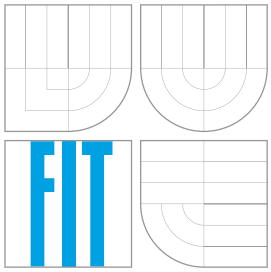
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ZOBRAZOVÁNÍ REFERENCÍ .NET KNIHOVEN

DISPLAYING .NET LIBRARIES REFERENCES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

KAREL PRAJS

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KŘIVÁNEK

BRNO 2010

Abstrakt

Bakalářská práce řeší analýzu a grafické zobrazování referencí knihoven v .NET projektech. První část dokumentu je věnována rozboru teoretických možností s důrazem na použití reflexe pro analýzu referencí a model magnetických pružin pro generování grafu. Druhá část je pak věnována praktické implementaci těchto metod, jejich propojení a začlenění do grafického rozhraní.

Abstract

The bachelor's thesis deals with an analyse of references of libraries and its graphical representation. First part of document introduces some options of solution with focus on reflection (to analyse references) and magnetic-spring model (to generate graph). Second part deals with implementation of these methods, its connection and integration into graphical user interface.

Klíčová slova

zobrazování referencí, Microsoft .NET, vykreslování grafu, hodnocení grafu, model magnetických pružin, WPF

Keywords

references displaying, Microsoft.NET, graph drawing, graph rating, Magnetic-Spring model, WPF

Citace

Karel Prajs: Zobrazování referencí .NET knihoven, bakalářská práce, Brno, FIT VUT v Brně, 2010

Zobrazování referencí .NET knihoven

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana ing. Jana Křivánka, a že jsem uvedl všechny informační zdroje, ze kterých jsem čerpal.

.....
Karel Prajs
17. května 2010

Poděkování

Děkuji panu ing. Janu Křivánkovi za ochotnou a přínosnou spolupráci.

Obsah

Kapitola 1

Úvod

1.1 O tématu

Při programování rozsáhlejší aplikace v prostředí Microsoft .NET dochází často k jejímu členění do několika vzájemně provázaných knihoven. Tyto vazby se s rostoucím projektem stávají velice nepřehledné a udělat si představu o provázanosti takového řešení může být bez grafického znázornění velice náročné. Z toho důvodu vznikají různé analyzační nástroje, které pomáhají vývojářům získat lepší přehled o technologické úrovni a struktuře jejich produktů. Takové informace mají vysokou hodnotu, ať už z hlediska optimalizace, znovupoužitelnosti kódu nebo prosté tvorby technické dokumentace.

1.2 Existující produkty

V současné době existuje na trhu několik desítek analyzačních nástrojů určených pro prostředí Microsoft .NET, poskytujících široký rozsah více či méně užitečných analýz a metrik. Tyto nástroje můžeme dle jejich zaměření rozdělit do třech základních skupin.

- metrické nástroje (poskytují statistické údaje o projektu)
- profilovací nástroje (poskytují statistické údaje o průběhu spuštěného programu)
- ostatní nástroje (dekompilery, čtečky mezikódů apod.)

V tomto dokumentu se budeme zabývat kategorií první, protože do ní tématicky zapadá tato práce.

1.2.1 Metrické nástroje

Metrické nástroje slouží k analýze údajů o statické části programu, o zdrojových kódech (narozdíl od profilovacích nástrojů, které poskytují informace o běhu programu). Patrně nejlepším a nejznámějším komerčním produktem je NDepend[1] firmy SMACCHIA.COM S.A.R.L, který poskytuje velké množství kvalitně zpracovaných statistik a grafů, mimo jiné právě analýzu referencí knihoven. Přestože je tento program poměrně drahý, lze ho volně používat pro akademické a vzdělávací účely, čehož jsem pro srovnání rád využil a výsledky uvedl v dalších částech dokumentu. Další programy, ať už volně šiřitelné nebo komerční, poskytují vesměs standardní funkce, které si šikovný programátor napíše sám za několik hodin s jednoduchým použitím reflexe.

[1] http://www.ndepend.com

Kapitola 2

Metody analýzy referencí

Analýzu referencí lze provést dvěma způsoby. Dobrým a špatným. Protože nemá valný smysl rozvádět špatný postup, probereme ho hned na začátku a dál se jím již nebudeme zabývat.

2.1 Analýza pomocí parsování .csproj souboru

Projektové soubory obsahují seznam referencovaných projektů i knihoven, můžeme z nich tedy získat představu o provázanosti projektů i celých řešení, má to ovšem několik zásadních nevýhod:

Potřeba zdrojových kódů je zcela zásadní nedostatek pro tu část uživatelů, která potřebuje získat informace o knihovně, k níž nemá zdrojové kódy (např. zakoupená komerční knihovna).

Množství informací uvedených v projektových souborech je omezené pouze na název knihovny.

Oddělení metadat od binární podoby nám bere záruku, že skutečně analyzujeme to, co používáme. To je potřeba mít neustále na paměti a analyzovat pouze čerstvě přeložené knihovny.

Implementační náročnost je vyšší, než s použitím reflexe.

Další drobnosti, jako například nižší rychlost zpracování už jen dovršují množství důvodů, proč tento způsob nepoužívat.

2.2 Analýza pomocí reflexe

Analýza pomocí reflexe eliminuje všechny zmíněné nedostatky parsování projektových souborů. Pracuje s binární podobou knihoven, a tedy nepotřebuje zdrojové kódy. S tím zároveň odpadá problém s potenciálním nesouladem mezi zdrojovými kódy a jejich binární podobou. Spolu s množstvím poskytovaných informací a snadnou implementovatelností představuje ideální metodu pro jakoukoliv analýzu knihoven, postavených pro platformu .NET.

Následující kapitoly jsou věnovány teoretickému úvodu do reflexe a dalších, nezbytně potřebných technologiích.

2.2.1 Reflexe

Reflexe je obecný termín popisující schopnost zkoumat části kódu a manipulovat s nimi za běhu programu. Ke své činnosti využívá metadata asociovaná s programovými elementy, která jsou tvořena při kompilaci a jsou součástí tzv. *assembly*. Assembly je základní stavební jednotka .NET projektů. Přestože i assembly používá koncovky DLL či EXE, od klasické knihovny se zásadně liší. Na rozdíl od prosté knihovny, assembly je kontejner zapouzdřující jeden či více souborů (resources, moduly apod.) a informace o nich [1]. Pro naše potřeby jsou zajímavé právě tyto informace. Poskytují detailní popis obsažených datových typů, jejich metod, vlastností, událostí i proměnných – ať už veřejných či privátních, objektových či statických. Možnost zpřístupnění a modifikace dokonce i privátních členů příkladně demonstruje sílu (a do jisté míry i nebezpečí) použití reflexe.

Reflexe je součástí standardního programovacího vybavení .NET frameworku. Nejedná se o žádný externí nástroj, chceme-li ji tedy využít pro analýzu existujících knihoven, je nutné napsat obslužný program. Protože primární použití reflexe je v práci nad běžícími částmi kódu, a tedy pracuje s datovými typy, není možné přistupovat k metadatům jako ke statickým datům (resp. číst je ze souboru tak, jak to známe například z práce s textovými soubory). Pro získání přístupu je nutné takovou část kódu (resp. celou assembly, do které patří) nahrát do *aplikační domény* programu a reflexi spouštět přímo nad získaným datovým typem. Co je aplikační doména a jak do ní nahrát assembly si vysvětlíme později. V tuto chvíli je důležitější zmínit jednu nepříjemnost, která z této skutečnosti pramení. Totiž assembly nahranou do aplikační domény nelze z této odstranit, to má pro naše potřeby nepříjemný důsledek – v jednom běhu programu bychom mohli analyzovat pouze jedinou verzi assembly.

2.2.2 Aplikační doména

Technologie předcházející .NET používaly k oddělení běžících aplikací procesy. Každá aplikace běžela ve vlastním procesu a k jiným procesům neměla přímý přístup. Tím bylo zajištěno, že jedna aplikace nemůže způsobit pád jiné aplikace (přímým zápisem do její části paměti). Technologie .NET tuto technologii rozšiřuje o tzv. aplikační domény, což jsou ohraničené části paměti, ne nepodobné procesům. Rozdíl je v tom, že více aplikačních domén může běžet v jediném procesu, aniž by se vzájemně ohrožovaly či si překážely. O to se za nás stará běhové prostředí .NET frameworku [1]. Běhové prostředí také zaručuje, že žádné prostředky nebudou uvolněny, dokud je kterákoliv běžící část kódu referencuje. S tím bohužel souvisí fakt, že jednou nahraná assembly nelze z domény odstranit. Proč tomu tak je zdůvodňuje Jason Zander (víceprezident vývojového týmu Visual Studia) mimo jiné tím, že sledování referencí vedoucích na assembly je výpočetně příliš drahé, proto se takto sledují pouze celé domény [6].

Aby však byl program pro analýzu použitelný, musí uživateli umožňovat testování více verzí jedné knihovny, aniž by musel neustále restartovat program. Vhodným řešením takového problému je vytvoření nové aplikační domény, ve které můžeme assembly analyzovat a následně ji celou odstranit, včetně všech do ní nahraných assembly. Tady ovšem narážíme na další překážku, která vychází ze samotného principu (a důvodu existence) aplikačních domén, a to fakt, že aplikační domény jsou izolované celky a nemohou spolu přímo komunikovat. Výsledky analýzy proběhlé v jedné doméně tedy nemůžeme přímo vrátit programu, běžícímu v druhé doméně. Řešením je technologie *.NET Remoting*.

2.2.3 .NET Remoting

Technologie .NET Remoting umožňuje komunikaci mezi odlišnými aplikačními doménami, existujícími nejen v rámci jednoho procesu, ale i v různých procesech či systémech. Poskytuje množství typů přenosů — síťových i lokálních a vykazuje velkou míru přizpůsobitelnosti. My však využijeme pouze malou část funkčnosti, kterou nabízí. Zajímat nás bude jediná třída — *MarshalByRefObject*. Odvozením libovolné třídy od třídy MarshalByRefObject vytvoří běhové prostředí referenci na objekt této třídy, kterou předá druhé doméně. Přes tuto referenci, fungující na principu proxy, můžou obě domény komunikovat. Další možností je označit třídu jako serializovatelnou. V takovém případě bude při každém pokusu o komunikaci celý objekt serializován, odeslán druhé doméně, kde bude deserializován a použit. Takový přístup je ale zjevně dražší a tudíž méně vhodný.

2.2.4 Využití reflexe pro analýzu referencí

Kombinací zmíněných třech technologií můžeme vytvořit silný nástroj pro analýzu (nejen) referencí knihoven. Pro samotnou analýzu si vystačíme s jednou třídou *Assembly* ze jmenného prostoru *System.Reflection* a dvěma jejími funkcemi:

GetReferencedAssemblies() vrací pole datových typů *AssemblyName[]*, reprezentující všechny referencované assembly

Load(AssemblyName) statická funkce, která podle parametru AssemblyName vytvoří objekt Assembly

Pomocí těchto dvou funkcí, jednoho cyklu a rekurze můžeme analyzovat celý strom referencovaných knihoven. Abychom mohli v jednom běhu programu analyzovat více verzi jedné assembly, využijeme možnosti provést analýzu v jiné aplikační doméně. K tomu nám budou stačit pouhé tři funkce třídy *AppDomain*:

CreateDomain() statická funkce, vytvoří novou aplikační doménu

CreateInstanceAndUnwrap(string, string) vytvoří v doméně nový objekt

Unload(AppDomain) statická funkce, odstraní doménu a s ní i všechny do ní načtené assembly

Shrneme-li dosud získané poznatky, můžeme celý průběh analýzy popsat v několika málo bodech:

- rozdělení hlavního programu a analýzy do dvou tříd
- odvození analyzační třídy od třídy MarshalByRefObject
- vytvoření dočasné domény s objektem analyzační třídy, předání zdrojových dat (cestu k analyzované assembly)
- spuštění analýzy, předání výsledků zpět hlavnímu programu
- odstranění dočasné domény

Kapitola 3

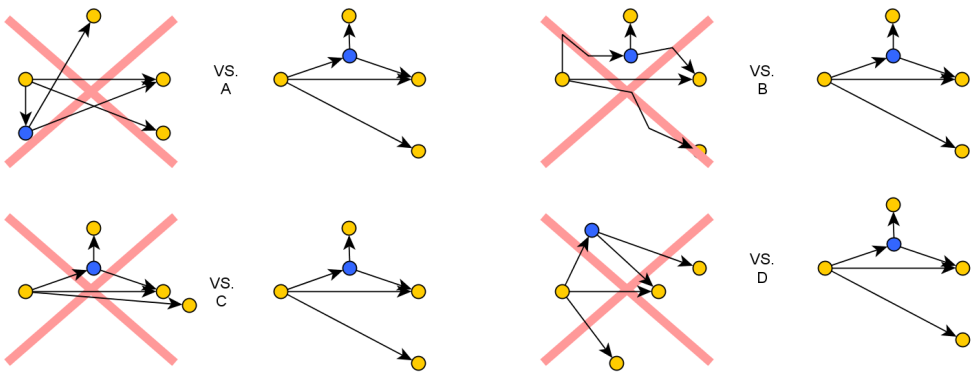
Metody pozicování grafu

Algoritmy pro pozicování grafu mají za úkol najít takovou kombinaci poloh uzlů, aby graf vyhovoval co nejvíce hodnotícím kritériím. Důsledkem dobrého hodnocení je pak jednoduchý, čitelný graf, ve kterém se člověk snadno zorientuje.

3.1 Metody hodnocení grafu

Pro hodnocení kvality grafu se používá několik hodnotících kritérií [2]. Dosáhnout maximálního hodnocení ve všech kritériích je značně obtížné, u rozsáhlých grafů téměř nemožné. Nehledá se tedy ideální rozložení, ale rozložení s nejvyšším hodnocením.

- Minimální počet křížení hran
- Minimální celkové rozměry grafu
- Minimální počet ohybů hran
- Dostatečně velký úhel svíraný dvěma hranami
- Co nejvyšší symetrie celého grafu



Obrázek 3.1: (A) Příliš křížení, (B) Příliš ohybů, (C) Příliš malý úhel, (D) Žádná symetrie

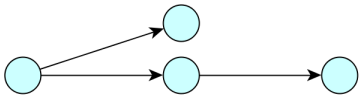
3.2 Typy grafů

Grafy můžeme rozdělit do několika základních kategorií. Každá se liší jak svým vzhledem, tak vhodností pro konkrétní použití. Algoritmus obvykle generuje jen jeden typ, proto je důležité seznámit se s nimi hned na začátku.

3.2.1 Dle uspořádání uzlů

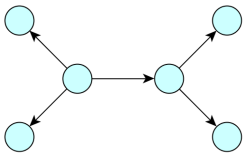
Stromový typ

Graf je uspořádán na způsob stromu. Je vhodný zejména pro orientované, acyklické grafy, u kterých je toto rozložení nejpřehlednější i nejlogičtější. Je tedy velmi vhodný pro cíl této práce – zobrazení referencí knihoven.



Force–based typ

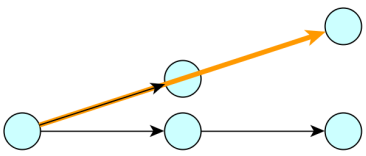
Tento typ je generován algoritmy založenými na fyzikálních jevech, které hledají takové rozložení, ve kterém graf vykazuje minimální energie. Jeho uzly si lze například představit jako magnety, které se navzájem odpuzují. Působením takových sil vznikne rozložení podobné tomu na obrázku.



3.2.2 Dle typu hran

Přímé hrany

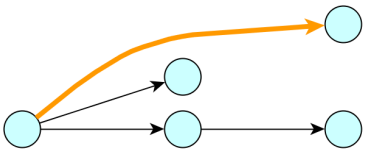
Základní a nejjednoduší forma hran. Uzly spojuje úsečka. Tato varianta je často nepraktická a do značné míry omezuje kvalitu rozložení grafu. Často způsobuje zbytečné křížení jak hran, tak hrany s uzly.



Obrázek 3.2: Nevýhoda přímých hran – oranžová hrana zbytečně kříží jinou hranu i uzel.

Nepřímé hrany

Nepřímé hrany jsou náročnější na implementaci, ovšem oproti přímým hranám jsou podstatně flexibilnější a umožňují kvalitní rozložení grafu na menším prostoru.



Obrázek 3.3: Nepřímá hrana se elegantně vypořádá i s potenciální kolizí.

3.3 Matematický model

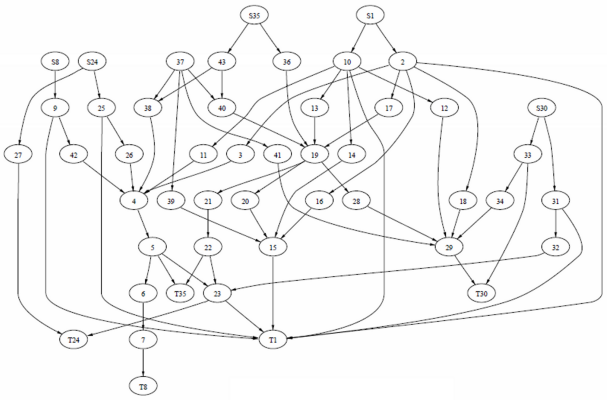
Představuje standardní přístup k pozicování grafu. Algoritmy se pokouší ohodnotit uzly, případně i hrany tak, aby na základě těchto ohodnocení sestavily optimální výstup.

3.3.1 Model vyvinutý AT&T Bell Laboratories

Příkladem matematického modelu je algoritmus vyvinutý v laboratořích AT&T Bell [3]. Jeho výstupem je stromový graf s nepřímými hranami, což ho činí ideální kandidátem pro zobrazení grafu referencí knihoven. Princip jeho fungování lze popsat těmito kroky:

- Ohodnocení uzlů
- Řazení uzlů (v rámci stromu)
- Pozicování (umísťování uzlů na výstup)
- Vykreslení hran

Jeho výhodou oproti dále uvedenému algoritmu je rychlost a především kvalita výstupu. Jeho nevýhodou je vysoká složitost a náročnost implementace.



Obrázek 3.4: Výstup algoritmu AT&T.

3.4 Fyzikální model

Fyzikální modely simulují prostředí s danými fyzikálními zákony. Jejich princip fungování se dá popsat ve dvou krocích:

- Inicializuj prostředí
- Nech prostředí působit na uzly

Základní princip je tedy velice jednoduchý. Další náročnost už je dána propracovaností fyzikálního prostředí, ale o tom už konkrétně v další části.

3.4.1 Model magnetických pružin

Model magnetických pružin představuje uzel jako magnet (jehož magnetické pole je dále prezentováno jako *ideální vzdálenost mezi uzly*) a hranu jako pružinu [5]. V tomto modelu platí dvě jednoduchá pravidla:

- Uzly se vždy odpuzují (nejnižší energii mají, pokud jsou daleko od sebe)
- Pružina může uzly přitahovat nebo odpuzovat, podle momentálního napnutí (nejnižší energii má při své původní délce)

Pružiny v takovém prostředí zaručují optimální délku hrany, magnety pak zajišťují minimální odstupy uzlů a částečně i dostatečné úhly mezi hranami. Kombinace obou prvků pak ve většině případů i úspěšně redukuje množství křížení hran.

Kroky algoritmu

1 Umísti uzly do výchozí pozice (viz níže)

2 Opakuj n-krát

  2.1 Spočítej síly působící na tento uzel

  2.2 Posuň uzel v souladu s vypočítanými silami

3 Vykresli graf

Počáteční podmínky

Pro dosažení co nejlepších výsledků je nutné před spuštěním algoritmu inicializovat pozice uzlů. Ideálním rozložením je kruh s poloměrem určeným vzorcem

$$\frac{k \cdot V}{2\pi}$$

kde k je ideální vzdálenost mezi uzly a V je počet uzlů. Uzly se umísťují náhodně. Výchozí rozložení může do značné míry ovlivnit celkový výsledek. Při nedostatečném výsledku je tedy možné celý proces zopakovat s jiným výchozím rozložením.

Magnetická síla uzlu

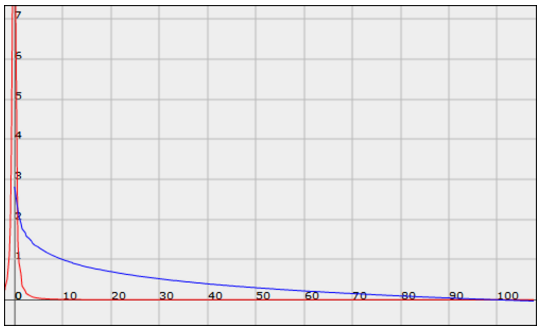
Magnetická síla uzlu je vždy odpudivá. Působí ve všech směrech a klesá s rostoucí vzdáleností od středu uzlu. Vliv této síly na jeden uzel je nutné počítat ve vztahu ke všem ostatním uzlům v systému. Podle citovaného zdroje se odpudivá síla počítá ze vzorce

$$c_r \cdot \frac{1}{d^2}$$

kde c_r je modifikátor pro ladění velikosti síly a d je aktuální vzdálenost mezi uzly (délka pružiny). Tato funkce ale klesá příliš strmě, a proto se mi jako vhodnější jeví vzorec

$$c_r \cdot \log\left(\frac{k}{d}\right)$$

kde k je ideální vzdálenost mezi uzly



Obrázek 3.5: Červená funkce představuje původní vzorec, modrá nový.

Přitažlivá/odpudivá síla pružiny

Síla působící pružinou může být přitažlivá či odpudivá v závislosti na tom, zda je její aktuální délka delší než základní (pružina je napnutá, působí přitažlivě) nebo kratší (pružina je stlačená, působí odpuvidě). Její vliv na aktuální uzel se počítá pouze ve vztahu k přímo připojeným uzlům. Nestačí ji však počítat pouze pro jeden uzel z dvojice. Je vždy nutné, počítat sílu a posun pro uzly na obou stranách hrany. Síla pružiny se počítá podle vzorce

$$c_s \cdot \log\left(\frac{d}{k}\right)$$

kde c_s je modifikátor sloužící pro ladění velikosti síly, k je ideální vzdálenost a d je aktuální vzdálenost mezi uzly.

Obrázek 3.6: Modrá pružina je v základní délce, červená je přepjatá. Pokud by se počítala její síla pouze ve vztahu k bodu B, dostal by se pod B doprostřed úsečky AC, což není ideální stav. Proto se musí počítat i vzhledem k bodu A, který se přisune k B a úsečky AB a BC tak budou mít optimální délku.

Kapitola 4

Vykreslování grafu

Základním posláním grafu je usnadnit pozorovateli pochopení stavu a vztahů věcí skrze jejich vizualizaci. Kvalita vykreslení grafu je tedy minimálně stejně důležitá, jako jeho napozicování. I dokonale napozicovaný graf může být nepoužitelný, pokud jeho grafický styl informace dezinterpretuje či zatemní. Naopak kvalitně vykreslený graf s jasnými pravidly dokáže vykompenzovat méně kvalitní pozicování.

4.1 Grafický styl

Pro vytvoření použitelného a snadno pochopitelného grafu je volba grafického stylu zcela zásadní — ať už jde o volbu barevnosti (barevný vs. monochromatický), rozmanitosti velikostí uzlů (statická vs. dynamická velikost) či zamýšleného celkového dojmu (populární vs. manažerský). Grafický styl by měl být volen tak, aby výsledný graf nenabízel méně ani více informací, než bylo zamýšleno. Graf má pomoci pochopit myšlenku, a proto by ji měl vyjádřit v její nejčistší podobě. Například graf mající ozřejmit vztahy jinak rovnocenných elementů nepotřebuje ani barvy, ani dynamickou velikost uzlů, protože oba tyto prvky poskytují nadbytečné informace a mohou grafu dávat jiný či další význam. A nejen to — nadbytečné informace svádí pozorovatele hledat souvislosti i tam, kde žádné nejsou a odvádět jeho pozornost od hlavní myšlenky. V nejhorším případě se pak může stát, že pozorovatel pochopí graf jinak, než byl úmysl tvůrce (což je mnohem horší, než kdyby ho nepochopil vůbec).

Barevnost je v grafech obyčejně chápána jako kategorizace. Prvky různých barev řadíme do stejné kategorie.

Dynamická velikost dodává uzlům rozměr důležitosti. Větší uzly jsou více důležité než menší.

Celkové pojetí může působit spíše psychologicky, i přesto je však velice důležité a může hlavní myšlenku deformovat.

Zvolíme-li parametry všech tří kategorie nevhodně, získáme graf, který i přes svou objektivní správnost není schopen předat potřebné informace. Všechny tři kategorie také poskytují prostor pro demagogii, úmyslné i nechtěné zkreslení informací apod. To už se ale dotýkáme spíše oblasti psychologie, která není předmětem této práce.

4.2 Zlepšení čitelnosti

V oblasti grafů můžeme slovo čitelnost považovat za synonymum pochopitelnosti. S rostoucí složitostí grafu jeho čitelnost významně klesá, proto je vhodné využít všechny dostupné prostředky pro její zvýšení.

4.2.1 Statická vylepšení

Vylepšení vhodná pro statické výstupy, především pro tisk.

Barevnost

Barvy mohou významně zvýšit čitelnost grafu, ovšem stejně tak ji mohou definitivně pohřbít. Abychom se vyhnuli druhému případu, je nutné definovat pro barevnost jednoznačná a snadno pochopitelná pravidla.

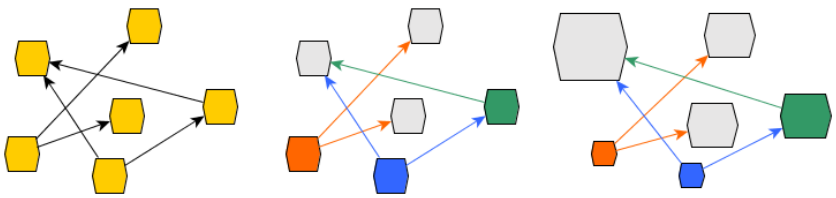
Kategorizace — Základní pravidlo a hlavní důvod, pro který se barevnost používá. Barvy mají části grafu kategorizovat a abstrahovat tak jeho obsah na vyšší úroveň (resp. opticky snížit počet prvků, které je nutné vnímat odděleně).

Méně je více — Počet barev by měl být minimální možný. Navazuje na předchozí pravidlo, příliš mnoho barev potlačuje abstrakci a zvyšuje množství informací, které je nutné vnímat odděleně.

Dostatečný barevný odstup — Barvy musí být snadno rozlišitelné (není vhodné použít vedle sebe mandarinkovou a pomerančovou oranžovou).

Dynamické velikosti

Změnou velikosti uzlu, případně i hrany, můžeme do grafu vnést nejen nové informace, ale i výrazně zvýšit jeho čitelnost. Významné uzly zvětšíme na úkor méně významných, čímž dosáhneme dalšího zvýšení abstrahovanosti grafu. I zde je však nutné držet se rozumné míry rozložení velikostí. S rostoucím poměrem velkých uzlů ku malým klesá účinnost tohoto vylepšení.



Obrázek 4.1: Obarvené uzly a jejich hrany zvyšují čitelnost grafu (uprostřed). Dynamickou velikostí umožníme pozorovateli soustředit se především na významné uzly a opět tak zlepšíme čitelnost (vpravo). V tomto případě dynamická velikost nepřináší nové informace, pouze zdůrazňuje stávající.

Přehlednější křížení

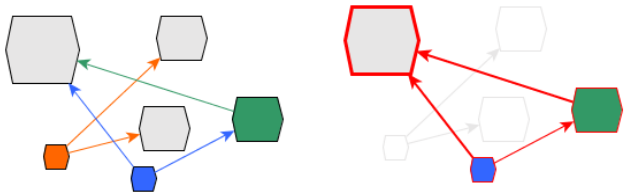
Nachází-li se v grafu větší počet křížení hran, je vhodné vykreslit křížení takovým způsobem, aby nebylo pochyb, která hrana jakým směrem pokračuje (přemostěním v pseudo 3D, ztučněním části jedné z čar apod.)

4.2.2 Dynamická vylepšení

Mohou dramaticky zvýšit čitelnost; podstatně více, než statická vylepšení. Jejich použitelnost je však omezena na elektronické použití ve specializovaných programech.

Zvýraznění části grafu

Dynamické zvýrazňování pouhých částí grafu (například v reakci na polohu kurzoru myši) nám umožní v jednom okamžiku vnímat řádově menší množství informací a vstřebávat tak graf po částech.



Obrázek 4.2: Zvýraznění pouhé části grafu dramaticky zvyšuje přehlednost.

Kapitola 5

Implementace analýzy referencí

5.1 Technologické zázemí

Modul analýzy referencí je implementován s maximální mírou nezávislosti na volajícím prostředí. Kompletní proces analýzy i generování výsledků probíhá v odděleném vlákně, které pro něj vytvoří sám modul. Vlastní analýza pak probíhá i v oddělené aplikační doméně, což je kriticky důležité, protože modul nezanáší aplikační doménu volající programu a umožňuje mu tak bezpečné opakování analýz.

5.2 Skladování výsledků analýzy

V souladu se strukturou referencí .NET knihoven (každá knihovna má pouze jednu úroveň referencí) stačí pro každou knihovnu udržovat pouze její přímé reference. Není nutné ani vhodné udržovat pro každou knihovnu kompletní strom. Zcela vyhovující těmto potřebám je třída *Dictionary<>* ze jmenného prostoru *System.Collections.Generic*.

Dictionary<> je generický datový typ fungující na principu hash tabulky (dvojice klíč–hodnota). Oproti standardní hash tabulce má její generická varianta tu výhodu, že klíč i hodnota mohou být reprezentovány libovolným datovým typem.

Jedna dvojice klíč–hodnota reprezentuje jednu assembly. Jako klíč je použit plný název assembly. Hodnotu reprezentuje vlastní datový typ obsahující plný název a hash tabulku referencovaných knihoven.

```
Dictionary<string, AssemblyDetail> AssembliesDetails;

class AssemblyDetail
{
   public string AssemblyName;
   public Dictionary<string, Assembly> ReferencedAssemblies;
}
```

Jak je vidět v ukázce kódu, v hash tabulce ReferencedAssemblies už stačí jako hodnotu uchovat identifikaci assembly – její detail lze získat z hash tabulky AssembliesDetails, protože právě tam se ukládají informace o všech assembly, na které analýza narazí.

5.3 Algoritmus vyhledávání referencí

Jak již bylo řečeno, jsou závislosti knihoven jednoúrovňové, což výrazně zjednodušuje návrh algoritmu. Je implementován pomocí jednoho cyklu a jednoho rekurzivního volání. Základní průběh analýzy v pseudokódu:

```
1  Dictionary<string, AssemblyDetail> results;
2
3  function Analyze(Assembly assembly)
4  {
5      references = GetRefrerencedAssemblies;
6      foreach (reference in references)
7      {
8          results[assembly.Name].ReferencedAssemblies
9                                .Add(reference.Name, reference);
10         Analyze(reference);
11     }
12 }
```

Ad řádek 1 Datové úložiště pro výsledky analýzy

Ad řádek 5 Načte referencované assembly, kterými pak na řádku 6 iteruje

Ad řádek 8 Pod klíč názvu analyzované assembly přidá záznam o referencované knihovně

Ad řádek 10 Pro každou referencovanou knihovnu spouští její analýzu

Po dokončení funkce Analyze() se v úložišti results nachází všechny referencované knihovny včetně seznamu jejich závislostí.

Analýza více knihoven

Budeme-li chtít analyzovat více knihoven a jejich vzájemné vztahy (typicky pro analýzu celých řešení či projektů), stačí použít funkci Analyze() v cyklu na každou knihovnu určenou k analýze. Díky elegantnímu návrhu a společnému úložišti výsledků můžeme takto jednoduše získat analýzu vztahů neomezeného množství knihoven (resp. množství omezené velikostí zásobníku – kvůli rekurzi).

5.4 Vstupní data

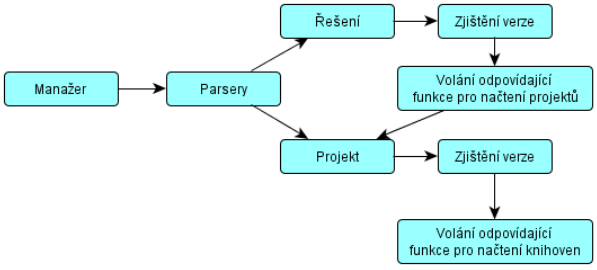
Jako vstupní data slouží seznam cest k souborům knihoven, který se ještě před spuštěním analýzy předá přes veřejnou vlastnost analýzy – LibrariesToAnalyze. Pro usnadnění práce programátorovi je součástí jádra programu i souborový manažer s vysokou mírou automatizace. Pro zjednodušení práce se soubory je součástí jádra programu souborový manažer. Výčet jeho schopností zároveň naznačuje záležitosti, kterými bylo nutné se zabývat.

- Načtení jednoho souboru
- Načtení celého adresáře

- Načtení celého VS projektu (.csproj)
- Načtení celého VS řešení (.sln)
- Verzování souborů .csproj i .sln
- Podpora konfigurací kompilátoru VS

5.4.1 Verzování souborů .csproj a .sln

Od verzování schémat projektových souborů i souborů řešení se odvíjí celá architektura manažeru. Různé verze Visual Studia používají různé verze XML schématu, jejichž struktura se tak může v některých důležitých aspektech lišit. Proto je v souborovém manažeru implementována detekce verze a následný automatický výběr odpovídající parsovací funkce. Souborový manažer využívá ke své činnosti statickou třídu *Parsers*, která poskytuje funkce



Obrázek 5.1: Načítání souborů z projektových souborů a souborů řešení.

potřebné ke korektnímu načtení rozličných verzí schémat. Automatizovaná detekce verzí umožnila abstrahovat verzování a programátor používající tuto třídu se jím nemusí vůbec zabývat (za předpokladu, že používá pouze podporované verze schémat).

Parsování projektového souboru .csproj

Projektový soubor obsahuje cesty ke zkompilovaným knihovnám. Protože těchto cest může obsahovat více (většinou minimálně Debug a Release), je tomu přizpůsobené i datové úložiště. To je vytvořeno ze dvou vnořených hash tabulek. První z nich používá jako klíč název konfigurace a jako hodnotu druhou hash tabulku; ta má jako klíč cestu ke knihovně a jako hodnotu detailní informace o knihovně. Samotné načítání konfigurací a cest k souborům je realizováno pomocí tříd ze jmenného prostoru *System.XML* a s ohledem na jeho triviálnost ho zde nebudeme rozvádět.

```
Dictionary<string, Dictionary<string, Library>>
```

Parsování souboru řešení .sln

Soubor řešení se od projektového souboru zásadně liší pouze v jedinné věci – není ve formátu XML. Proč tomu tak je i v nejnovější verzi Visual Studia (2010) mi není známo. Až na jiný přístup k parsování je princip jeho fungování defacto stejný jako u projektových souborů – verzování, konfigurace, cesty k projektovým souborům (s tím, že konfigurace na úrovni celých řešení není dořešená a v současné verzi programu chybí). Parsování

souboru probíhá pomocí regulárních výrazů a nástrojů pro práci s nimi (jmenný prostor *System.Text.RegularExpressions*). I přesto, že se nejedná o XML soubor, je získání cest k projektovým souborům poměrně snadné a stačí na něj jeden regulární výraz:

```
Regex("[^\"]*?.csproj")
```

5.5 Výstupní data

5.5.1 XML verze s plnými informacemi

Formát výstupního XML souboru defacto kopíruje strukturu datového uložiště v paměti. Výstupem je seznam všech analyzovaných assembly, přičemž každá obsahuje všechny části svého plného jména a kompletní seznam referencí.

```
<Assemblies>
  <Assembly Name="" Version="" Culture="" PublicKeyToken="">
    <References>
      <Assembly Name="" Version="" Culture="" PublicKeyToken="">
      ...
    </References>
  </Assembly>
  ...
</Assemblies>
```

5.5.2 Unifikovaný výstup pro generátor grafu

Pro generátor grafu obsahuje XML s plnými informacemi zbytečně moc údajů. S ohledem na univerzálnost generátoru (měl by umět rozmístit libovolné body, ne jen assembly) generuje modul analýzy redukovanou verzi XML. Tato verze byla naržena jako univerzální definice obecných bodů a jejich vztahů.

```
<Nodes>
  <Node Name="">
    <References>
      <Node Name="">
      ...
    </References>
  </Node>
  ...
</Nodes>
```

Kapitola 6

Implementace generátoru grafu

6.1 Model magnetických pružin

Fyzikální model je implementován ve třídě *Node*. Ta reprezentuje jeden každý uzel, který si dokáže sám spočítat síly, které na něho působí. Pro jeden krok generování grafu tak stačí na všechny uzly v systému zavolat funkci *MakeStep*. Protože každý uzel zná všechny ostatní uzly v systému a zná i uzly, se kterými je spojen hranou, spočítá na něj působící síly a s ohledem na ně, změní svou pozici. Každou nově vygenerovanou pozici ukládá do historie, takže je možné kdykoliv vyvolat jakýkoliv vygenerovaný stav. To se hodí především při hledání nejlepšího stavu grafu.

6.1.1 Měnitelné parametry modelu

Model se dá jednoduše ladit pomocí těchto parametrů:

SpringForceModifier upravuje sílu pružiny.

NodeRepulsionForceModifier upravuje odpudivou sílu uzlu.

NodeMinimalMargin definuje minimální odstup uzlu (každého uzlu – minimální odstup mezi dvěma uzly je tedy dvojnásobek této hodnoty).

IdealDistanceModifier upravuje ideální vzdálenost mezi uzly.

6.1.2 Působení sil v modelu

Magnetická síla uzlu

Ideální vzdálenost mezi uzly se v tomto případě dynamická a mění se spolu s velikostí daných uzlů. Připočítává se k ní i dvojnásobek hodnoty NodeMinimalMargin.

```
idealDistance = IdealDistanceModifier * (this.Size + neighbor.Size)
                + (2 * NodeMinimalMargin);
```

Funkce je lehce optimalizovaná tak, že pokud je aktuální vzdálenost mezi uzly větší než ideální vzdálenost, bez dalšího počítání vrací nulu.

Síla pružiny

Ideální vzdálenost v tomto případě není nutné nijak upravovat, takže se použije pouhý součet velikostí obou uzlů, násobený modifikátorem. O minimální odstup uzlů se totiž stará jejich magnetická síla.

```
idealDistance = IdealDistanceModifier * (this.Size + neighbor.Size);
```

Kombinace sil a posun

Protože celý algoritmus je defacto založen na vektorech, je součástí řešení i třída poskytující základní operace s nimi. Přesto není nutné kombinovat síly jejich vektorovým součinem, protože stačí po každém výpočtu nové síly ihned změnit pozici. Algoritmus používá při výpočtu vždy jednu neměnnou hodnotu, není tedy nutné obávat se vzniku chyb (algoritmus vždy používá pozice, které mají body v aktuálním čase – nové pozice se sice zaznamenají, ale použijí se až v dalším kole).

6.1.3 Průběh generování grafu a hodnocení stavů

Jeden běh generování grafu provede sto kroků pro každý uzel. Jelikož se stav grafu mění s každým krokem, je nutné po každém kroku aktuální stav ohodnotit. Výsledek hodnocení se ukládá pod stejným číslem kola jako pozice uzlů. Běh generování grafu je několikrát opakován, to proto, aby vycházel z jiných počátečních podmínek. Tím se zvýší šance na získání co nejlepšího hodnocení některého ze stavů. Po dokončení všech běhů generování se z lineárního seznamu hodnocení vybere nejnižší hodnocení a podle jeho časové známky (číslo kola) se obnoví adekvátní stav grafu z historie.

Hodnocení stavu

Stav se hodnotí podle dvou kritérií

- Počet křížení hran (každé křížení přičítá bod)
- Počet kolizí hrany s uzlem (každé křížení přičítá dva body)

Kolize hrany s uzlem je závažnější než křížení hran, proto je hodnoceno přísněji. Čím vyšší je hodnocení, tím horší je hodnocený stav grafu.

6.2 Vstupní data

Třída implementující algoritmus přijímá jako vstup XML dokument. Ke své práci potřebuje především hierarchii uspořádání uzlů a jejich velikosti. XML dokument má následující strukturu a minimálně uvedené atributy:

```
<Nodes>
  <Node Name="" Width="" Height="">
    <References>
      <Node Name="">
      ...
```

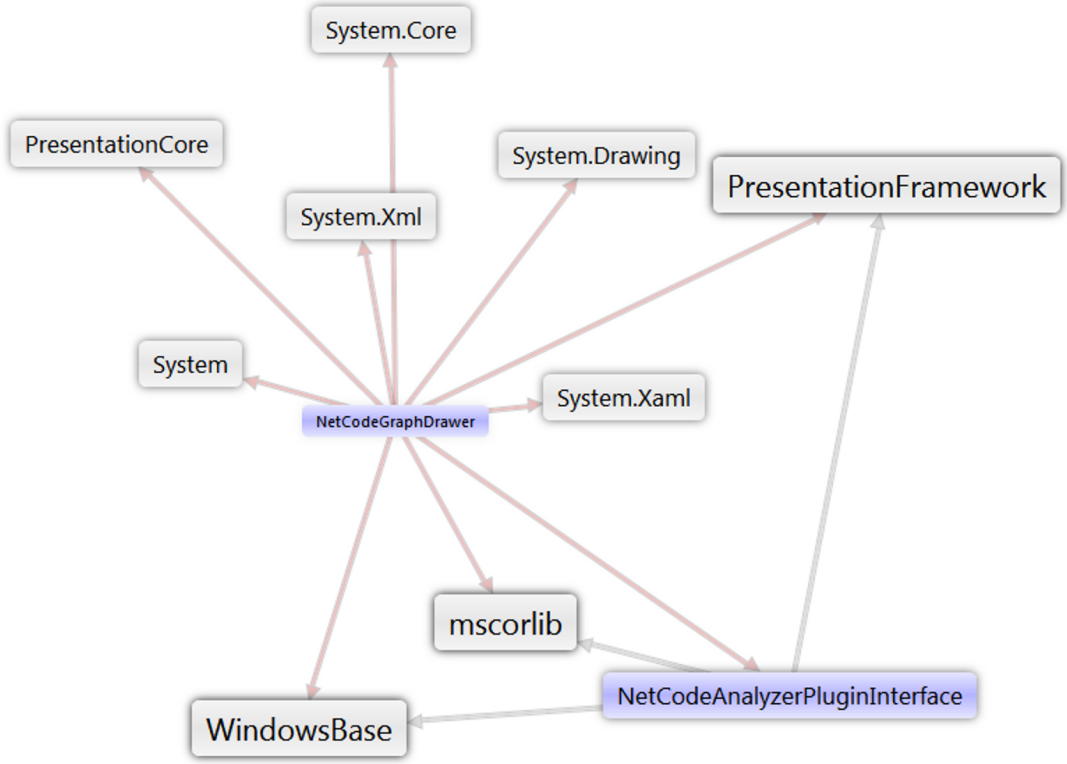
```
    </References>
  </Node>
  ...
</Nodes>
```

6.3 Výstupní data

Formát výstupního XML dokumentu je téměř totožný se vstupním, jen místo rozměrů je uvedena pozice uzlu.

6.4 Výsledky

Pro menší počet uzlů a referencí funguje algoritmus bezvadně. S rostoucím počtem uzlů a jejich vztahů samozřejmě roste i počet křížení, čemuž se ale většinou nelze vyhnout v žádném algoritmu. U tohoto algoritmu je to však o to větší problém, že používá přímé hrany a nestromovou strukturu.



Obrázek 6.1: Výsledek grafu po osmi krocích.

Kapitola 7

Implementace vykreslování grafu

Zobrazovač grafu používá technologii interaktivního zvýrazňování logických celků. K lepší přehlednosti přispívá také barevné odlišení logických celků a dynamická velikost uzlů. Ani jedno z uvedených rozšíření nebylo díky dobře navrženým datovým úložištím velký problém implementovat. Náročnější však byla implementace algoritmu pro vykreslení šipky – především část starající se o nalezení vhodného cílového bodu (tj. bodu na okraji rámečku uzlu).

7.1 Technologie

Vykreslovací systém používá pro zobrazení uzlů i šipek standardní ovládací prvky. Například uzel je prvek *UserControl*, skládající se z komponent *Grid*, *Border* a *Label*, nakombinovaných způsobem umožňujícím dynamické změny kdykoliv během existence uzlu. I během zobrazování se tedy dá měnit barva, průhlednost či velikost. Toho je plně využito v interaktivním zvýrazňování logických celků v grafu. Vykreslovací algoritmus komunikuje s ovládacími prvky skrze rozhraní, což do budoucna umožňuje provádět změnu stylů uzlů i šipek. V současné verzi aplikace však stylování není podporováno.

Uzel Uzel je definován jazykem XAML a využívá kombinaci tří komponent tak, aby se daly nezávisle měnit parametry jako barva pozadí, intenzita stínu či celková průhlednost. Aby byl uzel akceptovatelný vykreslovačem, musí implementovat rozhraní *INodeGraphics*. Toto rozhraní obsahuje překvapivě mnoho funkcí a vlastností, musí totiž uspokojit požadavky interaktivního zvýrazňovače, namátkou – základní barva pozadí, aktuální barva pozadí, průhlednost, průhlednost pozadí, rozměry…

Šipka Šipka je stejně jako uzel popsána jazykem XAML. Její implementace je výrazně jednodušší, i přesto je však složená ze tří komponent. Primárními parametry jsou pro ni délka a úhel natočení. Její polohový vektor tedy není určen dvěma body, ale výchozím bodem a úhlem. V jazyce XAML byl původně zapsán i trigger pro její zvýraznění po najetí myši. Bohužel se z nezjištěných příčin dostával do konfliktu s interaktivním zvýrazňovačem a musel být tedy odstraněn.

7.2 Pozicování

7.2.1 Uzly

Základní pozice je určena generátorem rozložení grafu. Ten ale počítá pouze s body, nebere ohled na velikost uzlu. Obdržené souřadnice se proto sníží o polovinu výšky/šířky uzlu, takže vygenerovaný bod se ocitne ve středu vykresleného uzlu. Tento posun se provádí především kvůli vykreslování šipek, které by měly vycházet ze středu a cílit opět na střed uzlu (jinak by bylo lhostejné, zda se uzly vykreslí na střed, levý horní roh či jakoukoliv jinou pozici).

Obrázek 7.1: Posun souřadnic tak, aby se střed uzlu kryl s generovaným bodem.

7.2.2 Šipky

Výchozí pozice šipky je dána vygenerovaným bodem. To je umožněno výše zmíněným posunem středu uzlu na generovaný bod. O něco obtížněji je potřeba získat délku šipky a úhel rotace.

Rotace

Úhel rotace je vypočítán z cosinovy věty:

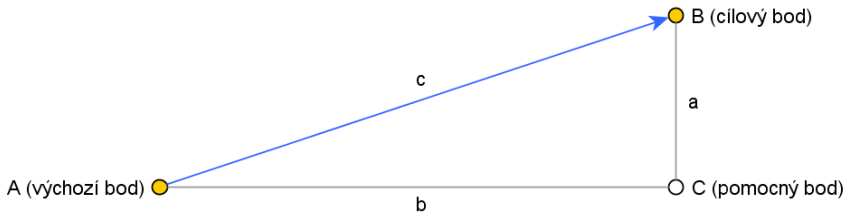
$$a^2 = b^2 + c^2 - 2bc \cdot \cos\alpha$$

Pro získání úhlu α upravíme rovnici do tvaru:

$$\alpha = \arccos\left(\frac{b^2 + c^2 - a^2}{2 \cdot b \cdot c}\right)$$

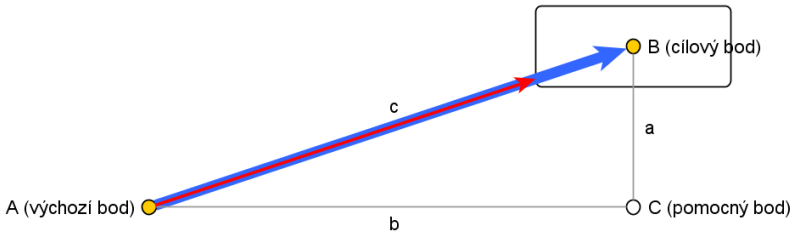
Délky stran a, b získáme jako rozdíl souřadnic výchozího a cílového bodu. Stranu c dopočítáme z pythagorovy věty pomocí stran a, b. Po spočítání úhlu je ještě potřeba upravit úhel podle toho, který z dvojice bodů výchozí–cílový má vyšší souřadnici Y.

```
výchozíUzel.Souřadnice.Y > cílovýUzel.Souřadnice.Y ? -1 * alfa : alfa;
```



Délka

Získání délky šipky naráží na problém, že uzly jsou různě veliké a šipka se musí zastavit na okraji uzlu. Před jejím vykreslením tedy musíme získat bod, který se nachází na průsečíku spojnice bodů a rámečku cílového uzlu. V tomto případě vychází algoritmus ze zjednodušené

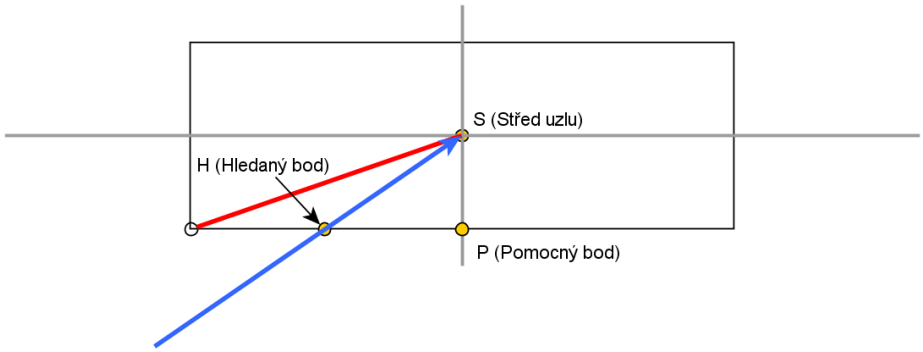


Obrázek 7.2: Délka modré šipky je známa, délku červené musíme dopočítat.

směrnicové rovnice přímky:

$$y = k \cdot x$$

Výpočty je nutno rozdělit do 8 částí. V první fázi je nutné zvolit kvadrant (tj. ten, kterým šipka prochází směrem ke středu) a v druhé fázi rozlišit, zda šipka prochází horizontální nebo vertikální hranou uzlu. Kvadrant jednoduše zjistíme pomocí souřadnic obou bodů



Obrázek 7.3: Náčrt pro výpočet ve třetím kvadrantu.

(porovnáním X a Y souřadnic). Zda šipka prochází horizontální či vertikální hranou uzlu můžeme zjistit například porovnáním směrnic šipky a přímky procházející středem uzlu a rohem kvadrantu (červená čára v obrázku 7.3). Cílem výpočtu jsou souřadnice bodu H. Zůstaneme-li u obrázku 7.3, postup výpočtu bude následující:

- Směrnice modré i finální šipky jsou stejné. Souřadnice krajních bodů modré šipky známe a můžeme tedy spočítat její směrnici.
- Ze tří proměnných zjednodušené směrnicové rovnice přímky nyní známe dvě – směrnici (k) a délku úsečky SP (y).
- Dosadíme k a y a získáme hodnotu x. Získali jsme souřadnice hledaného bodu H.
- Dopočítat finální délku šipky pomocí pythagorovy věty je už triviální.

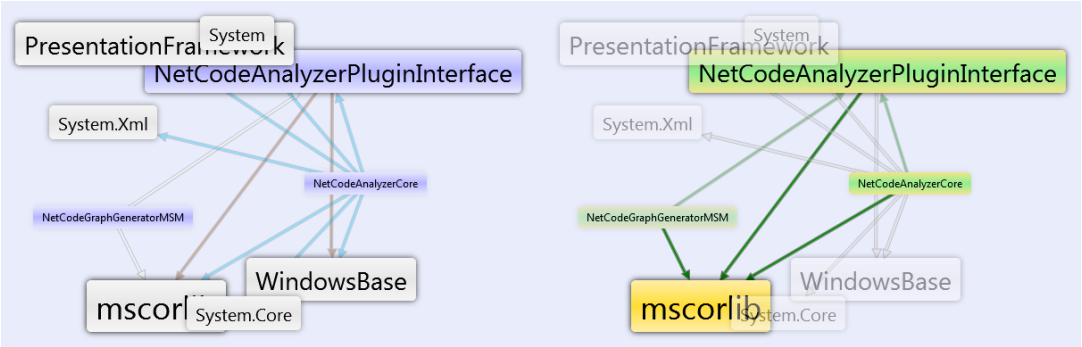
Uvedený algoritmus je potřeba implementovat v drobných obměnách pro všech osm částí. Pak už stačí jen vytvořit objekt šipky, předat mu hodnotu délky a zavolat funkci rotace, která šipku nasměruje správným směrem.

7.3 Načtení a uložení vstupních dat

Modul přijímá jako vstup XML dokument se stejnou strukturou, jakou používá generátor pozic. I způsob ukládání je podobný – v tomto případě je ale rozšířený i o zpětné reference. Uzel tak zná nejen referencované uzly, ale i uzly, které referencují jeho. Současně si udržuje informace o všech objektech příchozích i odchozích šipek. To je zcela zásadní pro implementaci interaktivního zvýrazňovače.

7.4 Interaktivní zvýrazňování

Interaktivní zvýrazňování představuje metodu významně zlepšující přehlednost grafu. Aby přinesla kýžený efekt, je potřeba vybrat vhodné logické celky. Po několika pokusech vyšla jako ideální kombinace dvou barev a dvou stupňů intenzity. Barvy slouží k rozlišení příchozích a odchozích referencí (objekt referencuje vs. objekt je referencován). Stupeň intezity pak vzdálenost reference (přímo referencované uzly jsou nejtmavší, uzly referencované přes několik jiných uzlů jsou světlejší).



Obrázek 7.4: Interaktivní zvýraznění přehlední i jinak nepoužitelný graf.

Proces zvýraznění

- Každému uzlu jsou přiřazeny obsluhy událostí MouseEnter a MouseLeave.
- MouseEnter
  1. Obarví odesílatele události.
  2. Rekurzivně obarvuje předky i následníky a v každém zanoření zvýší průhlednost.
     Rekurze se v tomto případě řídí jediným pravidlem – zachovává směr. Prochází buď jen referencované uzly nebo jen referující uzly. Tím je zaručeno, že označí opravdu pouze související uzly a ne rekurzivně celý graf.
  3. Znovu iteruje přímými následníky i předky, aby je obarvil s nulovou průhledností (mohly být nesprávně zprůhledněny zpětnou referencí jiného vázaného uzlu).

- MouseLeave

    1. Iteruje všemi uzly a hranami a nastavuje jim implicitní vlastnosti.

7.5 Další použité metody pro zpřehlednění grafu

7.5.1 Obarvení logických celků

Šipky vycházející ze stejného uzlu jsou obarveny stejnou barvou. Ta je generována náhodně a vychází z přednastavených barev třídy *System.Windows.Media.Brushes*. Odlišné barvy jsou jednoznačným přínosem pro přehlednost grafu.

7.5.2 Dynamická velikost uzlů

Rozměry uzlu jsou tím větší, čím více je uzel referencován. Tento způsob hodnocení vychází z předpokladu, že často používaný (a tedy referencovaný) uzel je důležitější než uzel, který sice referencuje mnoho knihoven, ale sám referencovaný není. Hodnota důležitosti uzlu se počítá jako poměr počtu referencujících uzlů ku počtu referencujících uzlů nejreferencovanějšího bodu.

```
mostImportant = Nodes.Max(a => a.IncomingReferences);
this.Importance = this.IncomingReferences / mostImportant;
```

Jak naložit s hodnotou důležitosti je již záležitost grafického stylu uzlu. V aktuálně použitém stylu se velikost uzlu přizpůsobuje velikosti písma, která se počítá jako dvojnásobek hodnoty důležitosti, přičtený k aktuální (základní) velikosti.

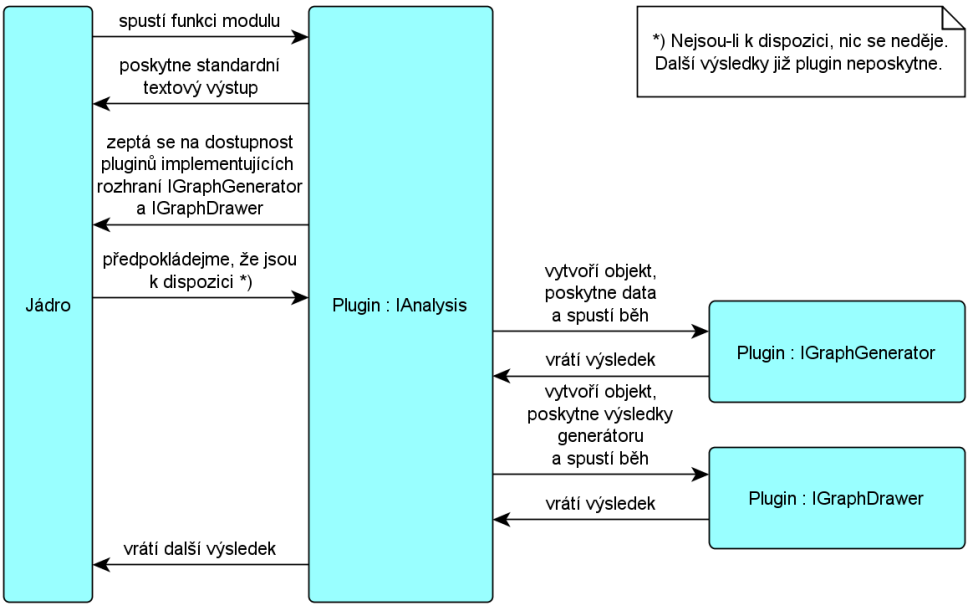
Kapitola 8

Aplikace

Aplikace a především její grafické rozhraní bylo navrženo a implementováno s důrazem na maximální jednoduchost a uživatelskou přívětivost s využitím moderním prvků ovládní. S ohledem na velké množství analýz, které je možné nad projekty provádět, je jádro aplikace navrženo na bázi zásuvných modulů. To umožňuje snadné začleňování modulů poskytujících nové funkčnosti.

8.1 Zásuvné moduly

Celý systém zásuvných modulů je navržen tak, aby jádro aplikace zastávalo co nejmenší úlohu. V podstatě pouze sdružuje načtené moduly. Důvodem je fakt, že modul (resp. jeho programátor) ví nejlépe co potřebuje a jak toho dosáhnout. Jádro poskytuje několik typizovaných rozhraní. Díky sdílenému seznamu načtených modulů si tak každý modul může zjistit, zda se v aplikaci nachází jiný modul, implementující dané rozhraní a tím pádem poskytující danou funkčnost. Na konkrétní implementaci přitom vůbec nezáleží. Jak je



Obrázek 8.1: Nástin principu fungování zásuvných modulů.

vidět z diagramu, základní myšlenka spočívá v tom, že modul může rozšiřovat své možnosti na základě přítomnosti jiných modulů. Například analýza referencí umí vrátit výsledek pouze jako XML dokument. Pokud jsou ale přítomny moduly pro generování a vykreslení grafu, modul analýzy si toho všimne a použije je pro generování další formy výsledku. Toto chování musí mít modul samozřejmě naprogramováno, neděje se tak automaticky.

8.1.1 Správa zásuvných modulů

O zásuvné moduly, jejich načítání a udržování se stará správce pluginů. Ten se spouští při startu aplikace, v separátním vlákně prochází soubory DLL v definované složce a snaží se je načíst a identifikovat. Pokud v knihovně identifikuje datový typ implementující jedno z předdefinovaných rozhraní, zaznamená ho do skladu pluginů pod příslušné rozhraní. Tento sklad je pak k dispozici zásuvným modulů skrze rozhraní jádra.

```
Dictionary<Type, List<IPlugin>>
```

8.1.2 Rozhraní

Rozhraní jádra

Rozhraní jádra je vskutku minimalistické – modulům jádro poskytuje pouze sklad načtených zásuvných modulů a dispatcher vlákna grafického rozhraní (aby mohly moduly vytvářet grafické prvky ve správném vlákně). To je umožněno/dáno skutečností, že moduly řídí samy sebe a komunikují s jinými moduly přímo, bez asistence jádra.

```
interface ICore
{
    Dictionary<Type, List<IPlugin>> LoadedPlugins { get; }
    Dispatcher GUIDispatcher { get; }
}
```

Rozhraní zásuvných modulů

Základní rozhraní zásuvného modulu, které musí implementovat každý zásuvný modul, také není příliš obsáhlé. Obsahuje pouze informace nutné k jeho identifikaci, předání reference jádra a spuštění jeho funkčnosti. Další možnosti se pak přidávají dynamicky podle typu modulu pomocí dalších rozhraní.

```
interface IPlugin
{
    string Name { get; }
    string Description { get; }
    ICore Core { set; get; }
    void Run();
}
```

Specifická rozhraní

Do této kategorie spadají rozhraní, která přidávají modulu specifickou funkčnost a rozšiřují tak možnosti základního rozhraní. V současné verzi jsou dostupná tato rozhraní:

IAnalysis musí implementovat každý modul poskytující analýzu. V opačném případě nebude veden jako analyzační a nezobrazí se v nabídce dostupných analýz.

IResult implementují moduly vracející nějaké výsledky (analýza tedy musí implementovat nejen rozhraní IAnalysis, ale minimálně i IResult)

IXmlImportSupport implementují moduly umožňující zadání vstupních dat skrze XML soubor

IXmlExportSupport implementují moduly poskytující výsledky ve formátu XML (slouží k jiným účelům než IResult – export XML není primárně určen k zobrazení uživateli)

IAcyclicDirectedGraphDrawer implementují moduly vykreslující acyklické orientované grafy

IAcyclicDirectedGraphGenerator implementují moduly generující rozložení acyklických orientovaných grafů

Rozhraní IResult

Toto rozhraní je velice důležité, protože napovídá princip fungování prezentace výsledků. V souladu s filozofií celé aplikace je i zásuvným modulům poskytována maximální volnost a zároveň odpovědnost. Zásuvný modul si musí sám sestavit grafický prvek, pomocí kterého chce své výsledky prezentovat. To je zcela zásadní výhoda oproti systému, kde se o výběr prostředku pro zobrazení stará jádro či grafické rozhraní. Možnosti takového systému jsou vidět v praktické části této práce, kde je implementován interaktivní zobrazovač grafu.

Modul impelemtující toto rozhraní musí mít vlastnost *Content* typu *ContentControl*, přes kterou nabízí svůj grafický výsledek.

```
ContentControl Content { get; }
```

8.2 Grafické rozhraní

Jednak umožňuje snadný výběr knihoven k analyzování, ale především jako výkladní skříň práce zásuvných modulů. Jinou funkčnost v podstatě nemá, ani nepotřebuje. Je navrženo s ohledem na maximálně jednoduchou obsluhu, přehlednost a přizpůsobitelnost.

8.2.1 Windows Presentation Foundation

Windows Presentation Foundation (WPF) je hardwarově akcelerovaný grafický systém pro operační systém Microsoft Windows. Silně ovlivněn technologiemi HTML, XML a Flash, představuje do jisté míry revoluční pojetí návrhu grafického rozhraní desktopových aplikací. Návrh grafického designu je v zásadě totožný s návrhem pomocí technologií HTML a CSS. WPF používá nový jazyk označovaný jako XAML (čteno jako „zaml“, zkratka pro

Extensible Application Markup Language). Ten umožňuje nejen statický návrh vzhledu a ovládacích prvků, ale i jejich libovolné přestylování a animování [4].

Grafické rozhraní aplikace je založeno právě na této technologii. Jsou na ní založeny i její dvě hlavní komponenty – Fluent Ribbon[1] (ovládací lišta) a AvalonDock[2]. (dokovatelné členění hlavního okna).

Fluent Ribbon

Ovládací prvek Fluent Ribbon je založen na technologii Microsoft Office Fluent™, poprvé použitý v kancelářském balíku Microsoft Office 2007. Fluent Ribbon ovšem není jen ovládací lišta, obsahuje i vlastní definici okna. To proto, aby mohla plně využít svůj potenciál. Z následujícího příkladu je zřejmé, že ani použití WPF, ani komponenty Fluent Ribbon není nikterak složité. Osmi řádky jednoduchého kódu se dá vytvořit základní prostředí připravené pro další vývoj.

```
1 <Fluent:RibbonWindow  x:Class="NetCodeAnalyzer.MainWindow"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4     xmlns:Fluent="clr-namespace:Fluent;assembly=Fluent">
5     <Fluent:Ribbon>
6         <Fluent:RibbonTabItem  Header="Project" />
7     </Fluent:Ribbon>
8 </Fluent:RibbonWindow>
```

- Řádek 1 definuje použití RibbonWindow místo klasického Window.
- Řádky 2–4 definují prostory jmen (obdoba using v C#).
- Řádek 5 definuje vlastní ovládací lištu.
- Řádek 6 definuje záložku s hlavičkou „Project“.

Obě komponenty jsou distribuovány pod volnou licencí (Microsoft Public Licence pro FluentRibbon, BSD pro AvalonDock).

[1] http://fluent.codeplex.com/
[2] http://avalondock.codeplex.com/

Kapitola 9

Závěr

9.1 Hodnocení

Aplikaci i obě knihovny se podařilo implementovat podle původních představ. Analýza referencí, vykreslení grafu i systém zásuvných modulů funguje bez problémů. I když vykreslený graf není dokonalý a určitě by šlo použít lepší algoritmus, poskytuje dostatečnou představu o závislostech knihoven. Vykreslovací plugin dokonce předčil původní očekávání, protože například interaktivní zvýrazňování původně v plánu vůbec nebylo. Systém zásuvných modulů připravil aplikaci na její další snadné rozšiřování a plně splnil původní představy.

9.2 Srovnání s podobnými produkty

Analýza referencí tímto programem poskytuje srovnatelné výsledky jako již zmíněný komerční produkt NDepend. Pravdou je, že jeho grafický výstup používá pokročilejší algoritmus pro generování grafu a je tedy přehlednější, ovšem ostatní záležitosti jako interaktivní zvýrazňování, dynamické velikosti uzlů apod. jsou na stejné úrovni.

9.3 Budoucnost aplikace

Díky systému zásuvných modulů není problém aplikaci rozšiřovat o další funkčnosti a analýzy. Po doplnění několika dalších základních analýz by mohla být uvolněna pro veřejnost. Vzhledem k rozsahu možných analýz není vyloučeno, že se může stát předmětem dalšího vývoje v rámci diplomové práce.

Literatura

[1] Christian Nagel, e. a.: *Professional C# 4 and .NET 4*. Wiley Publishing, 2010, iSBN 978-0-470-50225-9.

[2] Cruz, I. F.; Tamassia, R.: Graph Drawing Tutorial [online]. `http://www.graphdrawing.org/literature/gd-constraints.pdf`, 1994 [cit. 2010-05-1].

[3] Emden R. Gansner, e. a.: A Technique for Drawing Directed Graphs [online]. `http://www.graphviz.org/Documentation/TSE93.pdf`, 1999 [cit. 2010-05-17].

[4] MacDonald, M.: *Pro WPF in C# 2010 : Windows Presentation Foundation in .NET 4.0*. Paul Manning, USA, 2010, iSBN 978-1-4302-7205-2.

[5] Sugiyama, K.; Misue, K.: Research Report ISIS-RR-94-14E, Graph Drawing by Magnetic-Spring Model. `http://www.iplab.is.tsukuba.ac.jp/m̃isue/publications/techreport/isis-rr-94-14e.pdf`, 1994 [cit. 2010-05-8].

[6] Zander, J.: Jason Zander's WebLog [online]. Why isn't there an Assembly.Unload method? `http://blogs.msdn.com/jasonz/archive/2004/05/31/145105.aspx`, 2004-05-31 [cit. 2010-05-7].

Seznam příloh

Příloha A

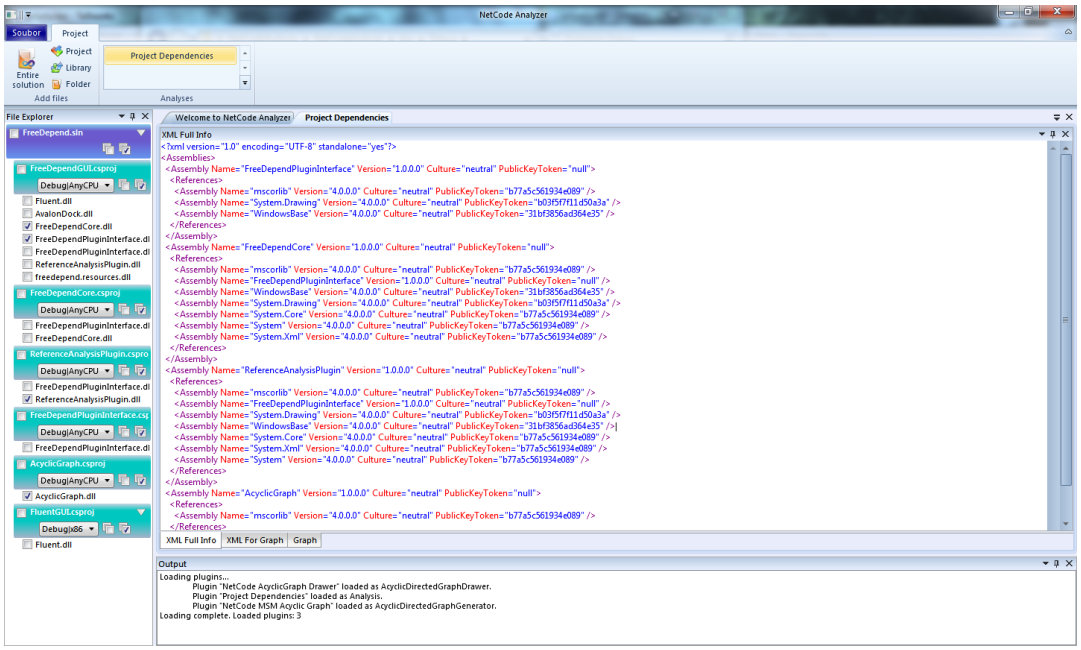
Grafické výstupy

Příloha B

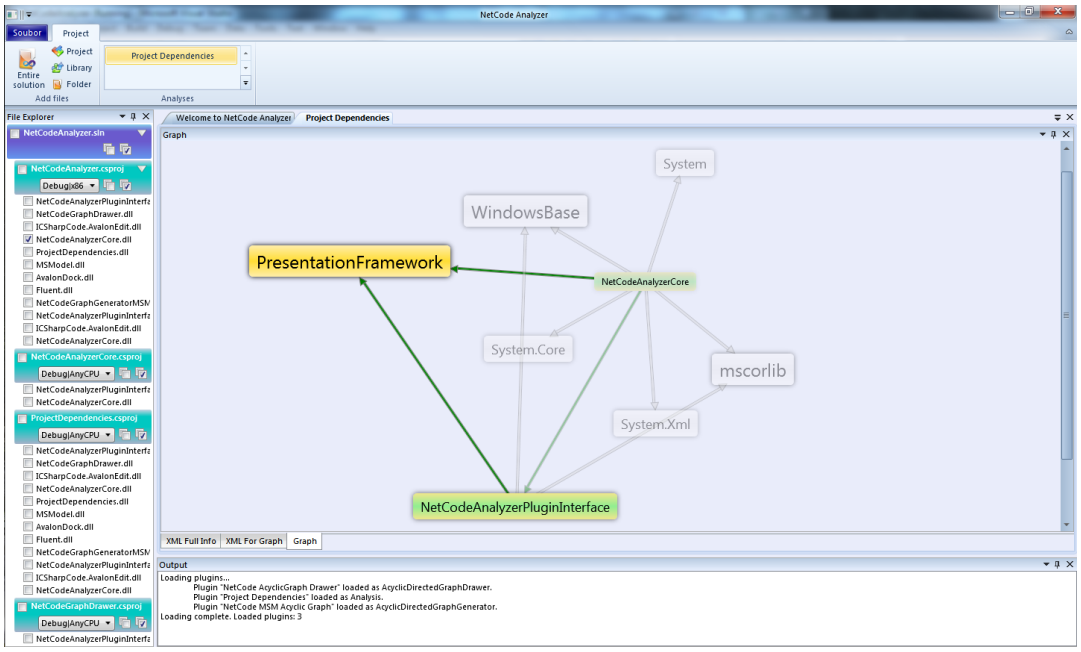
Obsah CD

Příloha A

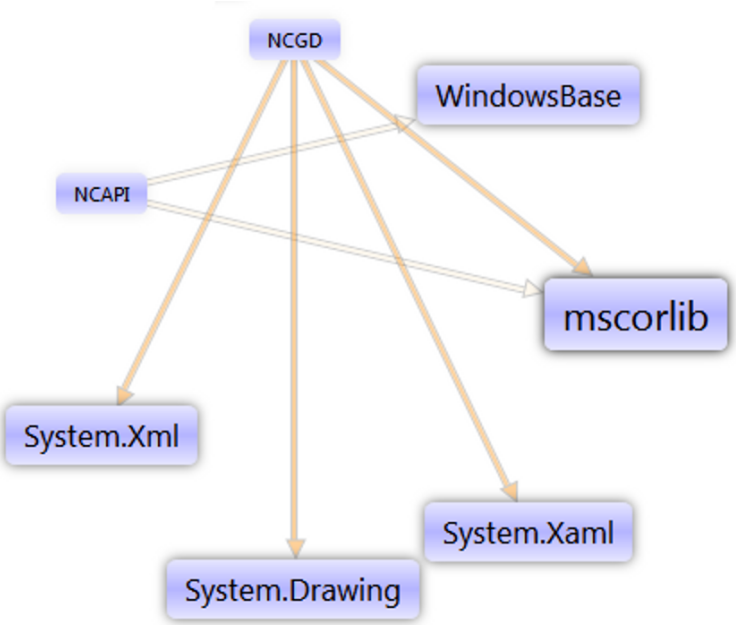
Grafické výstupy



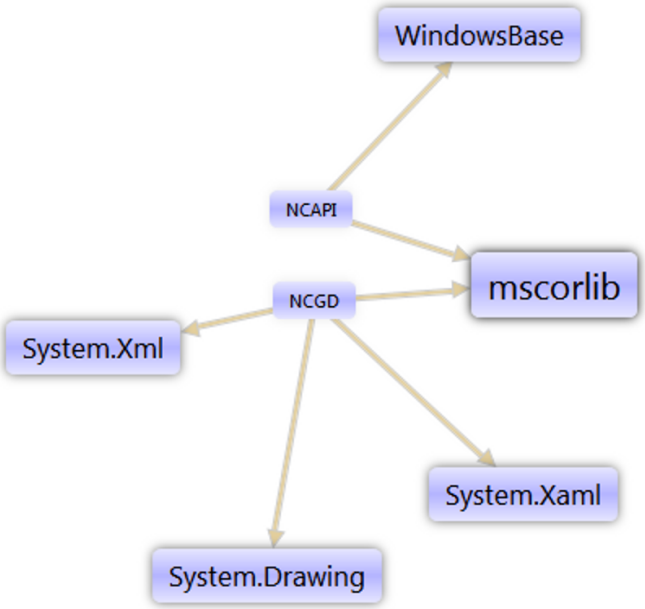
Obrázek A.1: Snímek aplikace zobrazující výstup analýzy v XML formátu.



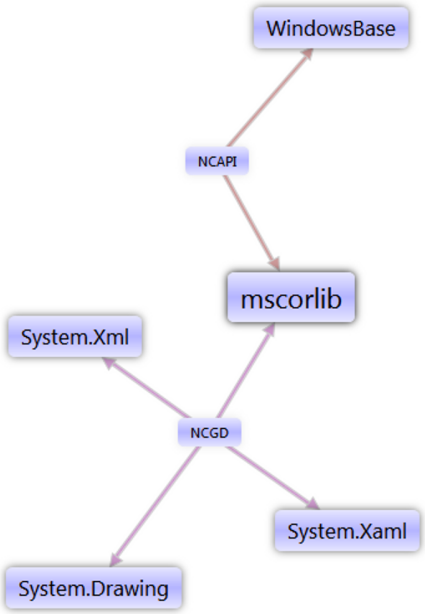
Obrázek A.2: Snímek aplikace zobrazující výstup analýzy v podobě grafu s ukázkou interaktivního zvýraznění.



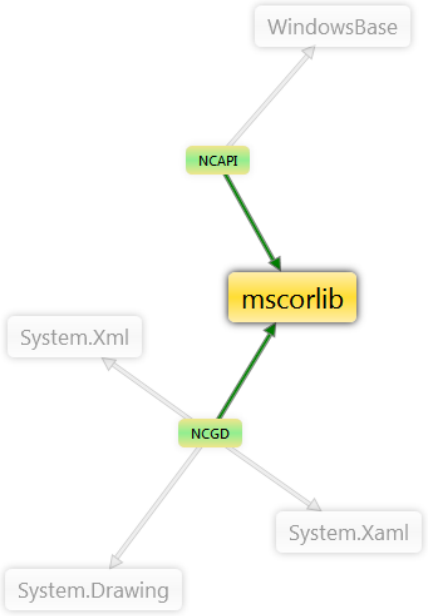
Obrázek A.3: Pozicování grafu: Výchozí rozestavení uzlů v kruhu.



Obrázek A.4: Pozicování grafu: Rozestavení uzlů po prvním kroku.



Obrázek A.5: Pozicování grafu: Finální rozestavení. Tyto pozice jsou konečné, protože všechny části grafu mají nejnižší možnou energii.



Obrázek A.6: Pozicování grafu: Finální rozestavení. Obrázek A.5 s ukázkou interaktivního zvýraznění části grafu.

Příloha B

Obsah CD

- soubor readme.txt
- spustitelná aplikace
- zdrojové kódy
- aplikace pro testování generátoru grafu
- programová dokumentace
- kopie tohoto dokumentu