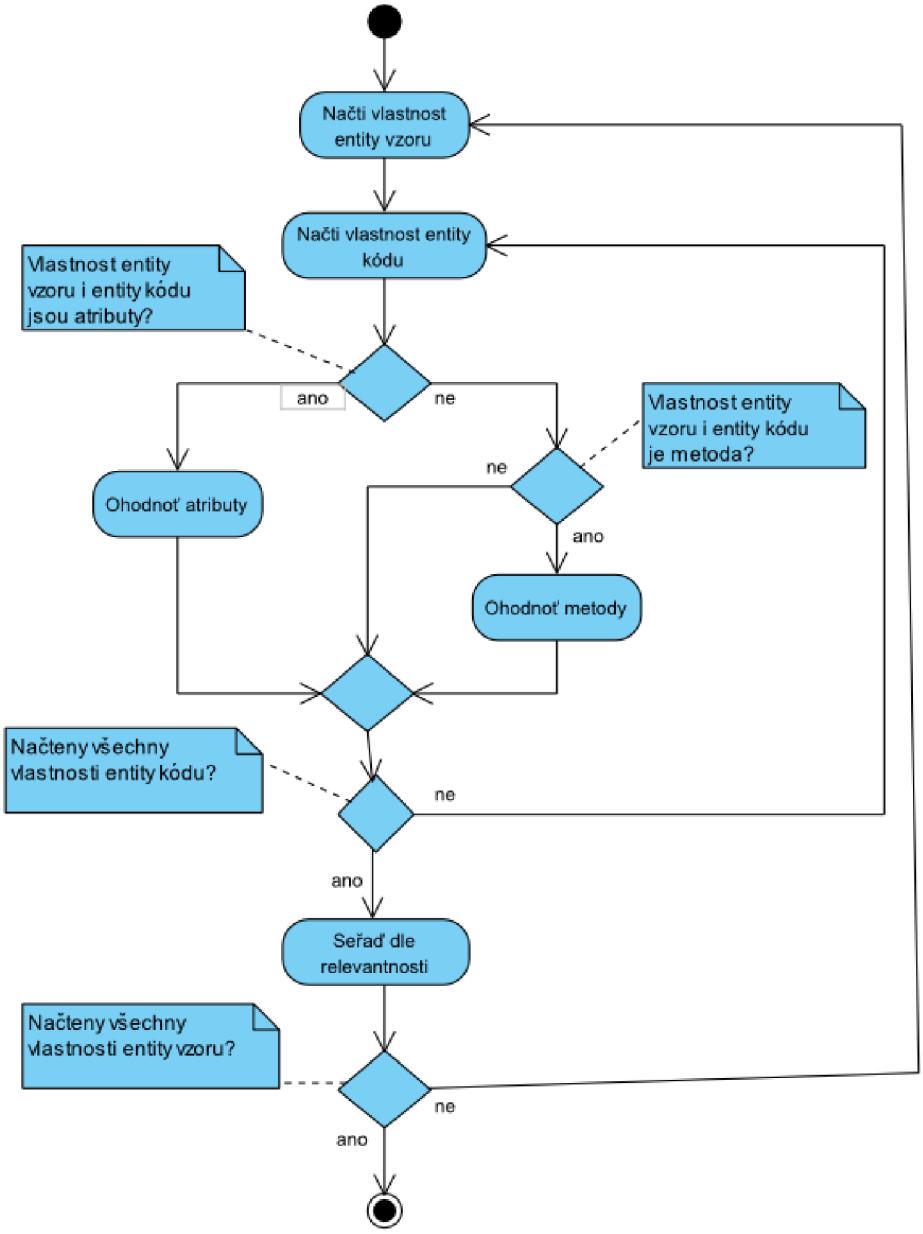
Obrázek 19: Diagram aktivity algoritmu pro nalezení vhodných entit.

Algoritmus postupně prochází entity návrhového vzoru a ke každé najde entity zdrojového kódu odpovídajícího typu. Aby nalezená struktura kódu byla přiřazena k dané entitě vzoru, je nutné, aby se shodoval jejich „typ“, tj. aby obě byly rozhraní nebo třída. Určení vhodnosti nalezené struktury provádí algoritmus, který je popsán v následující kapitole.

5.3.4 Algoritmus ohodnocení nalezených entit

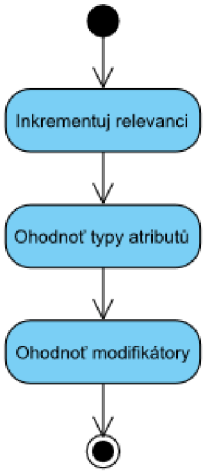
Cílem algoritmu ohodnocení nalezených entit je přiřadit jednotlivým entitám kódu, které byly přiřazeny entitám vzoru, hodnotu, jež určuje míru vhodnosti dané entity, tzv. relevantnost entity. Čím je tato hodnota vyšší, tím je entita vhodnější pro použití při aplikaci vzoru. Obvykle to znamená, že zbývá pro tuto entitu dodefinovat nebo změnit méně vlastností než v případě entity, která má hodnotu nižší. Čím je hodnota relevantnosti nižší, tím je nalezená entita kódu méně vhodná k aplikaci vzoru,

neboť se hodně liší svými vlastnostmi od entity vzoru. Tento algoritmus je velmi důležitým prvkem logiky aplikace, neboť velmi usnadňuje práci uživateli, který nemusí v případě velkého množství nalezených entit v kódu hledat tu vhodnější. Aplikace mu sama vybere ty nejlepší kandidáty a nabídne mu je. Algoritmus určí hodnotu relevantnosti entity dle specifických kritérií. Jeho činnost je ilustrována na následujícím diagramu aktivity.



Obrázek 20: Diagram aktivity algoritmu ohodnocení nalezených entit.

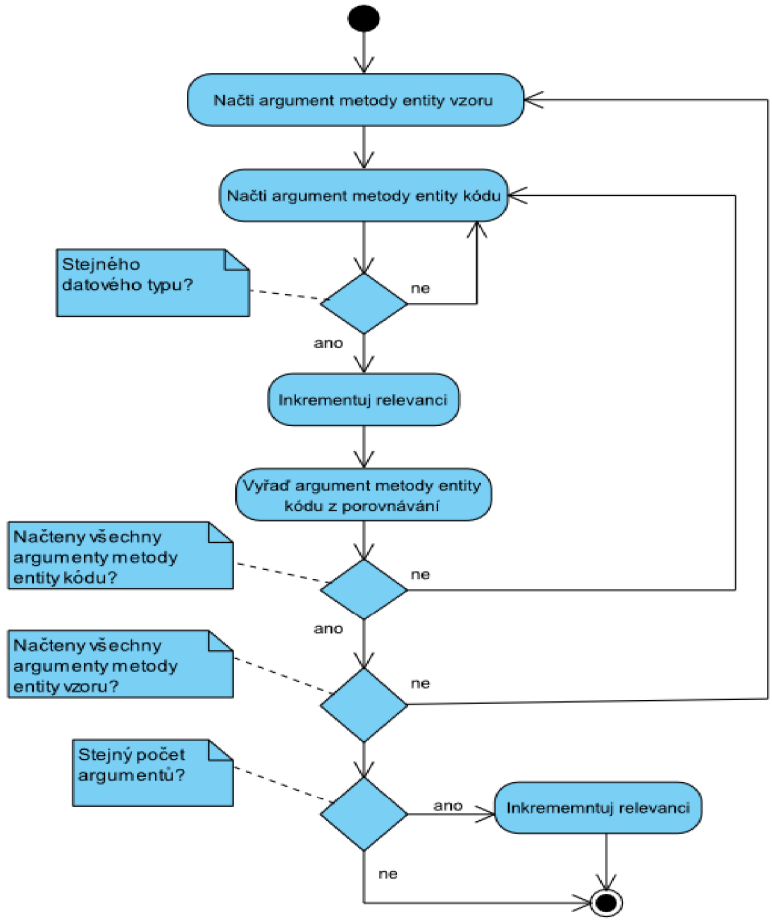
Vstupem algoritmu je vždy entita vzoru a k němu odpovídající entita kódu. Pokud je k entitě vzoru přiřazeno více entit kódu, je tento algoritmus volán pro každou takovou entitu. Algoritmus načítá všechny vlastnosti entity vzoru i kódu (atributy, metody apod.) a společně je ohodnocuje dle konkrétních kritérií. Pokud jsou obě načtené vlastnosti entit shodného typu (metody nebo atributy), algoritmus je ohodnotí, pokud ne, pokračuje se další načtenou vlastností entity kódu. Níže uvedený diagram aktivity ilustruje způsob ohodnocení atributů.



Obrázek 21: Diagram aktivity algoritmu pro ohodnocení atributů entit.

Při ohodnocení atributů je nejprve inkrementována hodnota vlastnosti zkoumané entity značící její relevantnost, neboť shoda „atribut-atribut“ už sama o sobě zvyšuje vhodnost struktury kódu. Dalším hodnotícím kritériem je datový typ atributů. Zde se zkoumá nejen shodnost typů atributů jako takových, tj. zda zkoumaný atribut je stejného datového typu jako atribut entity vzoru, ale i shodnost datového typu atributu k entitě, pod kterou patří. Tuto shodnost zjistí algoritmus z abstraktního zápisu definice vzoru (více viz kapitola 5.3.1. Definice vzoru).

V případě hodnocení metod jsou použita pro hodnocení stejná kritéria jako pro hodnocení atributů, která jsou ale navíc rozšířena o kontrolu argumentů metod. Algoritmus ohodnocení metod je uveden na následujícím diagramu aktivity.

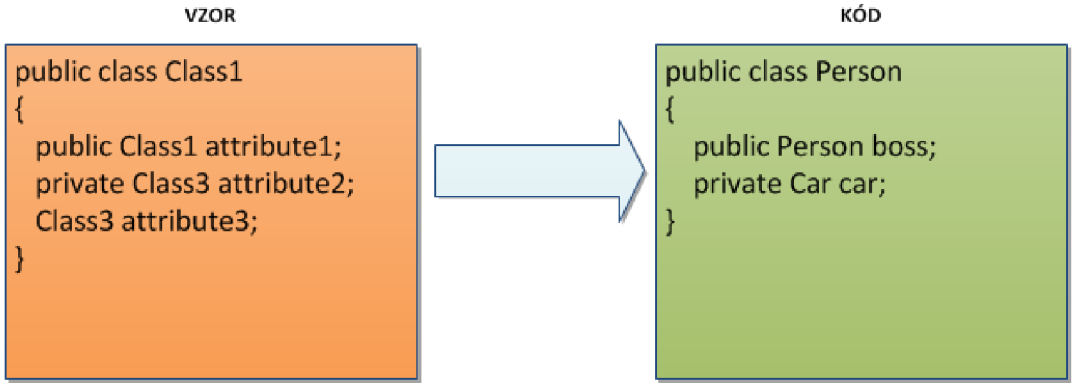


Obrázek 22: Diagram aktivity algoritmu pro ohodnocení argumentů metod.

Relevance zkoumané entity kódu se pak inkrementuje o hodnotu ohodnocení pro jednotlivé vlastnosti. V konečné fázi ještě algoritmus porovná u zkoumaných entit jejich datové typy, a pokud jsou shodné, inkrementuje relevanci zkoumané entity kódu.

5.3.5 Algoritmus nalezení rozdílu mezi entitami

Neméně důležitou částí logiky aplikace je také algoritmus pro nalezení rozdílu mezi entitou vzoru a entitou kódu. Cílem tohoto algoritmu je nalezení takových položek, které chybí entitě zdrojového kódu, aby obsahovala stejné položky jako entita vzoru. Vstupem algoritmu je entita vzoru a k ní nejvíce relevantní entita kódu. Algoritmus ovšem nedělá jen obyčejný rozdíl mezi entitami. Je schopen vyřadit jen takové položky vzoru, které mají nejvhodnější kandidáty v entitě kódu. Následující obrázek ilustruje problém, který může nastat při hledání rozdílu mezi entitami.

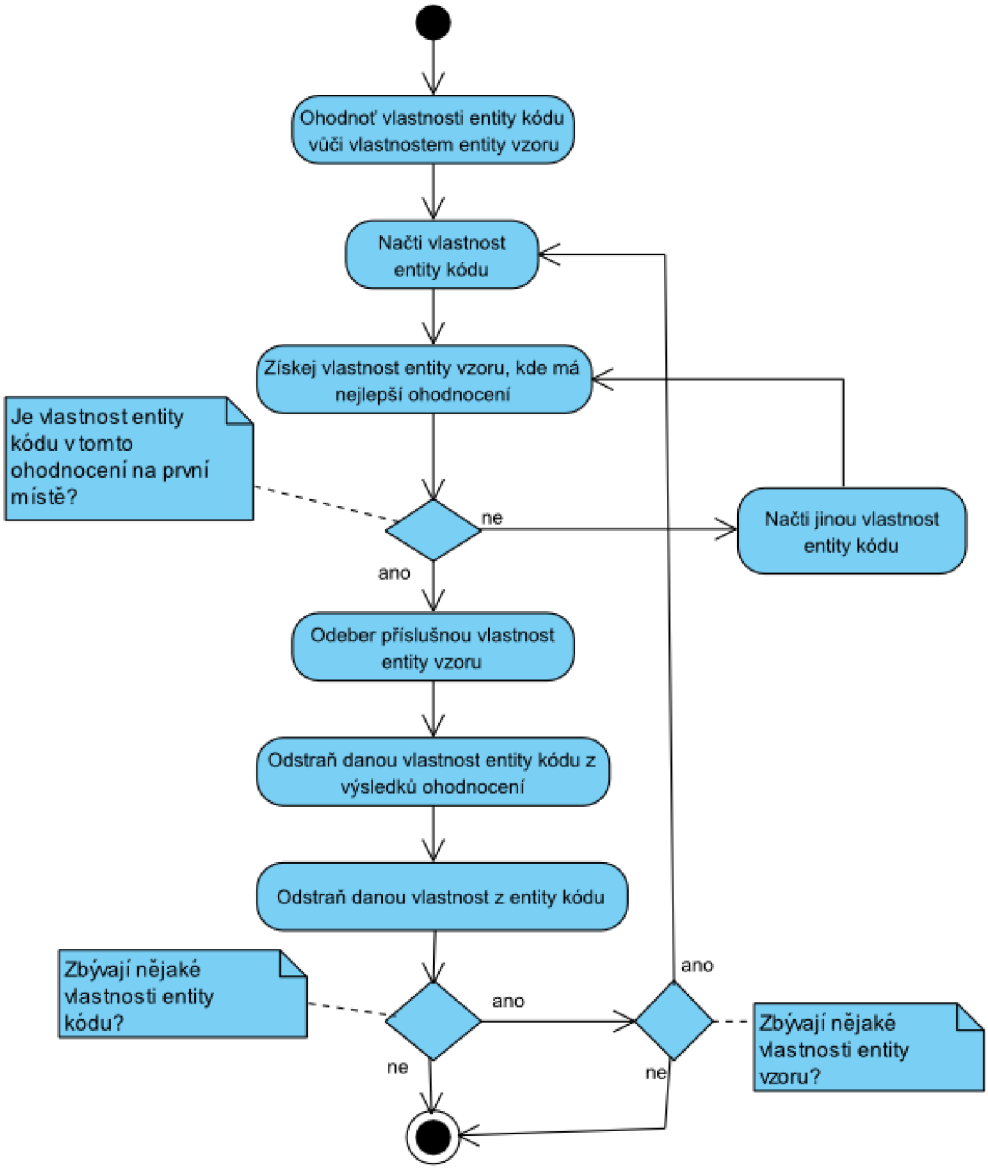


Obrázek 23: Mapování položek entity vzoru na položky entity kódu.

Pokud nastane situace, kdy algoritmus načte entitu vzoru, která má více položek než k němu odpovídající entita kódu, pak pro každou položku entity vzoru (v tomto případě pro každý atribut) bude výsledkem hledání seznam položek entity kódu seřazených dle relevantnosti. V případě uvedeném na obrázku výše budou seznamy vypadat takto:

- attribute1 => [boss(3), car(2)]
- attribute2 => [car(3), boss(2)]
- attribute3 => [boss(1), car(1)]

Pro *attribute1* entity vzoru je nejvhodnějším kandidátem atribut *boss* (relevance 3), protože se shodují v typu a modifikátoru, až poté je to atribut *car* (relevance 2). Opačná situace je pro *attribute2*. Pro *attribute3* je situace na první pohled stejná (nalezené položky v kódu mají stejnou hodnotu relevance), nicméně nejméně výhodná, protože existují výhodnější položky entity vzoru, na které lze „namapovat“ položky entity kódu. Pokud by ale v entitě vzoru bylo více položek, situace by byla mnohem komplikovanější. Algoritmus je schopen vyřešit situaci, kdy jednu položku entity kódu lze „namapovat“ na více položek entity vzoru, ale vzhledem k těmto položkám má různou hodnotu relevance. Činnost algoritmu ilustruje následující diagram aktivity.

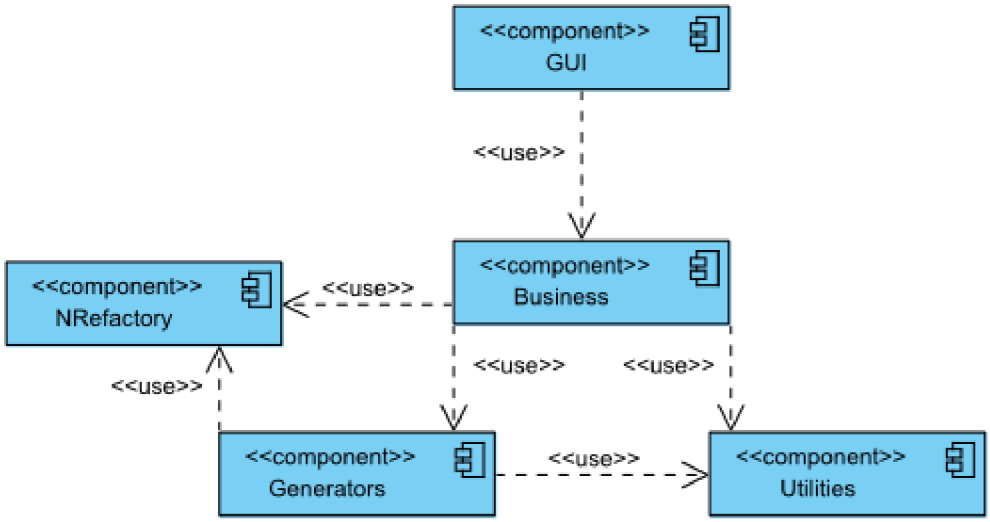


Obrázek 24: Diagram aktivity algoritmu pro nalezení rozdílu mezi entitami.

Algoritmus danou problematiku řeší tak, že nejprve všechny vlastnosti entity kódu ohodnotí vzhledem k jednotlivým vlastnostem entity vzoru. Poté prochází jednotlivé vlastnosti entity kódu a snaží se každou vlastnost „namapovat“ na nejvýhodnější vlastnost entity vzoru. Pokud je načtená vlastnost entity kódu pro své nejlepší ohodnocení pro danou vlastnost entity vzoru na prvním místě, došlo k úspěšnému „namapování“ položky entity kódu na položku entity vzoru a je možné je odebrat, protože v tomto se obě entity neliší. Pokud by ale daná položka nebyla na prvním místě pro položku vzoru, náhodně se vybere jiná, protože je možné, že bude výhodnější. Algoritmus zároveň kontroluje, zda ještě existují položky kódu, které nebyly „namapovány“, a stejně tak položky vzoru, na které se mapuje. Takto je zaručeno, že po skončení algoritmu daná entita vzoru obsahuje jen ty vlastnosti, které je třeba dodefinovat, anebo ke kterým neexistuje žádná vhodná vlastnost entity kódu, nebo jsou nejméně výhodné.

6 Implementace aplikace

Aplikaci jsem naimplementoval pomocí jazyka C# na platformě .NET. Vzhledem k požadavkům na aplikaci jsem prezentační vrstvu implementoval jako desktopovou aplikaci pomocí technologie WinForms. Program jsem navrhl tak, že se skládá z několika hlavních komponent: *GUI, Generators, Utilities, Business* a *NRefactory*. Strukturu systému na fyzické úrovni zobrazuje následující diagram komponent.



Obrázek 25: Diagram komponent zobrazující strukturu na fyzické úrovni.

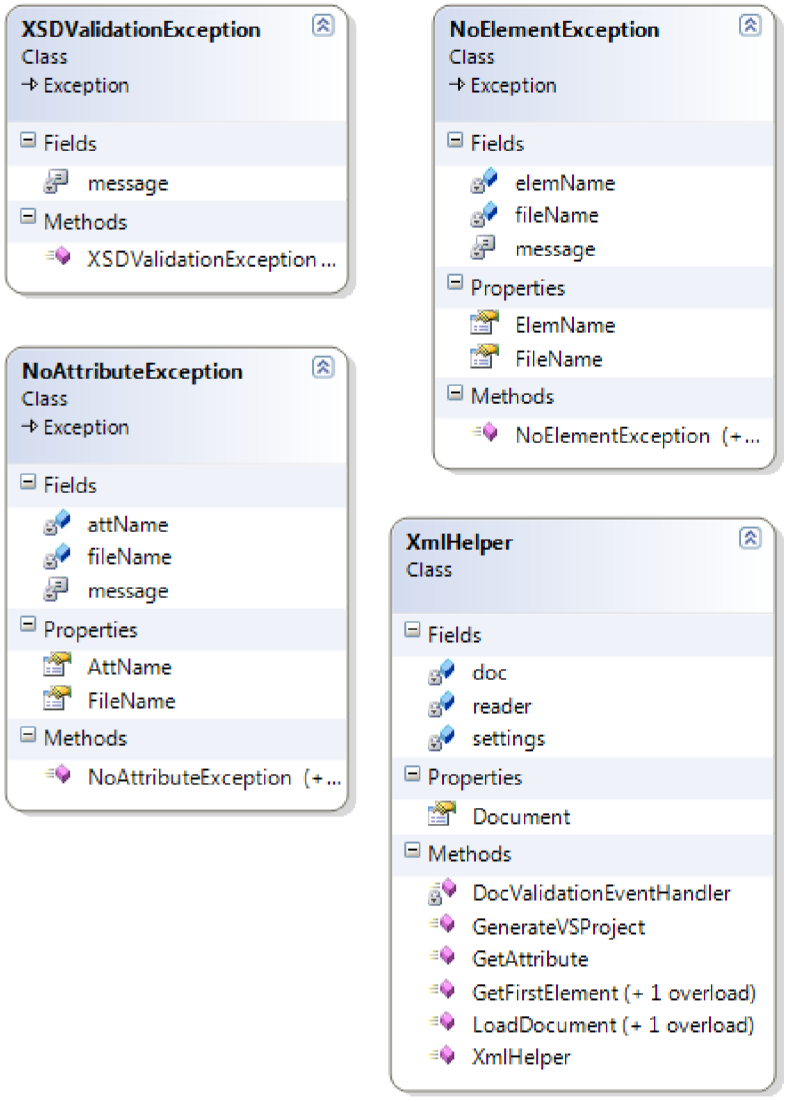
Komponenta *GUI* využívá služeb komponenty *Business*. Ta používá k realizaci svých služeb komponenty *Generators* i *Utilities*. Komponenta *Generators* využívá pouze služeb komponenty *Utilities*. Komponenta *NRefactory* představuje parser zdrojových kódů, který je volně dostupný (open-source) pod licencí GPL. Je součást projektu SharpDevelop, což je projekt open-source vývojového prostředí pro platformu .NET. Tento parser jsem zvolil proto, že z dostupných volně použitelných parserů poskytuje nejlepší funkcionalitu a podporu pro jazyky platformy .NET. Služby komponenty *NRefactory* využívají komponenty *Generators* a *Business*.

6.1 Logika aplikace

Vrstvu logiky aplikace jsem implementoval z komponent *Generators, Utilities*, *Business* a *NRefactory*, které jsou reprezentovány jako samostatné knihovny, které se jen připojí pomocí referencí ke komponentě reprezentující prezentační vrstvu.

6.1.1 Komponenta Utilities

Komponenta *Utilities* obsahuje pomocné třídy pro usnadnění práce s XML soubory a třídy reprezentující jednotlivé výjimky, které mohou nastat při zpracování XML souborů. Služby této komponenty využívá komponenta *Generators* i *Business* pro realizaci svých služeb. Následující diagram tříd zobrazuje strukturu komponenty *Utilities*.



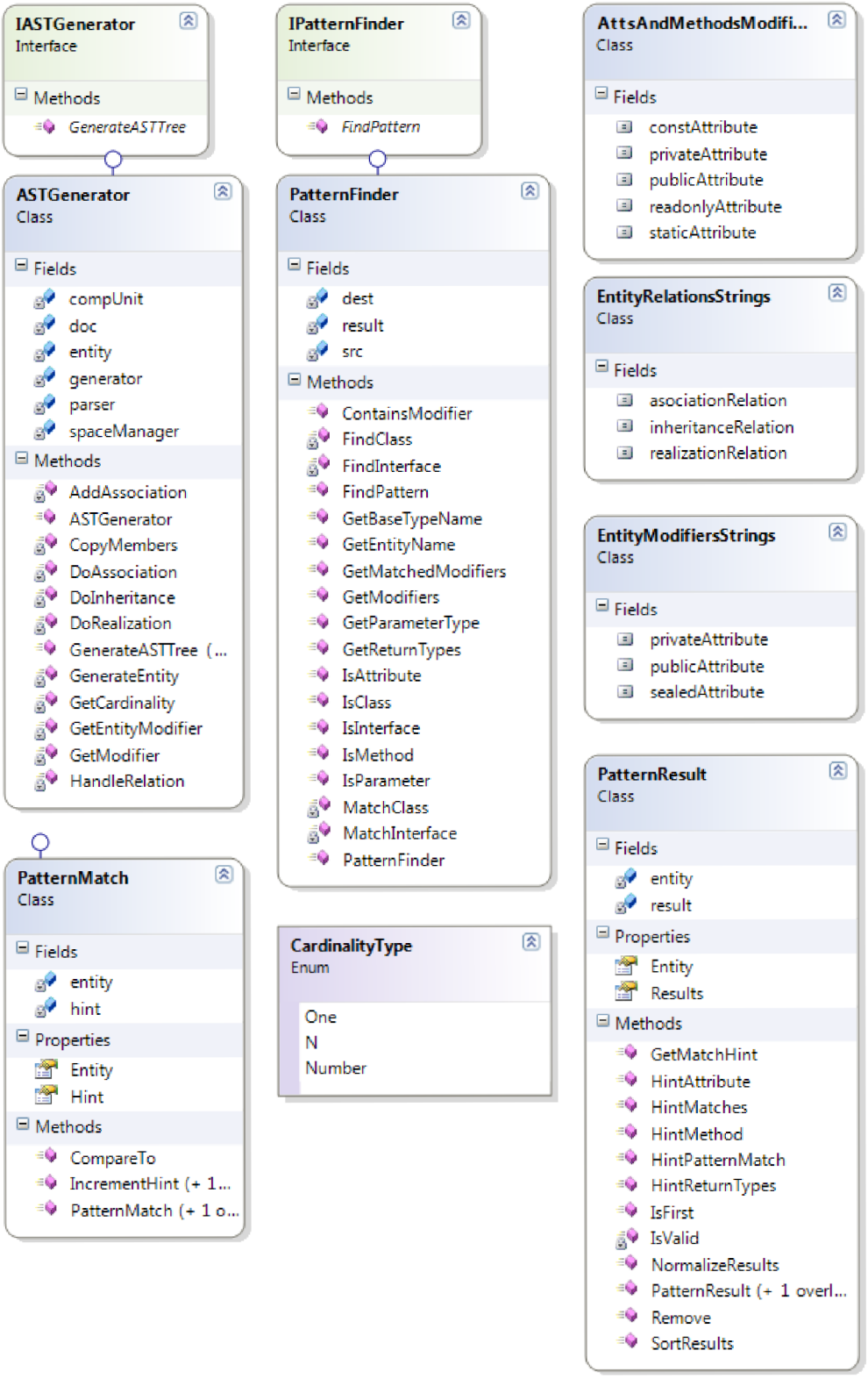
Obrázek 26: Diagram tříd komponenty Utilities.

Třída *XmlHelper* poskytuje služby pro práci s XML soubory. Pomocí ní je možné načíst XML soubor, validovat jej dle zvoleného XSD dokumentu, přistupovat k jednotlivým elementům, jejich atributům či jejich hodnotám. Zároveň obsahuje metodu *GenerateVSProject*, která vygeneruje metasoubor pro projekt do vývojového prostředí Visual Studia.

Třídy *NoAttributeException, NoElementException* a *XSDValidationException* reprezentují jednotlivé výjimky, které mohou nastat při zpracování XML souboru, a je třeba je detailně odlišit od jiných výjimek. Díky nim může algoritmus adekvátně reagovat na nedostatky XML souboru (zejména definice návrhového vzoru). Všechny tyto třídy dědí z třídy *Exception*, která je bázovou třídou, výjimky v prostředí .NET.

6.1.2 Komponenta Generators

Komponenta *Generators* obsahuje třídy pro generování abstraktního syntaktického stromu (AST) z definic vzorů nebo zdrojových kódů, třídy umožňující vyhledávání podobných prvků vzoru v AST a třídy reprezentující výsledky těchto hledání. Služby této komponenty využívá komponenta *Business*. Následující diagram tříd zobrazuje strukturu komponenty *Generators*.



Obrázek 27: Diagram tříd komponenty Generators.

Třída *ASTGenerator* reprezentuje generátor abstraktního syntaktického stromu (AST). Metoda GenerateASTTree je přetížená a umožňuje vygenerovat AST v uniformní datové struktuře buď z definice návrhového vzoru, nebo z načteného zdrojového kódu, který je reprezentován třídou *CompilationUnit*. Tato třída obsahuje privátní metody pro generování AST z jednotlivých prvků tříd vzoru (atributy, metody atd.) a realizuje rozhraní *IASTGenerator*.

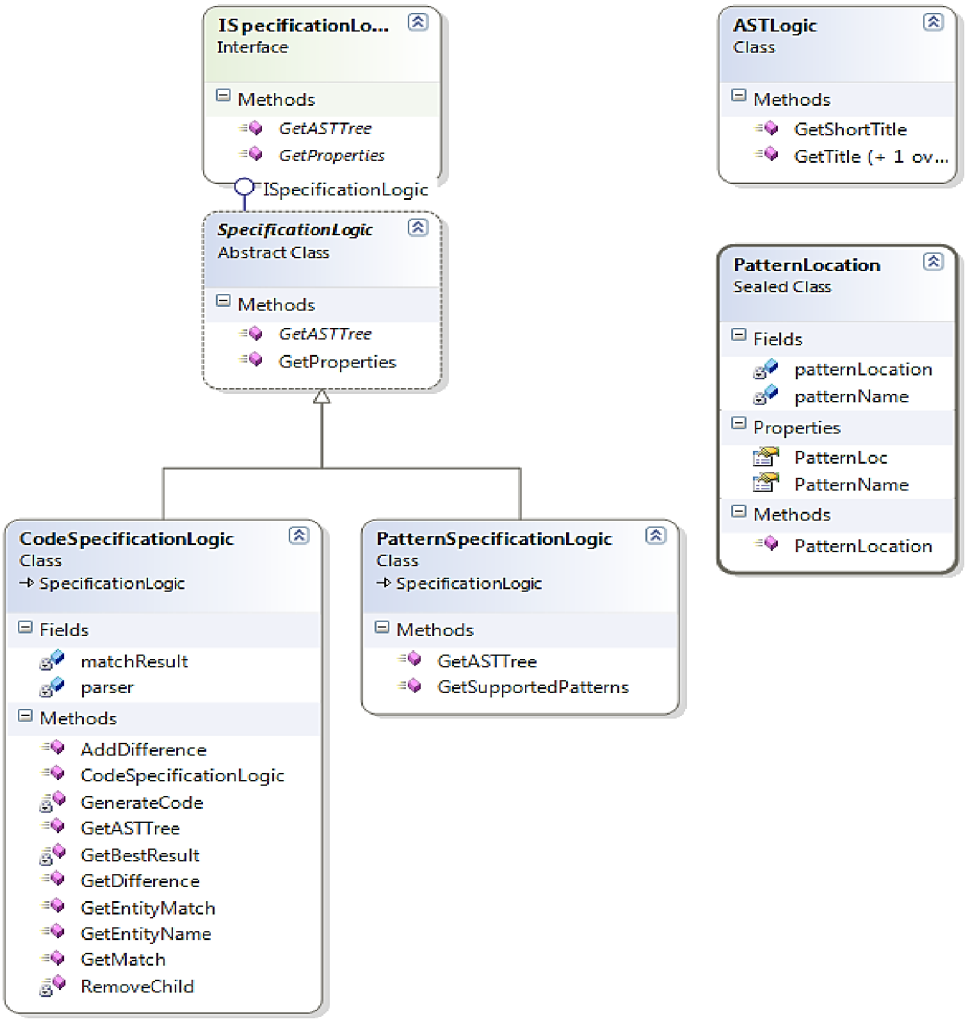
Další významnou třídou je *PatternFinder*, která reprezentuje logiku vyhledávání vzoru nebo jeho části v AST zdrojových kódů. Tato třída implementuje rozhraní *IPatternFinder*, která definuje jednu hlavní metodu *FindPattern*.

V komponentě se dále nacházejí třídy *PatternMatch*, respektive *PatternResult*. Třída *PatternMatch* reprezentuje nalezenou podobnou strukturu ve zdrojovém kódu včetně jejího ohodnocení, které určuje relevantnost nalezené entity vůči hledané entitě vzoru. Třída *PatternResult* reprezentuje výsledky takového hledání formou seznamu instancí třídy *PatternMatch* pro danou entitu vzoru.

Třídy *EntityModifiersStrings*, *AttsAndMethodsModifiersStrings* a *EntityRelationsStrings* obsahují pouze řetězcové konstantní proměnné, které reprezentují možné modifikátory entit, atributů a metod nebo relací, které jsou využity při generování AST. Výčtový typ *CardinalityType* definuje možné kardinality vztahů. Jeho využívá opět generátor AST.

6.1.3 Komponenta Business

Komponenta *Business* obsahuje zejména třídy pro práci se specifikacemi návrhových vzorů a zdrojovými kódy. Její služby využívá komponenta realizující uživatelské rozhraní. *Business* využívá k realizaci svých služeb služby komponenty *Generators* i *Utilities*. Následující diagram tříd zobrazuje strukturu komponenty *Business*.



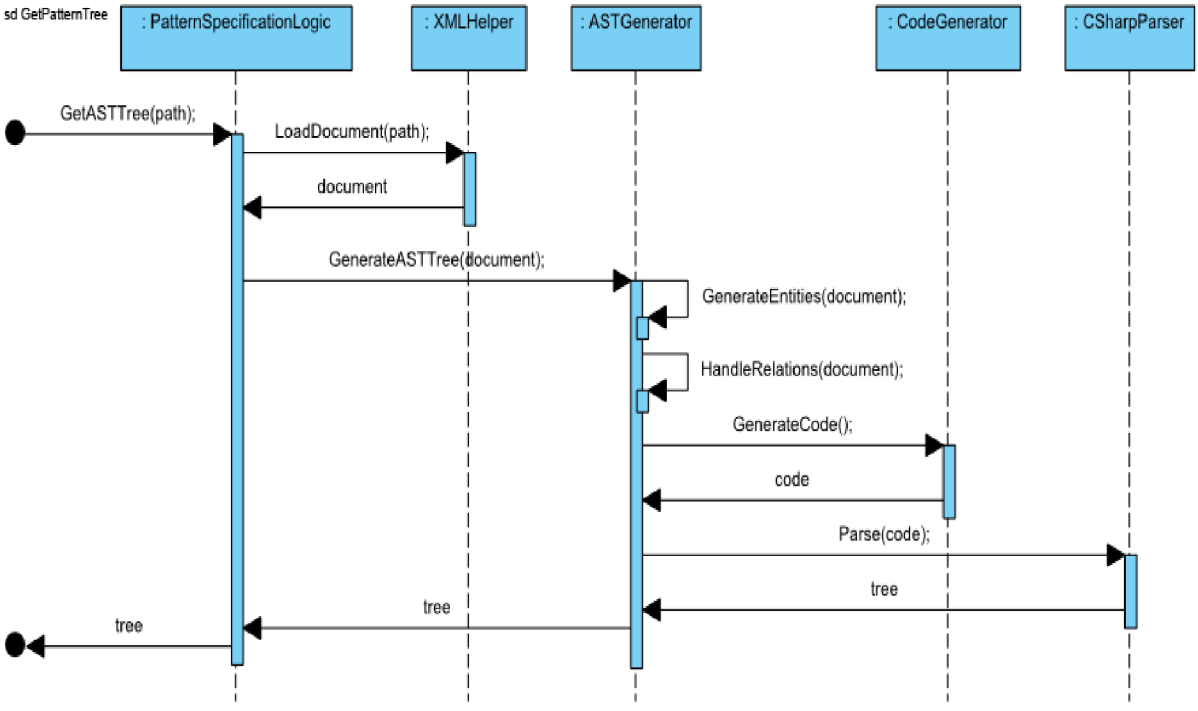
Obrázek 28: Diagram tříd komponenty Business.

Komponenta *Business* obsahuje dvě hlavní třídy: *PatternSpecificationLogic* a *CodeSpecificationLogic*. Obě tyto třídy dědí z abstraktní třídy *SpecificationLogic*, která implementuje rozhraní *ISpecificationLogic*. Třída *SpecificationLogic* definuje metodu *GetASTTree*, která ze zadaného vstupu (buď definice vzoru, nebo zdrojového kódu) vytvoří abstraktní syntaktický strom (AST). Chování této metody záleží na typu instance. Třída *CodeSpecificationLogic* reprezentuje logiku pro práci se zdrojovými kódy. Implementuje zejména metody pro reprezentaci zdrojových kódů jako AST, vyhledání entity vzoru ve zdrojovém kódu a modifikaci entit zdrojových kódů tak, aby realizovaly daný vzor. Třída *PatternSpecificationLogic* implementuje metody pro získání AST z definice vzoru a zjištění všech existujících validních definic návrhových vzorů určených adresářovou cestou.

Business ještě obsahuje třídy *PatternLocation*, která reprezentuje cestu a další potřebné informace k jednotlivým načteným definicím vzoru, a *ASTLogic*, která reprezentuje logiku pro práci s AST na vyšší úrovni.

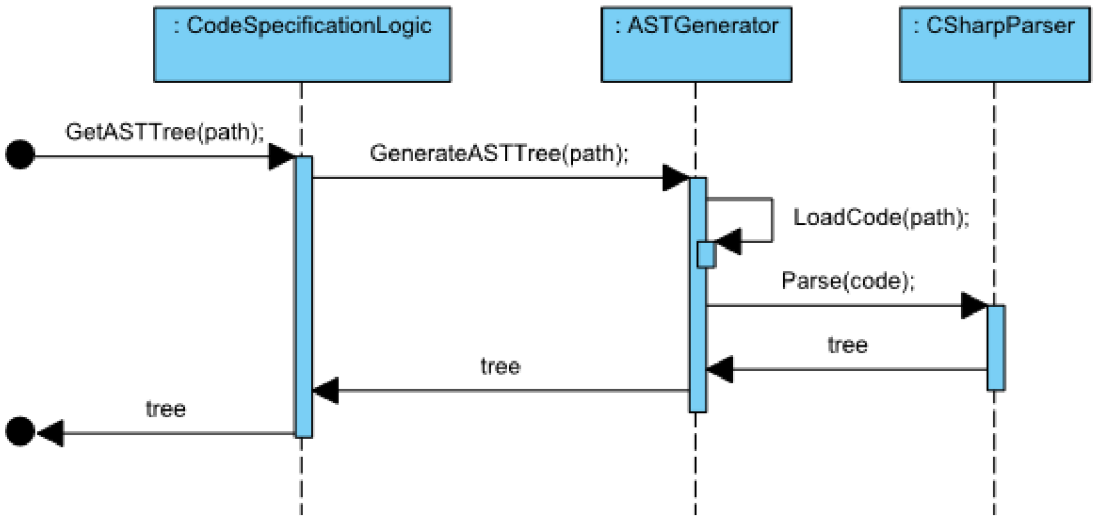
6.1.4 Načtení definice vzoru a zdrojových kódů

Načtení definice vzoru a její reprezentaci formou abstraktního syntaktického stromu realizuje metoda *GetASTTree* v třídě *PatternSpecificationLogic*. Definice vzoru je metodě předána jako cesta k souboru v adresářové struktuře. Metoda nejprve načte daný soubor pomocí třídy *XMLHelper* metodou *LoadDocument* v komponentě *Utilities*. Z načtené definice vzoru se AST vygeneruje pomocí generátoru, který se nachází v komponentě *Generators* a je reprezentován třídou *ASTGenerator*, voláním metody *GenerateASTTree*. Tato metoda reprezentuje definici vzoru v XML jako strukturu kódu pomocí CodeDom, kterou následně vygeneruje do programovacího jazyka totožného s jazykem vstupního zdrojového kódu aplikace. Následně je kód načten parserem NRefactory, který používá aplikace i pro načítání zdrojových kódů. Tím je zajištěno, že zdrojový kód i definice návrhového vzoru jsou reprezentovány stejnou datovou strukturou. Výsledek metody je reprezentován instancí třídy *CompilationUnit*. Následující diagram sekvence zobrazuje chování systému při načítání definice vzoru.



Obrázek 29: Diagram sekvence zobrazující chování systému při načítání definice vzoru.

Načtení zdrojových kódů a jejich reprezentace abstraktním syntaktickým stromem realizuje metoda *GetASTTree* ve třídě *CodeSpecificationLogic*. Zdrojový kód je metodě předán jako adresářová cesta k příslušnému souboru. Ze souboru se AST vygeneruje pomocí generátoru, který se nachází v komponentě *Generators* a je reprezentován třídou *ASTGenerator*, voláním metody *GenerateASTTree*. Tato metoda načte zdrojový kód ze souboru a předá ho parseru NRefactory, který používá aplikace i pro načítání definice vzorů. Tím je zajištěno, že zdrojový kód i definice návrhového vzoru jsou reprezentovány stejnou datovou strukturou. Výsledek metody je reprezentován instancí třídy *CompilationUnit*. Následující diagram sekvence zobrazuje chování systému při načítání zdrojových kódů.



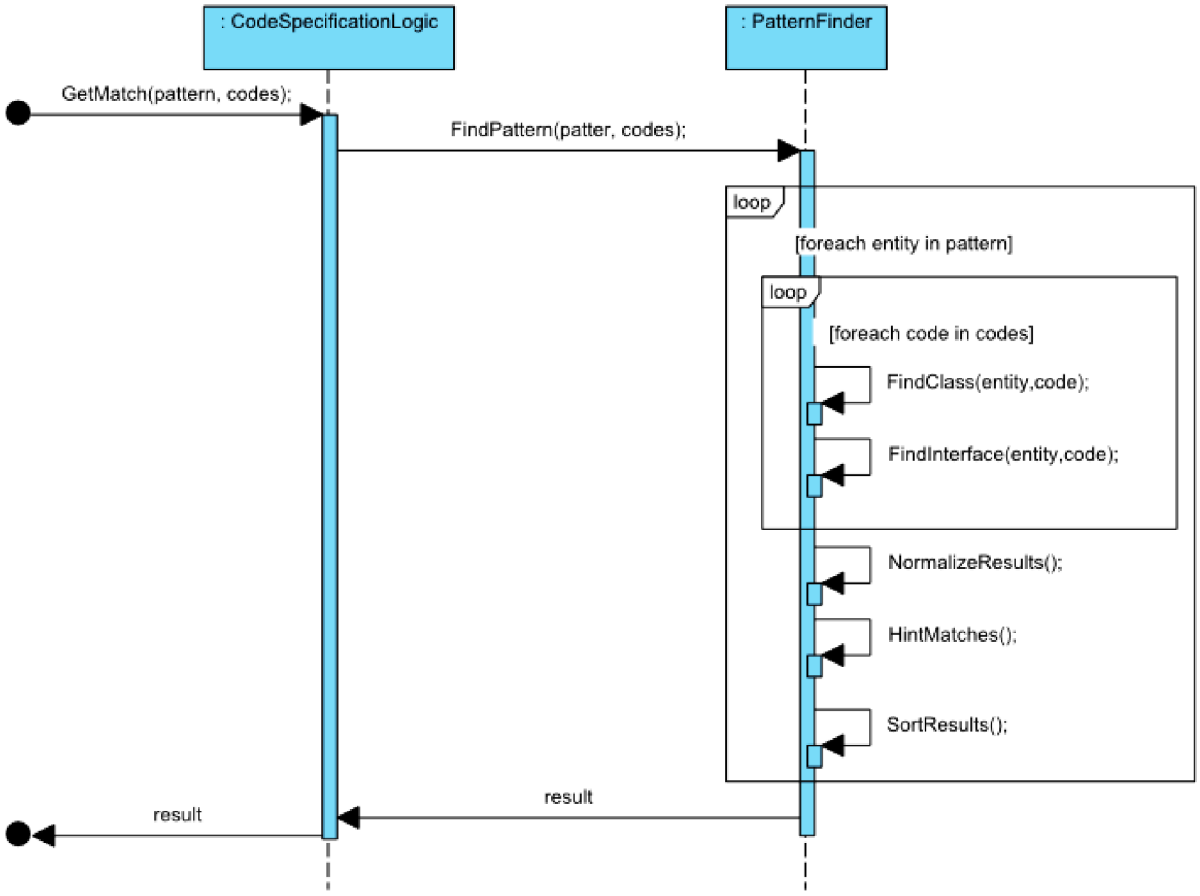
Obrázek 30: Diagram sekvence zobrazující chování systému při načtení zdrojových kódů.

6.1.5 Vyhledání podobných entit ve zdrojovém kódu

Vyhledání podobných entit vzoru ve zdrojovém kódu následuje až poté, kdy jsou načteny definice vzoru a zdrojové kódy. Tuto funkci zajišťuje pomocí svých tříd komponenta *Business* při využití služeb komponenty *Generators*. Vyhledání podobných entit umožňuje metoda *GetMatch* třídy *CodeSpecificationLogic* v komponentě *Business*. Tato metoda přijímá dva parametry: AST strom reprezentující návrhový vzor a seznam AST stromů reprezentující načtené zdrojové kódy. Metoda vytvoří novou instanci třídy *PatternFinder* z komponenty *Generators* a zavolá její metodu *FindPattern*, které předá AST stromy vzoru a zdrojových kódů. Hlavní funkčnost realizuje tedy třída *FindPattern*.

Tato třída v rámci své metody nejprve začne načítat jednotlivé entity vzoru. Pro každou načtenou entitu vytvoří instanci třídy *PatternResult* z komponenty *Generators*, která reprezentuje výsledky hledání pro jednotlivé entity vzoru. Poté začne procházet každou entitu zdrojového kódu a dle typu aktuálně načtené entity vzoru (třída nebo rozhraní) bude hledat podobné struktury ve zdrojovém kódu. Hledání realizují metody *FindClass* a *FindInterface* třídy *PatternFinder*. Tyto metody načtou entitu vzoru a aktuálně zkoumaný AST strom zdrojového kódu a vyhledají podobné struktury. Pro každou vhodnou nalezenou entitu zdrojového kódu tyto metody vytvoří instanci třídy *PatternMatch*, která reprezentuje nalezenou entitu zdrojového kódu včetně ohodnocení (hodnota určující relevantnost entity zdrojového kódu vůči entitě vzoru). Výsledkem činnosti metody *FindClass*, respektive *FindInterface*, je seznam nalezených podobných struktur zdrojového kódu ke konkrétní entitě vzoru, který je reprezentován jako seznam instancí třídy *PatternMatch* a který je jako celek „zastřešen“ instancí třídy *PatternResult*.

Následně metoda *FindPattern* pro každý výsledek hledání vyřadí takové výsledky, které nejsou vůbec vhodné, ohodnotí je (více viz následující kapitola) a seřadí dle relevantnosti. Výsledkem činnosti této metody je seznam instancí třídy *PatternResult*, kde pro každou entitu návrhového vzoru existuje právě jedna instance této třídy reprezentující nalezené struktury ve zdrojovém kódu. Následující diagram sekvence zobrazuje chování systému při vyhledání podobných entit ve zdrojovém kódu.

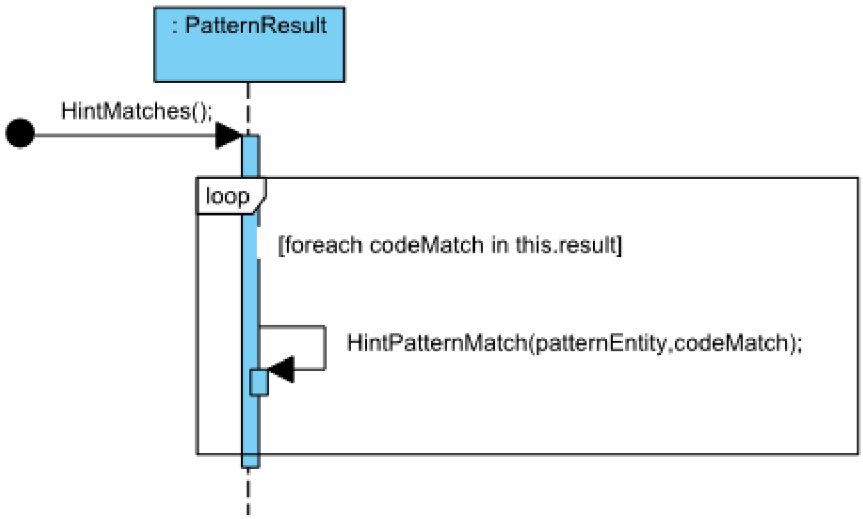


Obrázek 31: Diagram sekvence zobrazující chování při vyhledání entit ve zdrojovém kódu.

6.1.6 Ohodnocení nalezených entit

Ohodnocení nalezených entit se provádí bezprostředně poté, co jsou známy výsledky hledání podobných struktur ve zdrojovém kódu vůči entitě vzoru. Ohodnocení je provedeno na základě volání metody *HintMatches* třídou *PatternFinder*, která má informace o hledání (viz zpráva *HintMatches* v předchozím diagramu sekvence). Výsledky hledání reprezentuje třída *PatternResult* z komponenty *Generators*, která obsahuje entitu vzoru a nalezené podobné struktury zdrojového kódu reprezentované jako seznam instancí třídy *PatternMatch*.

Třída *PatternResult* obsahuje metodu *HintMatches*, která provede ohodnocení tak, že pro každou nalezenou entitu ve zdrojovém kódu a danou entitu návrhového vzoru zavolá metodu *HintPatternMatch* třídy *PatternResult*. Výše popsané chování systému zobrazuje následující diagram sekvence.



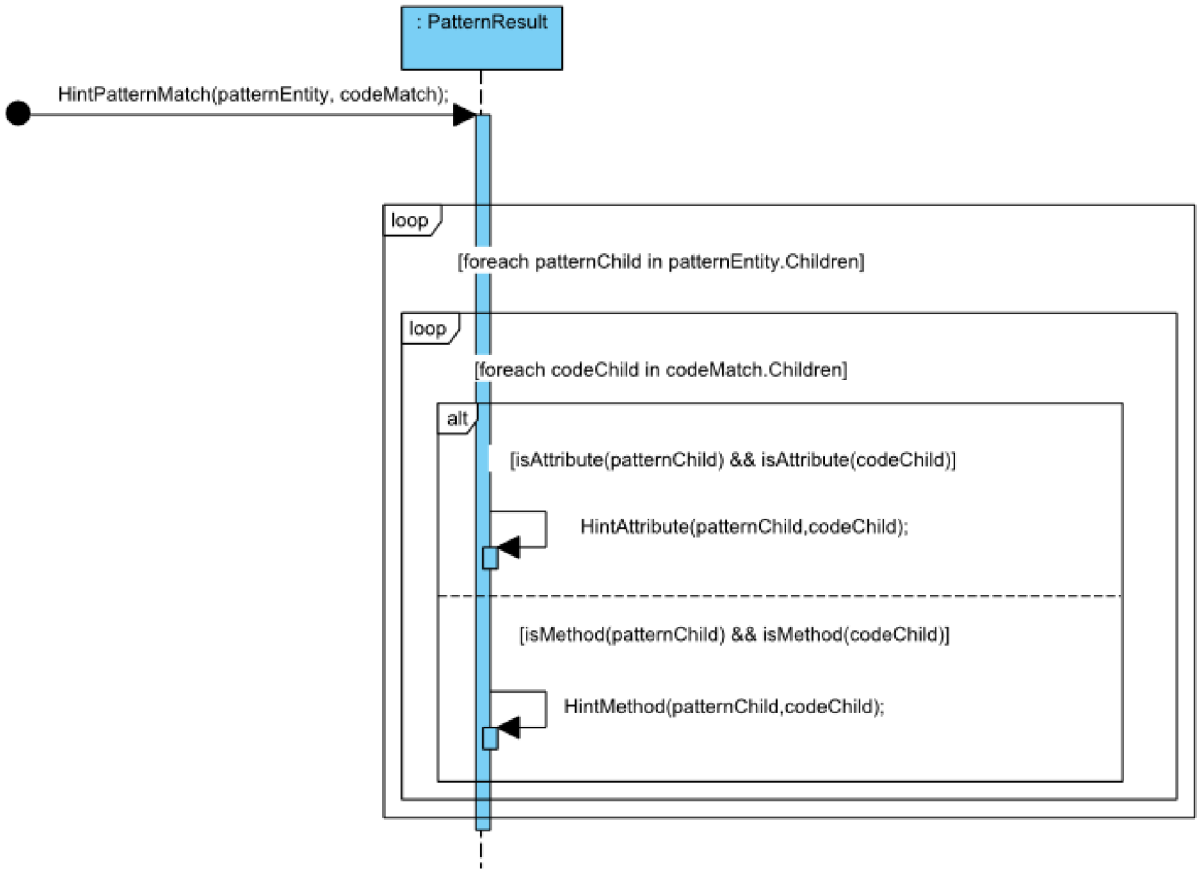
Obrázek 32: Diagram sekvence zobrazující chování při ohodnocení nalezených entit kódu.

Metoda *HintPatternMatch* začne procházet všechny atributy a metody entity vzoru a ke každé vždy načte jednotlivě všechny atributy a metody dané entity zdrojového kódu a ohodnotí je. Způsob ohodnocení se liší dle typu načteného prvku, tj. způsob ohodnocení se liší pro atributy a metody zvlášť. Ohodnocení atributů zajišťuje metoda *HintAttribute* třídy *PatternResult*, ohodnocení metod zajišťuje metoda *HintMethod* té samé třídy.

Obě metody při hodnocení kontrolují mnoho různých kritérií. U atributů je to např. shoda v počtu a typu modifikátorů, shoda v typu, shoda v korelaci typu atributu vůči typu entity, u metod je to např. shoda v pořadí, počtu a typu argumentů, modifikátorů, korelace typu jako takového či korelace typu vůči typu entity. Kromě ohodnocení atributů nebo metod metoda *HintPatternMatch* hodnotí jako celek entity vzoru a zdrojového kódu. V tomto případě má vliv na hodnocení např. shoda v dědičnosti nebo realizaci rozhraní, počtu atributů nebo metod apod.

Výsledkem ohodnocení je nezáporné celé číslo, které určuje relevantnost entity zdrojového kódu k entitě vzoru, tj. v podstatě vhodnost modifikace právě té dané entity zdrojového kódu tak, aby realizovala příslušnou entitu vzoru a v konečném důsledku i daný vzor.

Výsledek ohodnocení je uložen pro každou entitu zdrojového kódu samostatně (v instanci třídy *PatternMatch*) a třída *PatternResult* pak s ním pracuje dále, např. při vykonávání metod pro normalizaci výsledků hledání, jejich seřazení od nejvýhodnějších apod. Tuto službu většinou využívají třídy jí v logické hierarchii nadřazené. Následující diagram sekvence popisuje chování systému popsaného výše.



Obrázek 33: Diagram sekvence zobrazující chování systému při ohodnocení atributů a metod.

6.1.7 Zjištění rozdílu mezi entitami

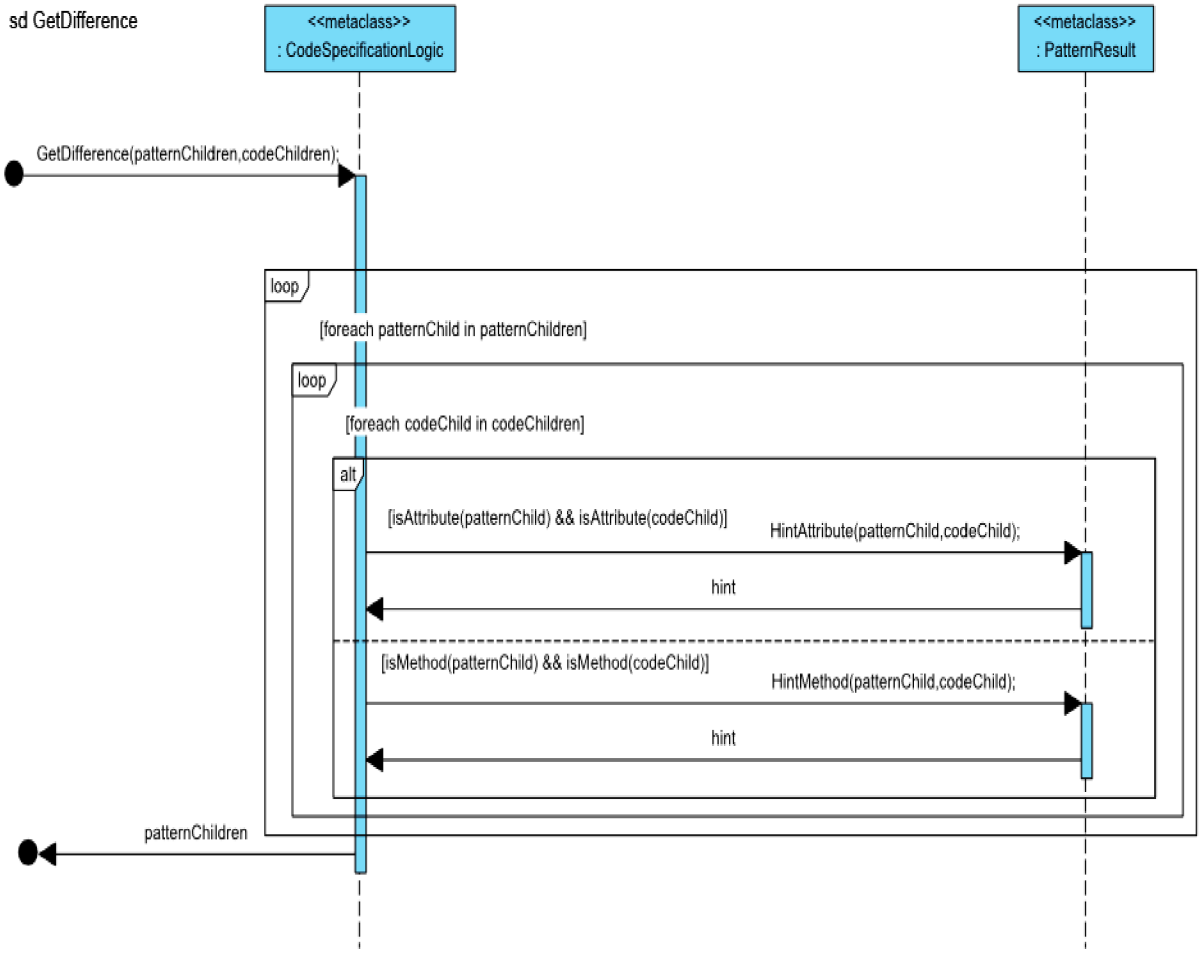
Zjištění rozdílu mezi vybranou entitou návrhového vzoru a entitou zdrojového kódu realizuje statická metoda *GetDifference* třídy *CodeSpecificationLogic*. Metoda přijímá dva parametry: seznam atributů či metod entity vzoru a zdrojového kódu (jsou reprezentovány instancemi třídy *AstNode*, která se nachází v komponentě *NRefactory*). Tato metoda je volána z tříd jí nadřazených (nejčastěji z tříd uživatelského rozhraní po vybrání entity zdrojového kódu k modifikaci). Cílem metody je zjistit, které atributy nebo metody entity zdrojového kódu chybí vůči entitě vzoru tak, aby po případném doplnění daný zdrojový kód odpovídal entitě vzoru.

Metoda nejprve začne procházet jednotlivé položky entity vzoru a pro každou z ní jednotlivé položky entity zdrojového kódu. Poté, pokud se obě aktuálně zpracovávané položky shodují v typu (obě jsou atribut nebo metoda), je provedeno ohodnocení pomocí metody *HintAttribute*, respektive *HintMethod* třídy *PatternResult* (více viz předchozí kapitola). Metoda se v podstatě tímto způsobem pokusí „namapovat“ jednotlivé položky entity kódu na položky entity vzoru a přitom zjistí, jak moc je to „výhodné“ (více viz kapitola 5.3.5). Výsledkem tedy je, že pro každou položku entity vzoru existuje seznam položek entity zdrojového kódu seřazený dle relevantnosti.

V druhé fázi metoda *GetDifference* začne načítat jednotlivé položky entity zdrojového kódu. Pro každou zjistí, jaké je její nejvýhodnější „mapování“ ze všech možností. K tomu používá metodu *GetBestResults* třídy *CodeSpecificationLogic*. Tato třída vrátí instanci třídy *PatternResult*, která reprezentuje v tomto případě položku entity vzoru a k ní vhodné položky entity zdrojového kódu. Následně se pomocí metody *IsFirst* třídy *PatternResult* zjistí, zda je toto „mapování“ pro danou položku entity kódu nejvýhodnější. Pokud ano, je položka odebrána z entity kódu a i příslušná položka vzoru a proces se opakuje, dokud nebyly „namapovány“ všechny položky nebo už byly

vyčerpány všechny položky vzoru. Chováni algoritmu nalezení rozdílu je vyjádřeno pomocí diagramu aktivity v kapitole 5.3.5.

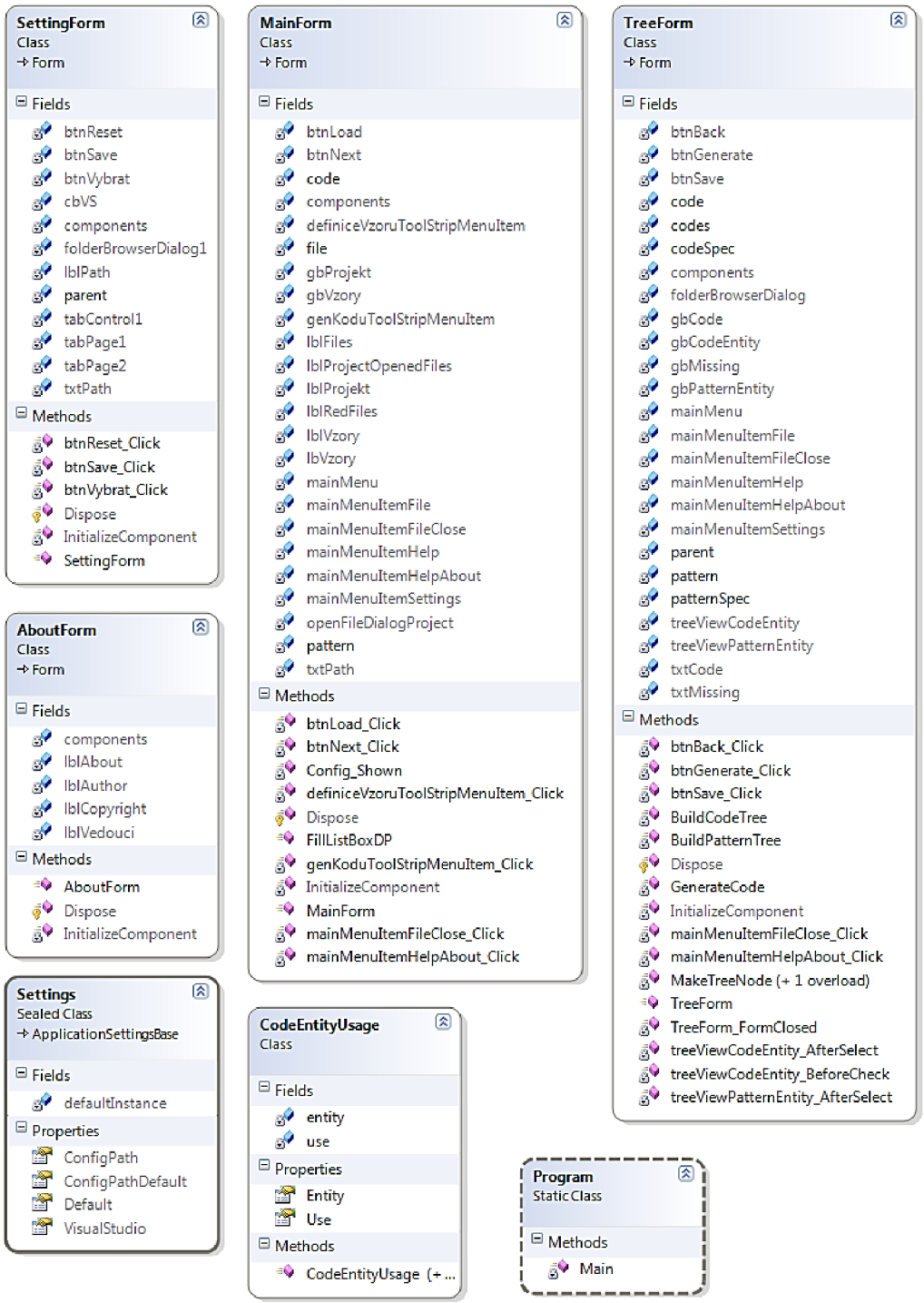
Výsledkem metody *GetDifference* je seznam položek (reprezentovány instancemi třídy *AstNode* komponenty *NRefactory*) entity návrhového vzoru, které schází dodefinovat dané entitě kódu. Následující diagram sekvence zobrazuje chování systému při nalezení rozdílu mezi entitami.



Obrázek 34: Diagram sekvence zobrazující chování systému při zjištění rozdílu mezi entitami.

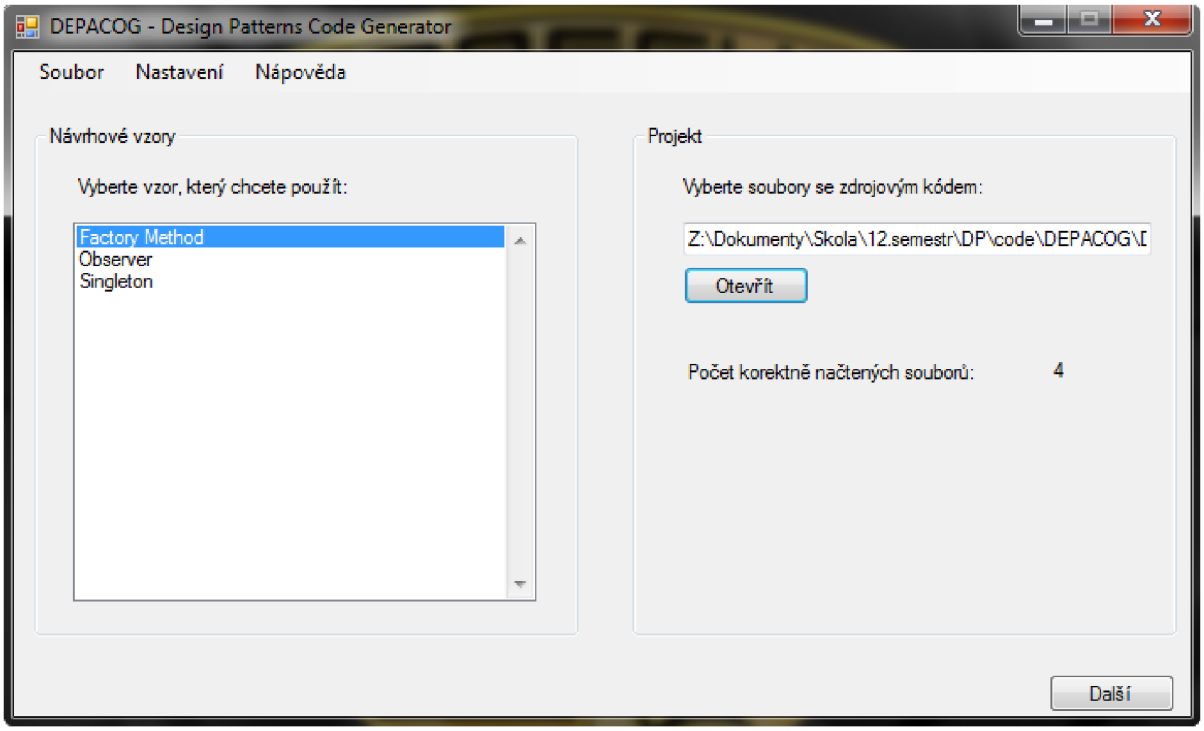
6.2 Uživatelské rozhraní

Prezentační vrstvu jsem implementoval jako desktopovou aplikaci. Její funkčnost zapouzdřuje komponenta *GUI*. Třídy této komponenty jsou zodpovědné za zobrazení formulářů pro načtení definic vzorů, zdrojových kódů, zobrazení nalezených podobných struktur nebo nastavení aplikace. Následující diagram tříd zobrazuje hlavní třídy komponenty *GUI*.



Obrázek 35: Diagram tříd uživatelského rozhraní.

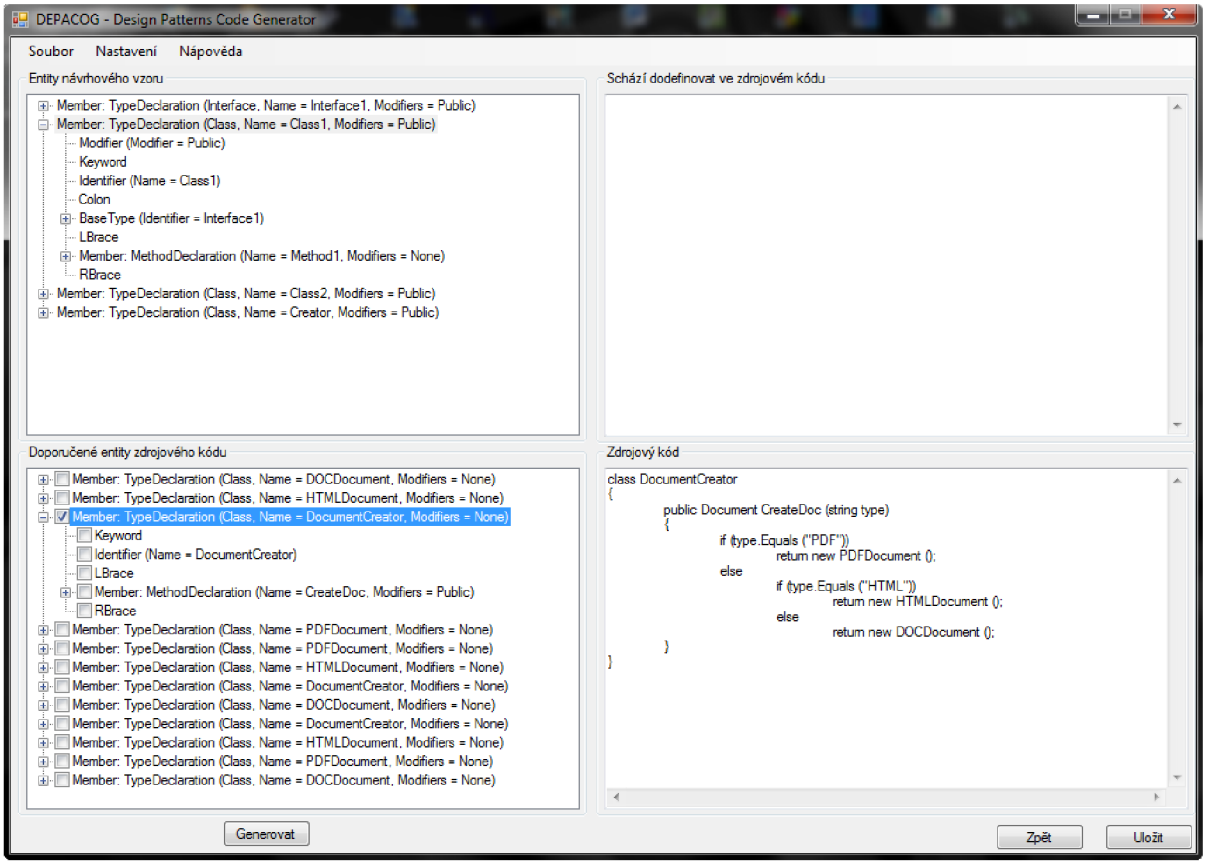
Třída *MainForm* reprezentuje hlavní okno aplikace, které se zobrazí po jejím spuštění. Umožňuje načíst definice vzoru z adresářové cesty uložené v nastavení aplikace a zobrazit uživateli dostupné vzory, které lze použít pro generování kódu. Dále také umožňuje načíst zdrojové kódy z umístění vybraného uživatelem přes dialogové okno a poskytuje jednoduché menu pro nastavení aplikace a zobrazení informací o aplikaci.



Obrázek 36: Hlavní okno aplikace.

Hlavní okno aplikace se skládá ze dvou hlavních částí. Vlevo se nachází dostupné definice vzorů, které byly načteny z umístění dle nastavení aplikace. Uživatel si může pro dané generování kódu vybrat vždy jen jeden vzor. Vpravo se poté zobrazuje cesta k adresáři se soubory se zdrojovými kódy, které se mají použít. Kliknutím na tlačítko další se zobrazí druhé hlavní okno aplikace.

Třída *TreeForm* reprezentuje druhé hlavní okno aplikace. Třída přehledně zobrazuje jednotlivé entity vzoru, k nim nalezené entity zdrojového kódu, jejich originální kód, ale i kód, který schází dodefinovat, aby daná entita zdrojového kódu realizovala daný vzor. Zároveň poskytuje prostředky pro uložení vybraných entit zdrojového kódu do souborů a případné vygenerování projektu do Visual Studia. Níže uvedená ukázka zobrazuje druhý formulář s výsledky hledání.



Obrázek 37: Okno aplikace s výsledky hledání.

Okno s výsledky hledání se skládá ze čtyř hlavních částí. Vlevo nahoře jsou zobrazeny entity vzoru ve stromové struktuře. Vlevo dole jsou nalezené vhodné entity kódu seřazené dle relevantnosti. Nalezené entity zdrojového kódu jsou vždy zobrazeny jen vůči aktuálně vybranému prvku vzoru vlevo nahoře. U každé této entity je zobrazen checkbox, který po zaškrtnutí znamená, že tato entita zdrojového kódu bude použita při výsledném generování kódů a jejich uložení do souboru. Vpravo nahoře se zobrazují prvky, které schází vybrané entitě kódu, aby realizovala entitu vzoru. Vpravo dole pak je zobrazen aktuální kód vybrané entity kódu. Pokud uživatel klikne na tlačítko „Generovat“, přidají se vybrané entitě kódu položky, které jsou uvedeny vpravo nahoře. Pokud klikne „Uložit“, uloží se vybrané entity kódu do souboru zvoleného adresáře.

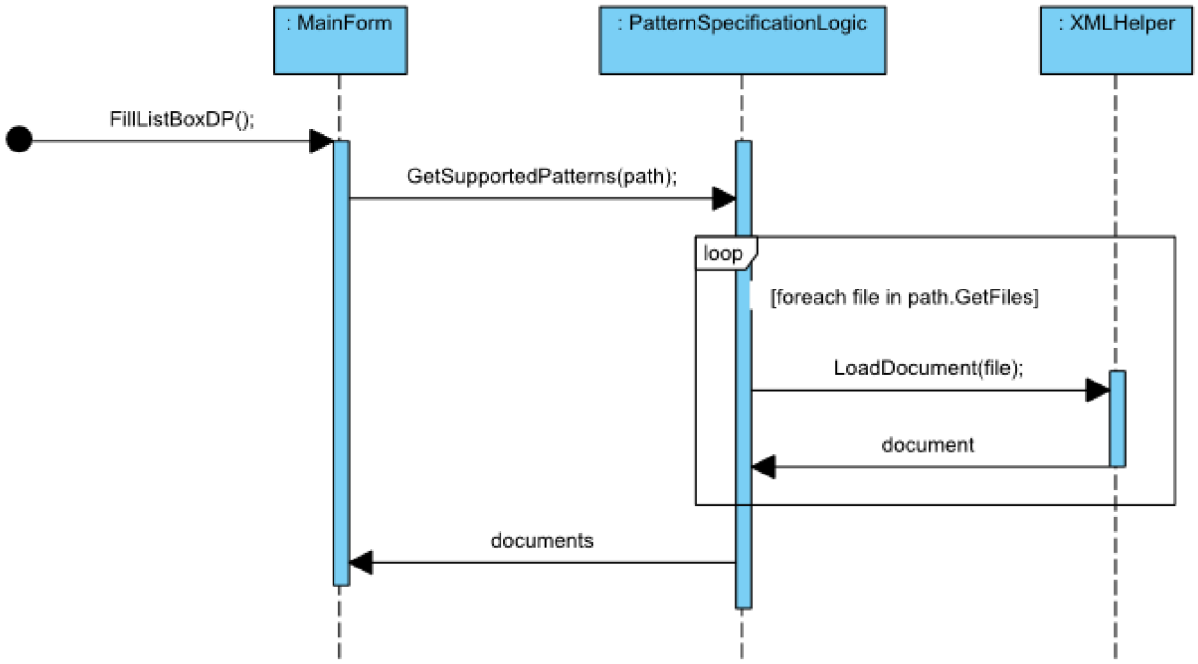
Třída *SettingForm* reprezentuje okno pro nastavení aplikace. Umožňuje měnit výchozí nastavení aplikace pro cestu k adresáři s definicemi vzorů a volbu, zda se má s výslednými soubory generovat i projekt do vývojového prostředí Visual Studio. K uložení nastavení využívá standardní konfigurační nástroje poskytované technologií WinForms. Tyto prostředky jsou reprezentovány třídou *Settings*.

Třída *AboutForm* reprezentuje okno s informacemi o autorovi a třída *Program* představuje výchozí bod po spuštění aplikace. Je zodpovědná za inicializaci všech podstatných komponent ve WinForms a zobrazení hlavního okna po spuštění.

6.2.1 Načtení definice vzorů a zdrojových kódů

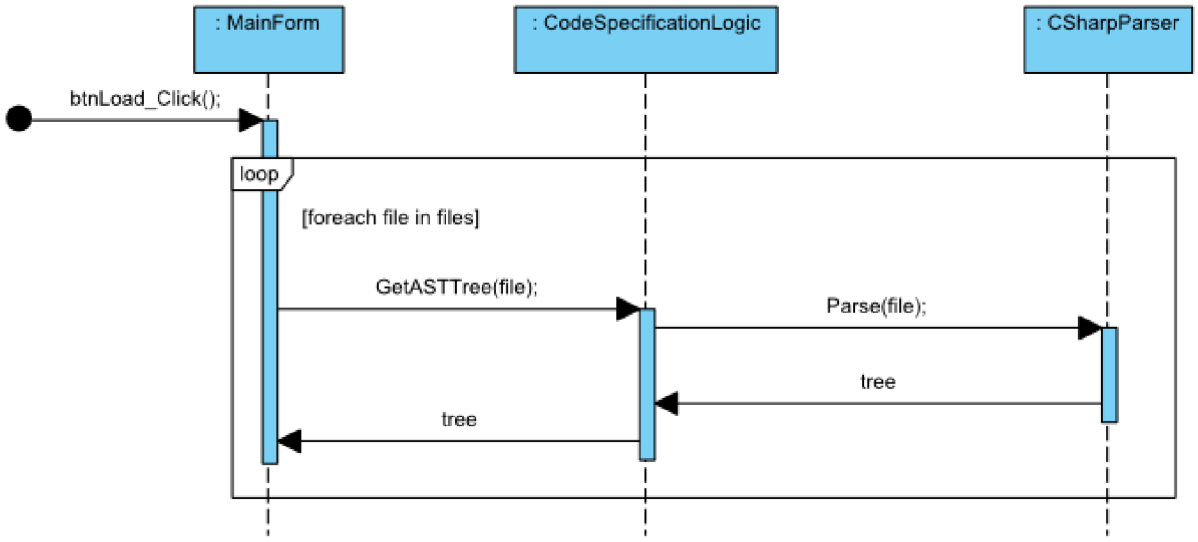
Jeden z hlavních formulářů uživatelského rozhraní je reprezentován třídou *MainForm*. Tato třída je zodpovědná za zobrazení dostupných definic návrhových vzorů uživateli a přijetí vstupu od uživatele ve smyslu výběru použitého vzoru pro generování a výběru souborů se zdrojovými kódy.

Výpis dostupných definic návrhových vzorů realizuje metoda *FillListBoxDP* třídy *MainForm*. Tato metoda načte z třídy *Settings*, která reprezentuje uživatelské nastavení aplikace, cestu k adresáři s popisem vzorů a následně zavolá metodu *GetSupportedPatterns* třídy *PatternSpecificationLogic* z komponenty *Business*. Tato metoda pomocí třídy *XMLHelper* načte postupně definice vzorů a provede jejich validaci. Korektně načtené definice reprezentuje jako seznam instancí třídy *PatternLocation*, která obsahuje nezbytné informace pro práci s nimi. Tento seznam instancí je vrácen a následně metodou *GetSupportedPatterns* zpracován. Postupně jsou vzory vypsány v komponentě ListBox. Následující diagram sekvence zobrazuje chování systému při načítání definic návrhových vzorů.



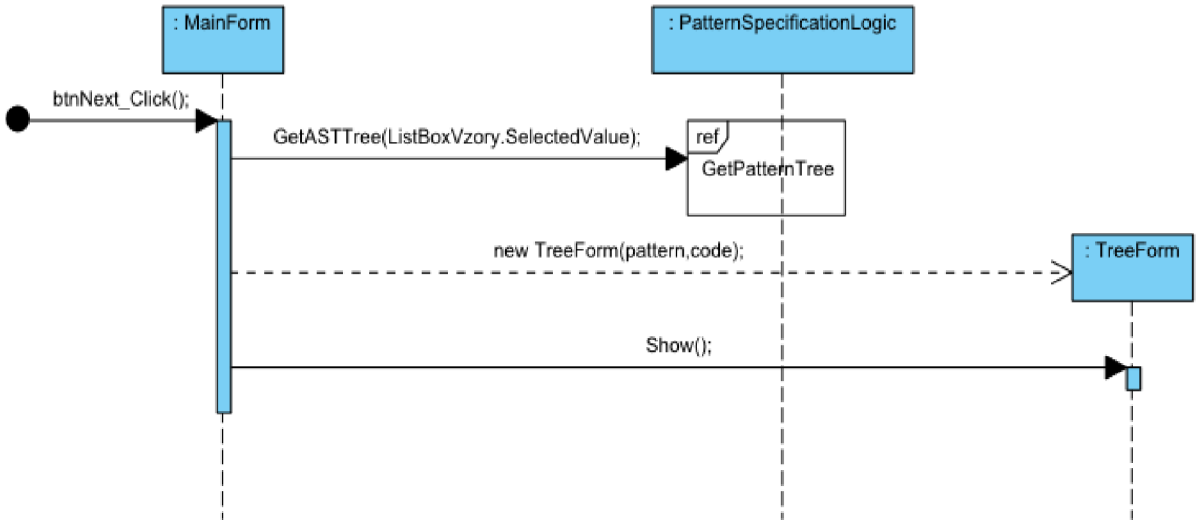
Obrázek 38: Diagram sekvence zobrazující chování při načítání dostupných definic vzorů.

Načtení souborů se zdrojovými kódy realizuje metoda *btnLoad_Click*, která je event handlerem kliknutí na tlačítko „Otevřít“. Metoda nejprve uživateli zobrazí dialogové okno pro výběr souborů se zdrojovými kódy. Následně pro každý načtený soubor zavolá metodu *GetASTTree* třídy *CodeSpecificationLogic* z komponenty *Business*, která naparsuje zdrojový kód příslušného souboru a reprezentuje ho jako AST strom, který vrací původní metodě. Ta ho uloží do seznamu korektně načtených zdrojových kódů připravených pro pozdější zpracování. Následující diagram sekvence zobrazuje chování systému při načítání zdrojových kódů.



Obrázek 39: Diagram sekvence zobrazující chování systému při načítání zdrojových kódů.

Poté, co uživatel načte zdrojové kódy a vybere příslušný vzor, který chce vyhledávat v kódu a aplikovat na něj, dochází k finálnímu načtení specifikace vzoru a její reprezentací stejnou strukturou, jako jsou zdrojové kódy. Tuto činnost realizuje metoda *btnNext_Click* třídy *MainForm*, která je event handlerem pro tlačítko „Další“. V rámci metody se načte vzor, převede na AST strom a následně se zobrazí druhé hlavní okno aplikace pro zobrazení entit vzorů a zdrojových kódů (reprezentováno třídou *TreeForm*). Následující diagram sekvence zobrazuje chování systému při načtení definice vzoru požadované uživatelem a její reprezentace uniformní datovou strukturou.



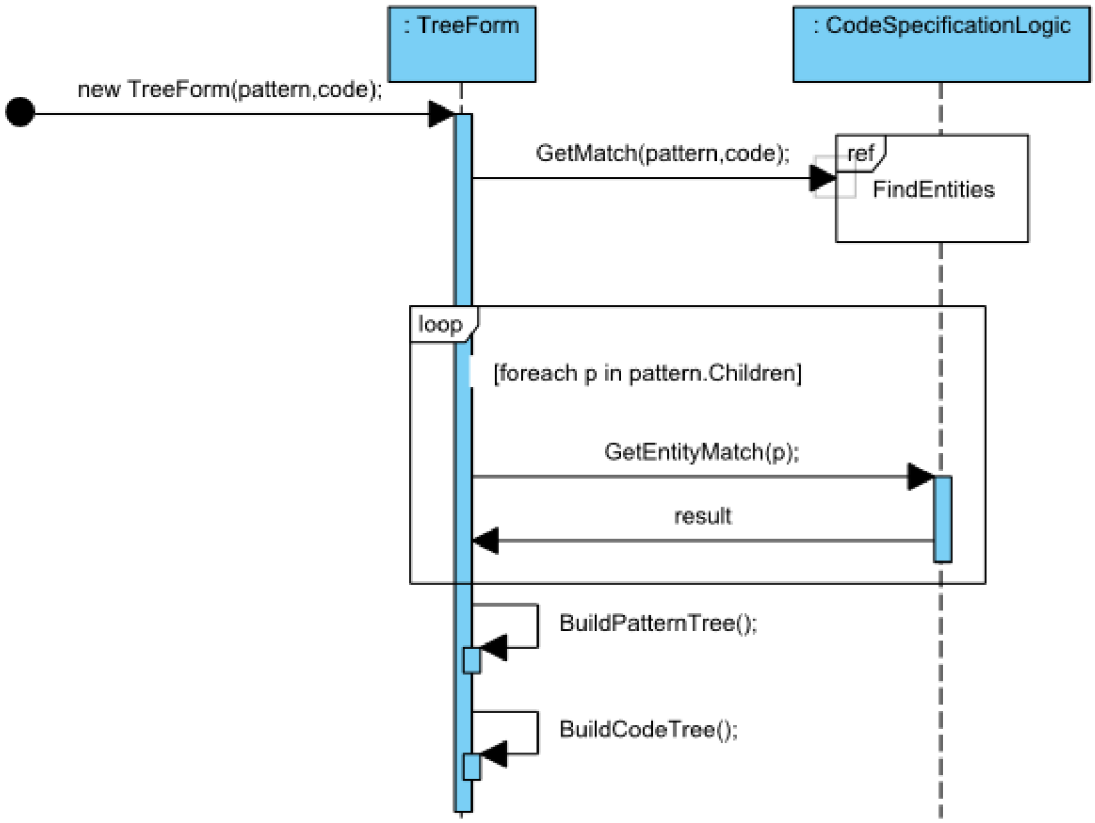
Obrázek 40: Diagram sekvence zobrazující chování systému při načtení definice vzoru.

Poté, co třída *MainForm* zavolá metodu *GetASTTree* ve třídě *PatternSpecificationLogic*, chování systému je totožné, jako je popsáno v kapitole 6.1.4.

6.2.2 Zobrazení entit vzorů a zdrojových kódů

Druhý z hlavních formulářů uživatelského rozhraní je reprezentován třídou *TreeForm*. Tato třída je zodpovědná mimo jiné za zobrazení entit načteného vzoru a k nim příslušející nalezené entity zdrojového kódu.

Okno s tímto formulářem se zobrazí až poté, co uživatel ve formuláři po spuštění vybral soubory se zdrojovým kódem a definic vzoru, který chce použít. Konstruktoru třídy *TreeForm*, který volá třída *MainForm*, jsou předány AST vzoru a načtených kódů. Následně jsou pomocí metody *GetMatch* třídy *CodeSpecificationLogic* nalezeny vhodné prvky kódu pro jednotlivé entity vzoru (více viz kapitola 6.1.5). Pak jsou pro každou entitu vzoru vybrány vhodné nalezené entity kódu. Toto spojení je vyjádřeno instancí třídy *CodeEntityUsage*, která obsahuje danou entitu kódu a příznak, zda má být použita při závěrečném generování kódu. Ke každé entitě vzoru tak existuje seznam takovýchto instancí. Nakonec jsou zavolány metody *BuildPatternTree* a *BuildCodeTree*, které zobrazí entity vzoru, respektive zdrojového kódu, pomocí komponenty WinForms TreeView. Následující diagram sekvence zobrazuje chování systému při zobrazení načtených entit vzoru a zdrojových kódů.



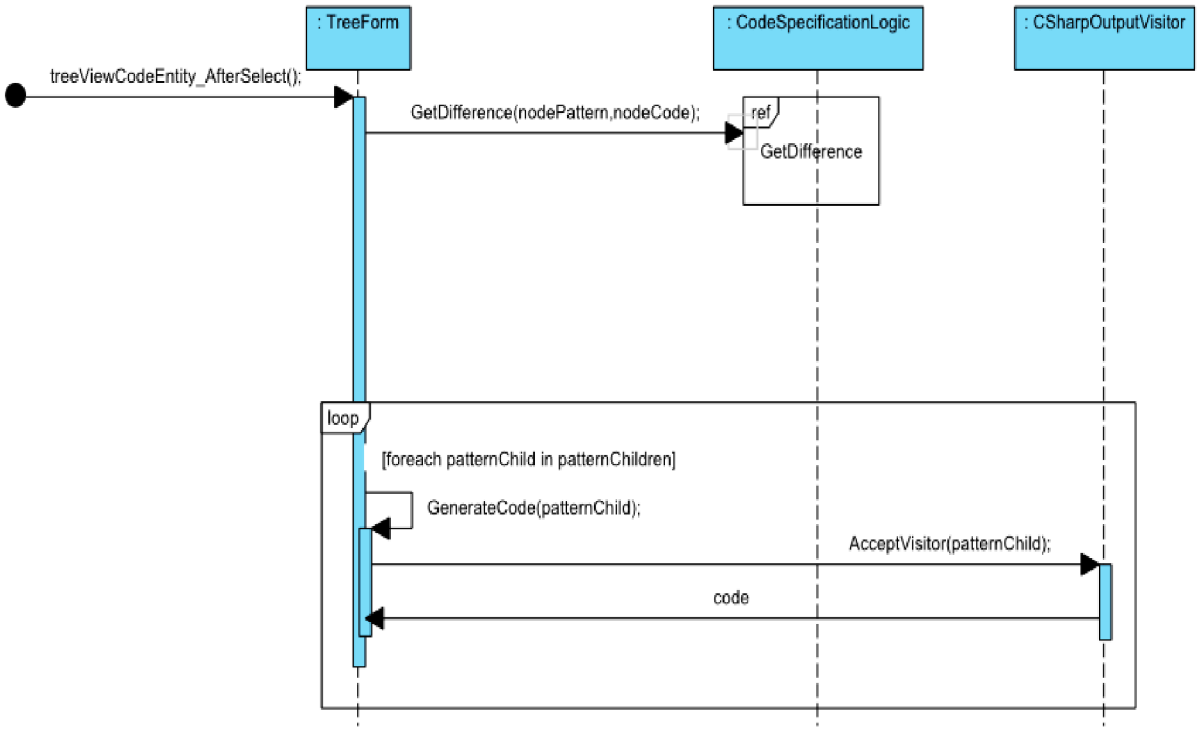
Obrázek 41: Diagram sekvence zobrazující chování systému při zobrazení entit vzoru a kódu.

Entity zdrojového kódu, které jsou zobrazeny, se vztahují vždy jen vůči aktuálně vybrané entitě vzoru. Při každé změně entity vzoru se v event handleru této události zavolá opět metoda *BuildCodeTree*, která zohlední vybranou entitu vzoru a podle toho vypíše příslušné entity kódu.

6.2.3 Zobrazení rozdílu mezi entitami

Aplikace umožňuje pro vybranou entitu kódu zobrazit, co jí chybí do aktuálně vybrané entity vzoru. Tuto činnost realizuje metoda *treeViewCodeEntity_AfterSelect* třídy *TreeForm*, která nejprve načte vybranou entitu vzoru, poté vybranou entitu kódu a zavolá metodu *GetDifference* třídy *CodeSpecificationLogic* z komponenty *Business*. Tato metoda nalezne rozdíl mezi entitami (více viz kapitola 6.1.7). Jejím výsledkem je seznam položek, které je třeba dodefinovat. Následně je zavolána pro každou položku výsledku metoda *GenerateCode* třídy *TreeForm*, která deleguje generování kódu třídě *CSharpOutPutVisitor*, jež patří do komponenty *NRefactory*. Výsledkem je poté zdrojový kód, který je zobrazen na příslušném místě formuláře. Metoda *treeViewCodeEntity_AfterSelect*, je event

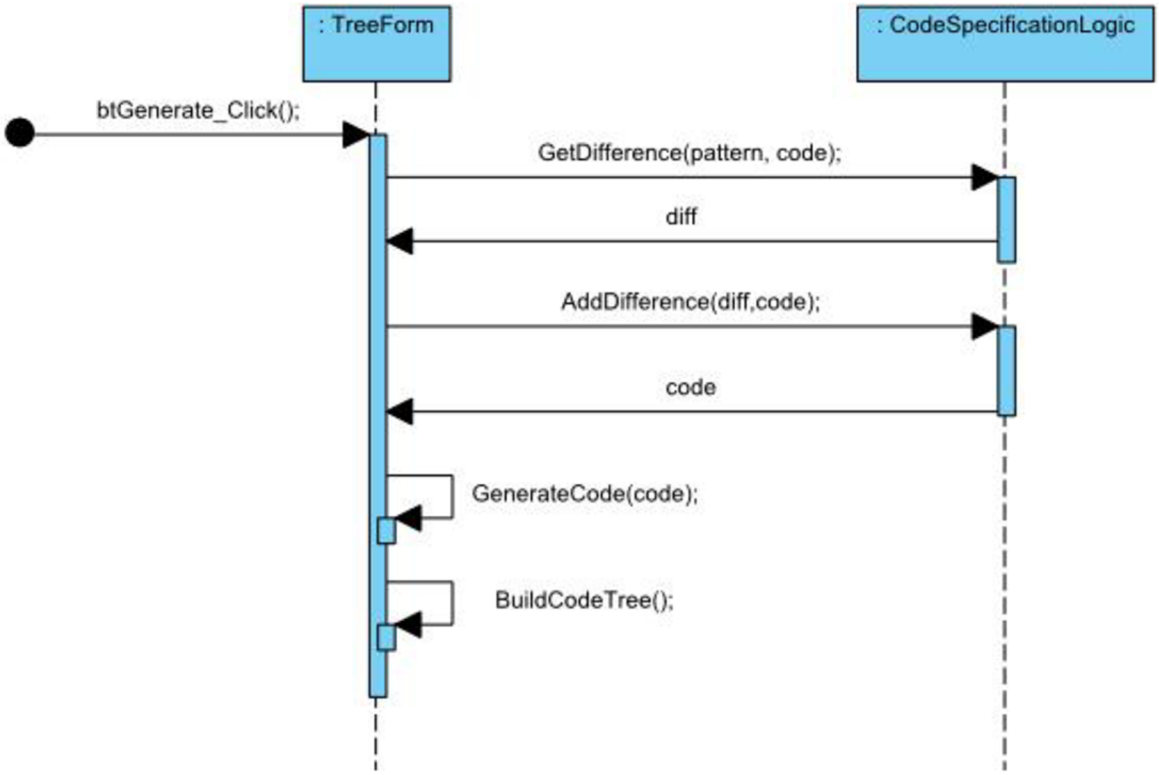
handler události výběru uzlu v komponentě TreeView, která zobrazuje entity zdrojového kódu. Následující diagram sekvence zobrazuje chování systému při zobrazení rozdílu mezi entitami.



Obrázek 42: Diagram sekvence zobrazující chování systému při zobrazení rozdílu mezi entitami.

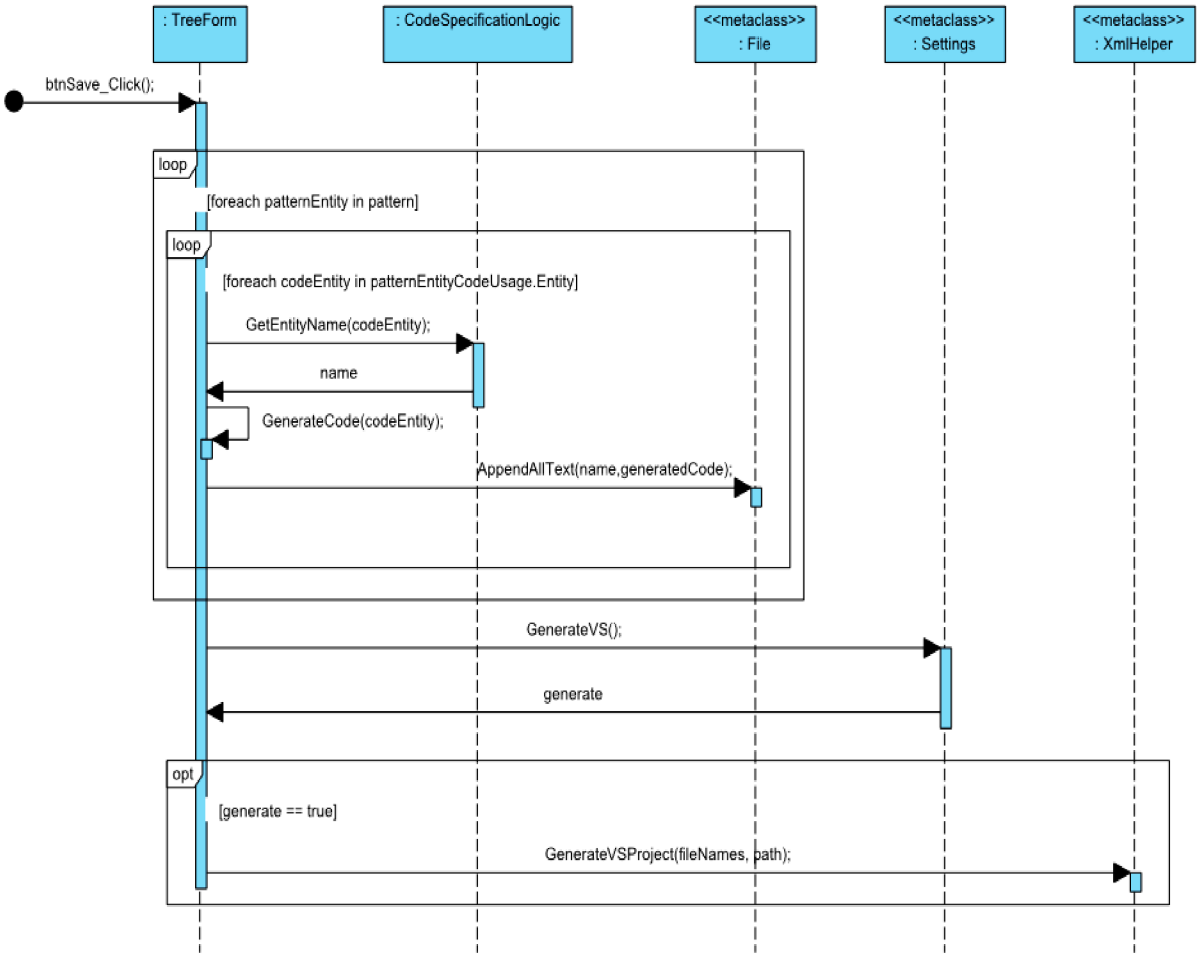
6.2.4 Generování kódu

Na úrovni uživatelského rozhraní realizuje třída *TreeForm* generování kódu pomocí dvou metod: *btnGenerate_Click* a *btnSave_Click*. První metoda je event handler na kliknutí na tlačítko „Generovat“, které uživatel použije tehdy, pokud chce do zvolené entity zdrojového kódu přidat scházející atributy nebo metody entity vzoru. Metoda nejprve načte zvolenou entitu vzoru a zvolenou entitu kódu, pak pomocí metody *GetDifference* třídy *CodeSpecificationLogic* zjistí rozdíl mezi entitami (více viz kapitola 6.1.7). Následně zavolá metodu *AddDiference* té samé třídy, která zmodifikuje (přidá jednotlivé atributy a metody) entitu kódu, kterou si aplikace uloží a zobrazí pomocí metody *BuildCodeTree*. Následující diagram sekvence zobrazuje chování systému při modifikaci entity zdrojového kódu.



Obrázek 43: Diagram sekvence zobrazující chování systému při modifikaci entity kódu.

Druhá metoda je event handler na kliknutí na tlačítko „Uložit“. Metoda nejprve zobrazí dialogové okno pro výběr nebo vytvoření adresáře pro uložení zdrojových kódů. Tato metoda vybere pro každou entitu vzoru uživatelem zvolené entity kódu (reprezentované instancí třídy *CodeEntityUsage*), zjistí jejich název pomocí metody *GetEntityName* třídy *CodeSpecificationLogic* a každou uloží do samostatného souboru pojmenovaného jménem entity do adresáře vybraného uživatelem. K tomu používá třídu *File*. Poté ještě kontaktuje třídu *Settings*, která reprezentuje uživatelské nastavení aplikace, a zjistí, zda je třeba vygenerovat projekt do Visual Studia, a případně jej vygeneruje pomocí metody *GenerateVSProject* třídy *XMLHelper*. Následující diagram sekvence zobrazuje chování systému při generování zdrojových kódů a jejich uložení do souborů.



Obrázek 44: Diagram sekvence zobrazující chování při generování kódů.

7 Testování

Aplikaci jsem úspěšně testoval několik týdnů ve vývojovém prostředí Visual Studio 2010 na platformě Windows 7 Professional. Činnost a správnost algoritmu jsem si ověřil na definicích existujících návrhových vzorů Singleton, Factory Method, Publisher-Observer a dalších. Jako zdrojové kódy jsem použil svá vytvořená testovací data, ale i náhodně vybrané reálné zdrojové kódy. Všechny testy proběhly uspokojivě a s dobrým výsledkem.

Během testování jsem přesto narazil na problém nemožnosti vyhledávání struktur v abstraktním syntaktickém stromu (AST), který byl reprezentován datovou strukturou určenou použitým parserem *NRefactory*. I když dokumentace k parseru deklarovala výše zmíněnou funkčnost, v parseru nebyla implementována. Tento problém jsem vyřešil tak, že jsem si vyhledávání podobných struktur v AST naimplementoval sám.

Dalším problémem, který se vyskytl během testování, bylo, že použitý parser *NRefactory* špatným způsobem pracoval s AST z hlediska přidávání nových uzlů do stromu. Tento problém již byl vývojářům nahlášen, nicméně nevyřešen. Vyřešil jsem to tak, že jsem upravil stávající kód parseru tak, aby vyhovoval mým potřebám.

Dále jsem během testování zjistil, že navržené rozložení uživatelského rozhraní není vhodné. Bylo nepřehledné a neumožňovalo zobrazit veškeré informace, které jsem považoval za důležité. Proto jsem formulářové okno uživatelského rozhraní pro zobrazení nalezených entit přepracoval tak, aby bylo uživatelsky přívětivě přehledné, a aplikace se tak stala snadnou pro použití.

Při testování jsem také zjistil, že mnou zvolená komponenta uživatelského rozhraní pro výpis entit návrhových vzorů a zdrojových kódů není přehledná a neumožňuje výpis dodatečných informací o entitách či možnost uchovat informaci o tom, které entity uživatel zvolil pro generování. Navrhl jsem proto jiné řešení výpisu entit vzoru a kódů za použití komponenty TreeView, které plně vyhovuje požadavkům na aplikaci.

Většina problémů, které jsem odhalil během testování, byla způsobena nedokonalou nebo chybějící funkčností použitého parseru, i když byla proklamována. Tyto problémy se mi podařilo úspěšně vyřešit. Zároveň s tím jsem algoritmus implementoval tak, aby byl co nejméně závislý na použitém parseru a případná změna parseru byla co nejjednodušší.

8 Závěr

Tato práce se zabývá problematikou zápisu návrhových vzorů se zaměřením zejména na jejich jednoznačnost a využitelnost při generování kódů aplikací a zavádění vzorů do existujících kódů. Cílem je vytvoření aplikace, která načte definici vzoru, ve zdrojových kódech vyhledá vhodné struktury pro aplikaci vzoru, nabídne je uživateli a následně provede refaktorizaci kódu takovým způsobem, aby realizoval daný vzor.

8.1 Vlastní přínos

Během práce jsem se seznámil s návrhovými vzory a detailně jsem prostudoval problematiku definice návrhových vzorů z hlediska využitelnosti při generování kódů. Dále jsem se zabýval možnostmi generování zdrojových kódů pro jazyk C# a detailně jsem prostudoval možnosti formálních specifikací návrhových vzorů a nejnovější trendy výzkumu v této oblasti.

Následně jsem navrhl algoritmus pro generování zdrojových kódů pomocí návrhových vzorů. Zaměřil jsem se zejména na zavádění vzorů do již existujících zdrojových kódů pomocí refaktorizace jednotlivých tříd takovým způsobem, aby realizovaly daný vzor. Navržený algoritmus jsem implementoval a jeho funkčnost jsem si ověřil formou prototypové aplikace. Výslednou aplikaci jsem úspěšně otestoval na běžně používaných návrhových vzorech a reálných zdrojových kódech.

Navržený a implementovaný algoritmus, který je hlavním výsledkem mojí práce, položil základy zavádění vzorů do zdrojových kódů, a díky tomu se tak může stát například hlavní logikou pluginů do vývojových prostředí anebo základem pro detekce vzorů ve zdrojových kódech.

Výsledky této práce jsem také prezentoval formou odborného článku ve sborníku EEICT a zároveň jsem se účastnil i soutěže.

8.2 Možnosti budoucího rozšíření

Možnosti dalšího vývoje projektu spatřuji ve dvou směrech: rozšíření stávající funkčnosti algoritmu či zpřesnění stávající a použití algoritmu jako základu pro jiné nástroje. Algoritmus by například mohl být dále rozšířen o podporu getterů a setterů jednotlivých tříd, navzájem do sebe zanořených tříd nebo namespaců, vícenásobné dědičnosti, realizace více rozhraní anebo rozšířením hodnotících kritérií o další parametry.

Další možností rozšíření je použití navrženého algoritmu jako základu pro jiné nástroje. Jednou z možností je využít ho jako základu logiky pluginu do vývojových prostředí, který by tak zajistil vývojářům pohodlné zpracování vzorů na úrovni zdrojových kódů. S tím by souviselo i vytvoření veřejného katalogu předdefinovaných popisů návrhových vzorů ve formátu požadovaným algoritmem, který by byl veřejně dostupný. Jeho smysl použití by byl podobný jako v případě registrů služeb u SOA (Service Oriented Architecture). Tím by se stal jednoduše dostupným a použitelným i pro jiné aplikace.

Algoritmus by ale také šlo v budoucnu využít jako základu pro systém detekce návrhových vzorů ve zdrojovém kódu, díky čemuž by bylo možné vylepšit automatizovanou validaci zdrojových kódů rozsáhlých systémů, nebo ho využít jako základu pro systém automatizované verifikace návrhových vzorů.

Literatura

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, R.*: Design Patterns - Elements of Reusable Object Oriented Software*. NewYork: Addison-Wesley, 2004. ISBN 02-016-3361-2

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, R., překlad Makovec, P.: *Návrh programů pomocí vzorů*. Praha: Grada, 2003. ISBN 80-247-0302-5

[3] Bishop, J., překlad Koutný, J.: *C# - návrhové vzory*. Brno: Zoner Press, 2010. ISBN 978-80-7413-076-2

[4] Pecinovský, R.: *Návrhové vzory*. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4

[5] Lairman, C.: *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. New Jersey: Prentice Hall, 2006. ISBN 0-13-148906-2

[6] Taibi, T., Ngo, D.C.L.: *Formal Specification of Design Patterns – A Balanced Approach.* Journal of Object Technology [online]. 2003 [cit. 2011-11-05]. Dostupné na URL: http://www.jot.fm/issues/issue_2003_07/article4

[7] Bayley, I.; Hong Z.: *Formalising Design Patterns in Predicate Logic*. Software Engineering and Formal Methods: Fifth IEEE International Conference [online]. Oxford, 2007 [cit. 2011-11-05]. ISBN: 978-0-7695-2884-7. Dostupné na URL: http://ieeexplore.ieee.org/search/freesrchabstract.jsp?tp=&arnumber=4343921&queryText%3Dformalising+design+patterns+in+predicate+logic%26openedRefinements%3D*%26filter%3DAND%28NOT%284283010803%29%29%26searchField%3DSearch+All

[8] Taibi, T.: *Design Patterns Formalization Techniques: Formal specification and verification of design patterns* [online]. London, 2007 [cit. 2011-11-05]. ISBN: 978-1-59904-221-3. Dostupné na URL: http://www.google.com/books?id=lQH6Ejk9OwUC&lpg=PA1&hl=cs&pg=PA1#v=onepage&q&f=false

[9] Bayley, I., Hong, Z.: *Specifying Behavioural Features of Design Patterns in First Order Logic*. 32nd Annual IEEE International [online]. 2008 [cit. 2011-11-05]. ISBN: 978-0-7695-3262-2. Dostupné na URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4591558&tag=1

[10] Cinnéide, M.O., Nixon, P.: *Automated Software Evolution Towards Design Patterns*. Proceedings of the 4th International Workshop on Principles of Software Evolution [online]. New York: ACM New York, 2001 [cit. 2011-11-05]. ISBN: 1-58113-508-4. Dostupné na URL: http://dl.acm.org/citation.cfm?id=602499&CFID=54190592&CFTOKEN=56050054&preflayout=tabs

[11] Blewitt, A., Bundy, A., Stark, I.: *Automatic Verification of Design Patterns in Java.* Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering [online]. New York: ACM New York, 2005 [cit. 2011-05-11]. ISBN: 1-58113-993-4. Dostupné na URL: http://dl.acm.org/citation.cfm?id=1101908.1101943&coll=DL&dl=ACM&CFID=54190592&CFTOKEN=56050054

[12] Kačmář, D.: *Programujeme .NET aplikace.* Praha: Computer Press, 2001. ISBN 80-7226-569-5

[13] Sharp, J.: *Microsoft Visual C# 2005 krok za krokem.* Praha: Computer Press, 2006. ISBN 80-251-1156-3

[14] Microsoft: *Microsoft Developer Network.* Oficiální dokumentace pro .NET [online]. 2011 [cit. 2011-11-05]. Dostupné na URL: http://msdn.microsoft.com

[15] Hanák, F.: Systém pro získávání dat o leteckých trasách z webových portálů, bakalářská práce, Brno, FIT VUT v Brně, 2009

[16] Křena, B., Kočí, R.: *Úvod do softwarového inženýrství, studijní opora*. 2006

[17] Zendulka, J., Bartík, V., Květoňová, Š.: *Analýza a návrh informačních systémů, studijní opora.* 2006

[18] Šlapal, J.: *Matematická logika: Úvod do predikátového počtu 1. řádu, studijní opora.* 2010

[19] Lamport, L.: *Introduction to TLA.* Technical note [online]. 1994 [cit. 2011-11-05], Dostupné na URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1994-001.html

[20] Štepánek, P.: *Predikátová logika, skripta.* Matematicko-fyzikální fakulta UK [online]. Praha, 2010 [cit. 2011-11-05]. Dostupné na URL: http://kti.mff.cuni.cz/index.php?select=teaching&section=sources&lang=czech

[21] Hanák, F. Code Generation Using Design Patterns. In: *Proceedings of the 18th Conference STUDENT EEICT 2012: Volume 2.* Brno: LITERA, 2012. ISBN 978-80-214-4461-4. Dostupné z: http://www.feec.vutbr.cz/EEICT/

Seznam obrázků

Seznam použitých zkratek

AST	Abstract Syntax Tree
BPSL	Balanced Pattern Specification Language
CodeDom	Code Document Object Model
GEBNF	Graphic Extended Bacus Naur Form
GPL	General Public License
GUI	Graphic User Interface
LGPL	Lesser General Public License
OOP	Object Oriented Programming
SOA	Service Oriented Architecture
UML	Unified Modelling Language
XML	Extensible Markup Language
XSD	XML Schema Definition

Seznam příloh

Příloha 1

Obsah DVD

V následující tabulce je uveden obsah přiloženého DVD.

dp.pdf	Elektronická verze této práce ve formátu PDF.
readme.txt	Readme k obsahu DVD.
manual.pdf	Uživatelský manuál k aplikaci.
./dokumentace	Složka s programovou dokumentací.
./kod	Složka se zdrojovými kódy.
./uml	Složka s UML diagramy týkající se návrhu a implementace.
./vzory	Složka s předdefinovanými návrhovými vzory.

Tabulka 5: Obsah přiloženého DVD.

Příloha 2

Příklady formálních specifikací návrhových vzorů

Prolog

```
polymorph(Interface,Imp,Binding,ConcreteImpSet,ImpOperation,Operation):-
assert(abstractClass(Imp)),
assert(method(Imp,public,ImpOperation),
forAll(member(ConcreteImp,ConcreteImpSet),assert(inherit(Imp,ConcreteImp))),
forAll(member(ConcreteImp,ConcreteImpSet),assert(class(ConcreteImp))),
forAll(member(ConcreteImp,ConcreteImpSet),assert(method(ConcreteImp,
      public,ImpOperation))),
assert(abstractClass(Interface)),
assert(attribute(Interface,private,Binding,Imp)),
assert(method(Interface,public,Operation)),
assert(invoke(Interface,Operation,Binding,ImpOperation)).

extend_polymorph(Imp,NewConcreteImpSet,ImpOperation):-
forAll(member(ConcreteImp,NewConcreteImpSet),assert(inherit(Imp,ConcreteImp))),
forAll(member(ConcreteImp,NewConcreteImpSet),assert(class(ConcreteImp))),
forAll(member(ConcreteImp,NewConcreteImpSet),assert(method(ConcreteImp,
      public,ImpOperation))).

retract_polymorph(Imp,OldConcreteImpSet,ImpOperation):-
forAll(member(ConcreteImp,OldConcreteImpSet),retract(inherit(Imp,Concrete
      Imp))),
forAll(member(ConcreteImp,OldConcreteImpSet),retract(class(ConcreteImp))),
forAll(member(ConcreteImp,OldConcreteImpSet),retract(method(ConcreteImp,pu
blic,ImpOperation))).
```

Kód 8: Příklad zápisu polymorfismu v jazyce Prolog.

```
bridge(Abstraction,Implementor,Imp,RefinedAbstractionSet,ConcreteImplementorSet,
       ImpOperation,Operation) :-
polymorph(Abstraction,Implementor,Imp,ConcreteImplementorSet,ImpOperation,
       Operation),
forAll(member(RefinedAbstraction,RefinedAbstractionSet),assert(inherit(
       Abstraction,RefinedAbstraction))),
forAll(member(RefinedAbstraction,RefinedAbstractionSet),assert(class(
       RefinedAbstraction))).

extend_bridge_abstract(Abstraction,NewRefinedAbstraction):-
assert(inherit(Abstraction,NewRefinedAbstraction)),
assert(class(NewRefinedAbstraction)).

extend_bridge_imp(Implementor,NewConcreteImplementor,ImpOperation):-
extend_polymorph(Implementor,NewConcreteImplementor,ImpOperation).

retract_bridge_abstract(Abstraction,OldRefinedAbstraction):-
retract(inherit(Abstraction,OldRefinedAbstraction)),
retract(class(OldRefinedAbstraction)).

retract_bridge_imp(Implementor, OldConcreteImplementor, ImpOperation):-
retract_polymorph(Implementor, OldConcreteImplementor, ImpOperation).
```

Kód 9: Příklad specifikace návrhového vzoru Bridge v jazyce Prolog.

Příloha 3

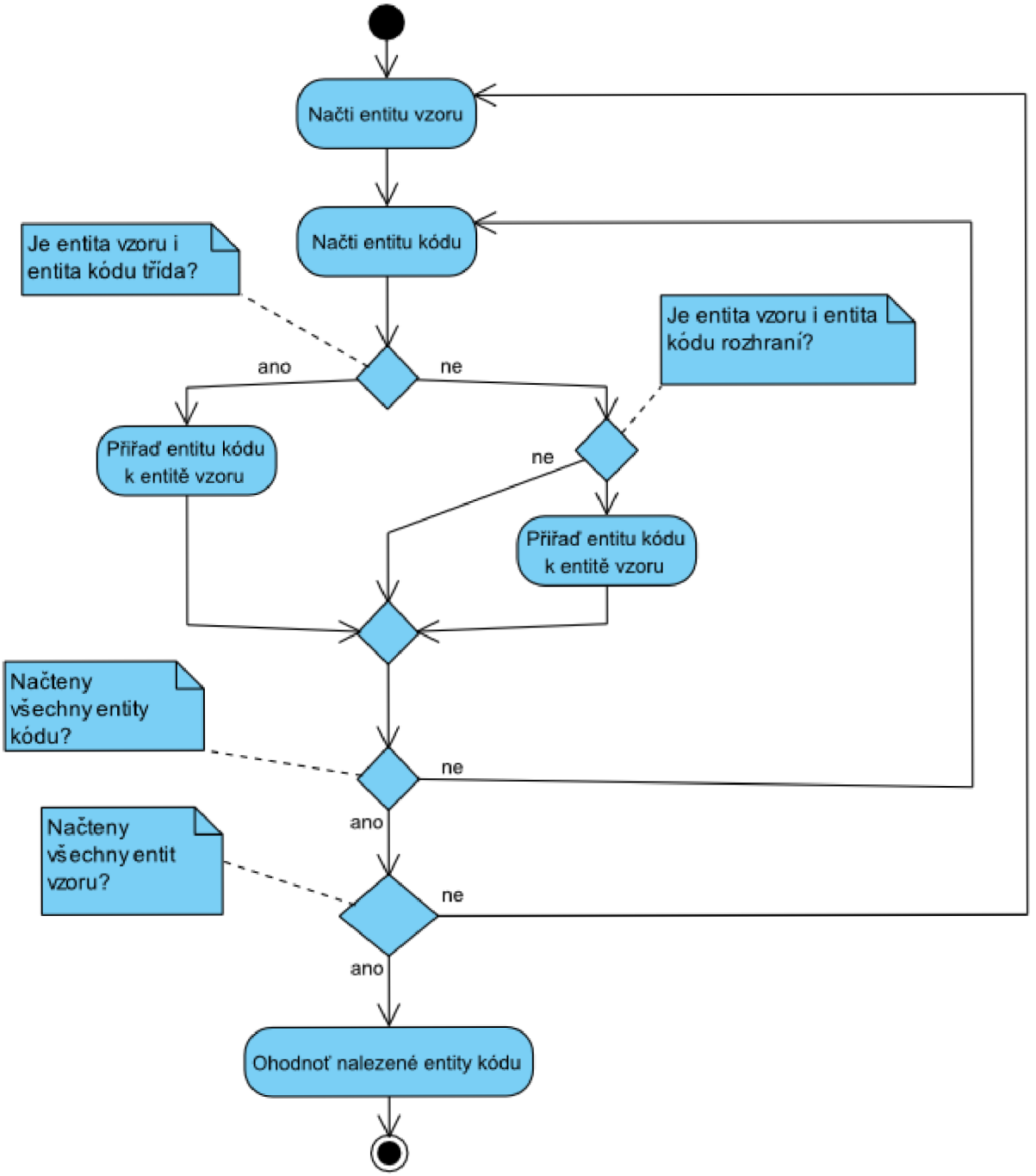
Příklady generování kódu v CodeDom

```
using System;
using System.Text;
using System.CodeDom;
using System.CodeDom.Compiler;

public class CodeGenerator {
 static void Main(string[] args) {

   CodeTypeDeclaration class = new CodeTypeDeclaration("Person");
   class.IsClass = true;
   class.TypeAttributes = TypeAttributes.Public;

   CodeMemberField field1 = new CodeMemberField("System.String",  "firstName");
   field1.Attributes = MemberAttributes.Private;
   CodeMemberField field2 = new CodeMemberField("System.String",  "lastName");
   field2.Attributes = MemberAttributes.Private;
   class.Members.Add(field1);
   class.Members.Add(field2);

   CodeParameterDeclarationExpression param1 = new
       CodeParameterDeclarationExpression("System.String","fName"));
   CodeParameterDeclarationExpression param2 = new
       CodeParameterDeclarationExpression("System.String","lName"));

   CodeFieldReferenceExpression ref1 =
    new CodeFieldReferenceExpression(new CodeThisReferenceExpression(),
    "firstName");
   CodeFieldReferenceExpression ref2 =
    new CodeFieldReferenceExpression(new CodeThisReferenceExpression(),
    "lastName");

   CodeAssignStatement as1 = new CodeAssignStatement(ref1,param1);
   CodeAssignStatement as2 = new CodeAssignStatement(ref2,param2);

   CodeConstructor cc = new CodeConstructor();
   cc.Attributes = MemberAttributes.Public;
   cc.Parameters.Add(param1);
   cc.Parameters.Add(param2);
   cc.Statements.Add(as1);
   cc.Statements.Add(as2);
   class.Members.Add(cc);

   TextWriter tw = new StreamWriter("code.cs");
   ICodeGenerator generator = new CSharpCodeProvider.CreateGenerator();
   generator.GenerateCodeFromType(class,tw,null);
   tw.Close();
  }
}
```

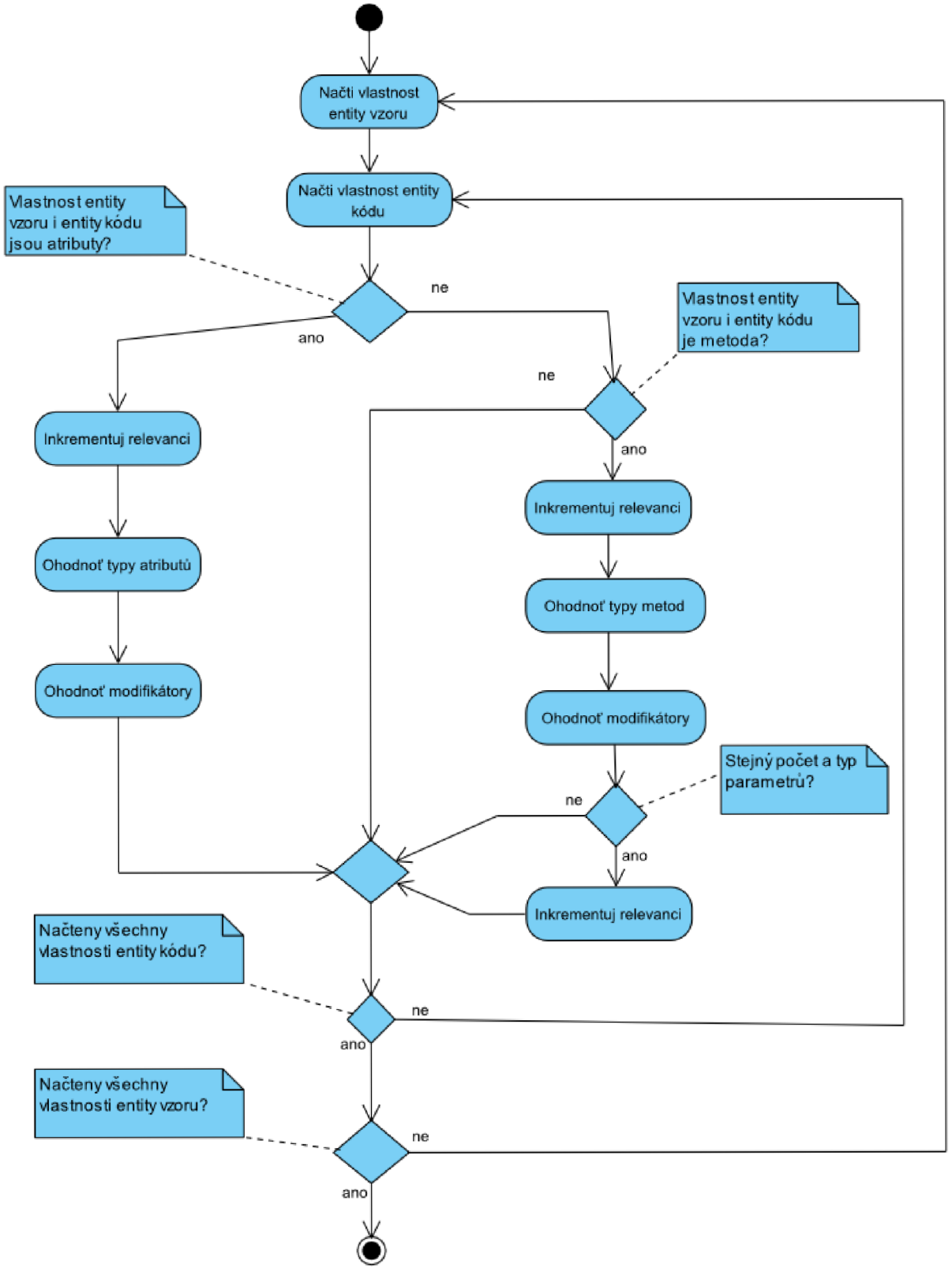
Kód 10: Příklad zápisu logické struktury zdrojového kódu v C# pomocí CodeDom.

Příloha 4

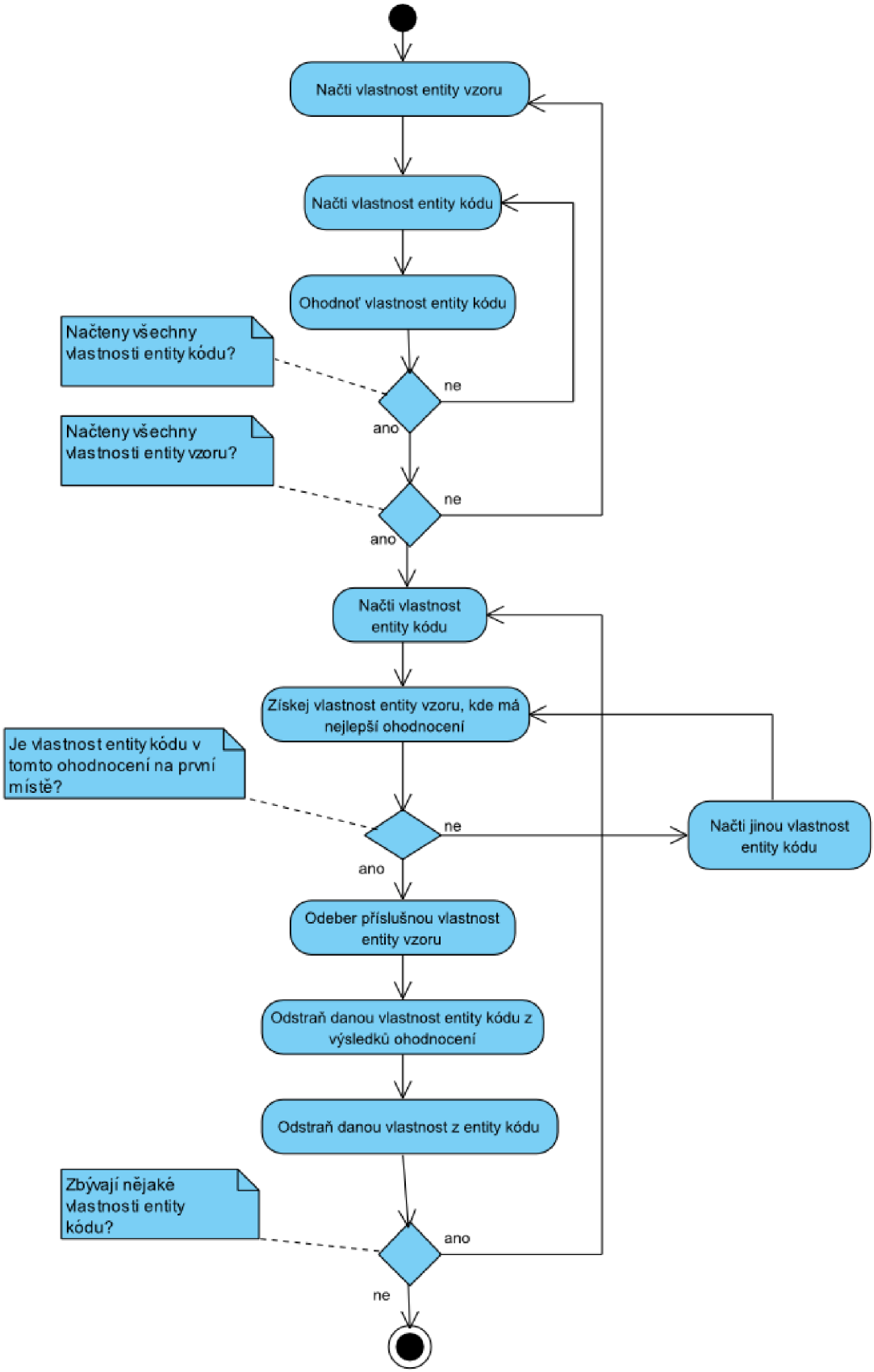
Diagramy aktivity



Obrázek 45: Diagram aktivity algoritmu nalezení vhodných entit.



Obrázek 46: Diagram aktivity algoritmu ohodnocení nalezených entit.



Obrázek 47: Diagram aktivity algoritmu nalezení rozdílu mezi entitami.