VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

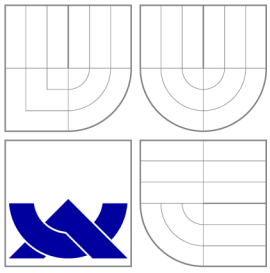
SADA OVLÁDACÍCH PRVKŮ PRO WPF

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

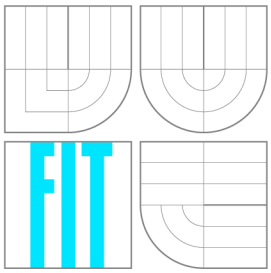
AUTOR PRÁCE
AUTHOR

ONDŘEJ VRŠAN

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SADA OVLÁDACÍCH PRVKŮ PRO WPF

WPF CONTROL LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ VRŠAN
AUTHOR

VEDOUCÍ PRÁCE — Ing. RUDOLF KAJAN
SUPERVISOR

BRNO 2012

Abstrakt

Tato práce se zabývá vytvářením znovupoužitelných ovládacích prvků pro WPF. Srovnává dostupné ovládací prvky na trhu a open source projekty. Popisuje vytvoření vlastní sady ovládacích prvků pro technicky zaměřené aplikace. Součástí práce je demonstrační program, který prezentuje možnosti vytvořených ovládacích prvků. Výsledný projekt je zveřejněn na portálu Codeplex.com

Abstract

This thesis is focused on design reusable WPF user control library. It compares the available control library on the market and open source projects. Describes creating user control library for the technically oriented applications. Part of this work is demonstration program presenting possibilities of developed controls. Project is published on the Codeplex.com portal.

Klíčová slova

WPF, XAML, C♯, .NET framework, Sada ovládacích prvků, Tech4WPF, Knob Control, Gauge Control, Chart Control

Keywords

WPF, XAML, C♯, .NET framework, User Control Library, Tech4WPF, Knob Control, Gauge Control, Chart Control

Citace

Ondřej Vršan: Sada ovládacích prvků pro WPF, bakalářská práce, Brno, FIT VUT v Brně, 2012

Sada ovládacích prvků pro WPF

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Rudolfa Kajana. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Vršan
16. května 2012

Poděkování

Děkuji svému vedoucímu, panu Ing. Rudolfu Kajanovi, za vedení mé práce, inspirace pro návrh ovládacích prvků a užitečné rady.

Obsah

Úvod

Tato práce se zabývá vytvářením znovupoužitelných vizuálních komponent pro návrh GUI[1] aplikací pomocí technologie WPF[2] firmy Microsoft.

Kapitola 1 podává shrnutí přístupu k vytváření GUI a požadavků na jeho vývoj v minulosti. Představuje dnešní technologe a návrhový vzor Model-View-ViewModel.

Kapitola 2 obsahuje přehled současného stavu, komerční produkty dostupné na trhu, open source produkty a jejich zhodnocení.

Kapitola 3 zdůvodňuje výběr ovládacích prvků pro implementaci a popisuje jejich vytvoření. Jednotlivé prvky následně představuje demonstrační aplikace.

V závěru práce jsou zhodnoceny dosažené výsledky a navrhnuty možnosti dalšího pokračování projektu.

[1] *Graphical User Interface* – Grafické uživatelské rozhraní
[2] *Windows Presentation Foundation*

Kapitola 1

Přístup k vytváření prvků GUI

Tato kapitola pojednává o dnešním přístupu k vytváření prvků GUI a představuje technologii WPF. Nejdříve si ale ukážeme, jaká vedla cesta k jejímu vzniku.

1.1 Počátky GUI

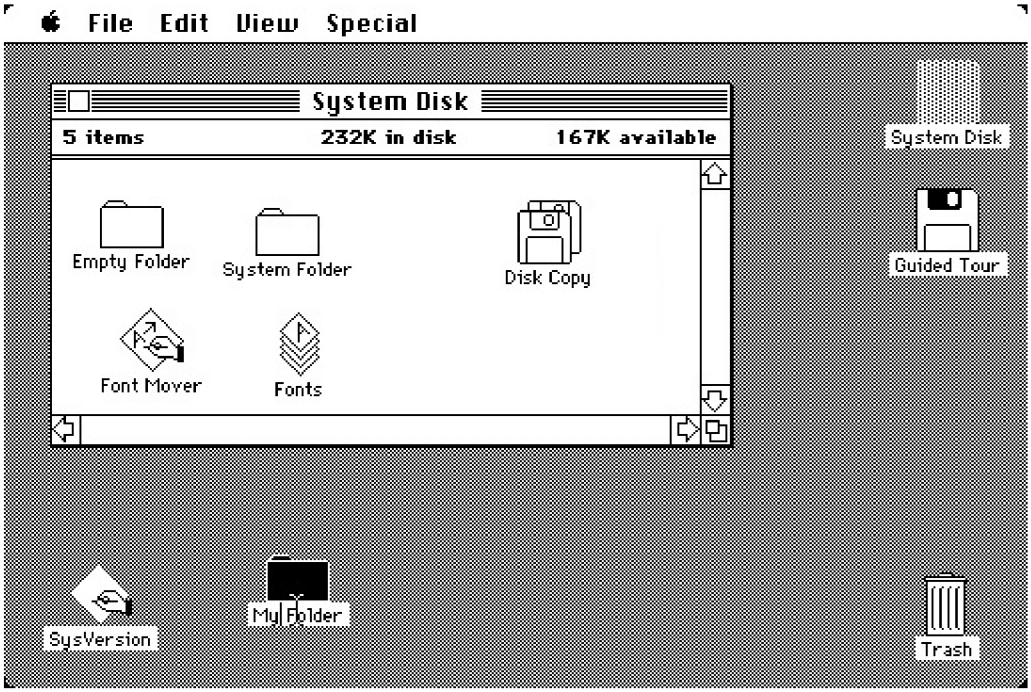
Na *Standford Research Institute* vznikl v 60. letech 20. století *On-Line System* (NLS). Tento systém byl vytvořen v rámci projektu Douglase Engelbarta *Augmentation of Human Intellect* a pracoval s hypertextem pomocí myší ovládaného kurzoru a prací ve více oknech.

Engelbartova práce inspirovala pracovníky z výzkumného centra *Xerox PARC*. V roce 1974[11] Xerox PARC vyvinul osobní počítač *Alto*. Ten měl bitmapovou obrazovku a byl prvním počítačem, který zavedl metaforu *pracovní plochy* a pojem *grafické uživatelské rozhraní* (GUI). Alto nebyl komerčním produktem, přesto se ho vyrobilo několik tisíc kusů a byl po mnoho let používán v Xerox PARC, také v ostatních sídlech firmy a na některých univerzitách. V roce 1981 představil Xerox počítač *Star*. Ačkoli nebyl komerčně úspěšný, významně ovlivnil budoucí vývoj např. počítačů *Apple*[6], *Microsoft* a *Sun Microsystems*[14].

Rozšířit GUI ke koncovým uživatelům se podařilo až firmě Apple s počítačem *Lisa* (vydán roku 1983, cena cca $10.000[11]) a především s cenově dostupnějším následovníkem *Macintosh* (vydán roku 1984, cena cca $2.500[6]). Na vývoji GUI také pracovalo několik členů původního týmu Xerox PARC. Počítače Apple byly první komerčně úspěšné produkty používající víceokenní grafické uživatelské rozhraní. Soubory byly graficky znázorněny jako list papíru a adresáře vypadaly jako složky (viz obr. 1.1/str. 4). Oproti GUI Xerox přidal Apple funkci přesouvání ikon pomocí tažení myši mezi adresáři (funkce *Drag and Drop*) a rozbalovací nabídky.[11].

Grafické uživatelské rozhraní bylo důležitým faktorem popularity počítačů Apple Lisa a Macintosh. Na to reagovala firma Microsoft vývojem svého vlastního GUI a roku 1985 vydáním operačního systému *Windows* verze 1.0. Windows byl nástavbou nad systémem MS-DOS, která zprostředkovávala speciálně napsaným programům využívat menu, ikony a okna[1]. Za zmínku stojí, že na rozdíl od GUI systému Apple se okna v systému Windows 1.0 nemohla překrývat. Překrývání oken přinesl až Windows 2.0[2].

Operační systémy s GUI firmy Apple nebo Microsoft samozřejmě nebyly na trhu samotné. Vedle nich existovaly další operační systémy s grafickým rozhraním od jiných společností, které se však z mnohých důvodů neujaly (např. pro ně nebyly dostupné aplikace nebo je z trhu vytlačila silnější konkurence) a jejich dnešní verze již na trhu nejsou, případně jsou zastoupeny minoritně. Pro úplnost alespoň vyjmenujme některé z nich jako např.



Obrázek 1.1: Plocha počítače Apple Macintosh[14]

Graphical Environment Manager (GEM), *Amiga Workbench*, *NeXTSTEP* nebo *OS/2*[14]. I tyto systémy však používaly principy známé od Apple a Microsoftu, které byly již definovány týmem Xerox PARC jako WIMP pradigma – *Window, Icon, Menu, Pointing device*[1].

Dále také můžeme zmínit *X Window* typické pro Unix-like systémy (také označováno i jako *X11* nebo *X*), které je zajímavé tím, že využívá model klient-server. Na systému X Window běží také dnešní GUI operačního systému *MacOS X* firmy Apple[12].

S příchodem barevných CRT obrazovek se začala objevovat barevná GUI. Příliš mnoho nových principů však nepřibývá, vývoj se přizpůsobuje novým technologiím a aplikačnímu potenciálu počítačů. Vzhled GUI se postupně modernizuje a s narůstajícím výkonem počítačů přibývá animací, zvuků a jiných efektů, které mají za úkol tvořit zpětnou vazbu pro různé akce provedené uživatelem, usnadnit některé úkony nebo indikovat stav právě prováděné operace[4].

1.2 Aplikace s GUI

V době, kdy operační systém s GUI nebyl samozřejmostí, existovaly aplikace, které si grafické rozhraní vykreslily kompletně samy, bez podpory operačního systému. Jako příklad můžeme uvést *Deluxe Paint* nebo *Adobe Acrobat Reader*. Zajímavostí aplikace Acrobat Reader je, že byla schopna běžet jak v grafickém prostředí Windows 3.x, tak v prostředí příkazové řádky MS-DOS. V případě, že aplikace byla spuštěna z příkazové řádky, vykreslila si vlastní rozhraní; pokud byla spuštěna pod Windows, použila GUI tohoto systému[14].

Implementace grafického rozhraní aplikace (uspořádání ovládacích prvků, jejich vzájemná interakce, barvy apod.) mnohdy představuje více než polovinu práce na programovém vybavení. V počátcích tvorby aplikací s grafickým uživatelským rozhraním nebyl kód

[1] Okno, Ikona, Menu, Polohovací zařízení

s vizuální podobou aplikace nijak oddělen od výkonného kódu, který prováděl vlastní činnost, pro kterou byla aplikace vytvořena. Protože neexistovaly specializované nástroje pro popis uživatelského rozhraní, bylo GUI naprogramováno ve stejném jazyku, jako byl programovací jazyk aplikace. Tedy například v jazycích FORTRAN, Pascal, C, nebo i jazyk symbolických adres Assembler[18].

Tento přístup však vyžadoval velkou míru pracnosti a byl neefektivní. Případná úprava celkového designu rozhraní byla pak časově velmi náročná, změna vzhledu podle požadavku trhu či dokonce dle libosti uživatele byla téměř nemožná.

Kód, který obstarával vzhled aplikace, byl úzce svázán s jejím výkonným kódem a daty. Proto kterákoliv změna GUI vyžadovala úpravu aplikace. Protože GUI i samotný výkonný kód měl na starosti jeden člověk, byla tato situace únosná. S rozvojem IT technologií, robustností a komplexností programů, se zvyšoval i počet osob pracujících v týmu na jednom programu. Tým obsahoval odborníky jak na aplikaci samotnou (a problém, který aplikace řeší), tak odborníky na GUI a jeho design. Zde už míra svázanosti kódu pro popis vzhledu aplikace a kódu aplikační logiky, spolu s kódem obstarávajícím manipulaci s daty, nebyla dále únosná.

S příchodem objektově orientovaného programování bylo možno navrhnout architekturu aplikace tak, aby popis vzhledu mohl být od aplikačního kódu oddělen a vyvíjen i v jiném programovacím jazyku.

1.3 Technologie WPF a jazyk XAML

Moderní prostředek v oblasti tvorby uživatelského rozhraní je technologie WPF, která využívá k popisu vzhledu, vazeb a částečně i vizuálního chování jazyk XAML.

1.3.1 WPF – Windows Presentation Foundation

Technologie Windows Presentation Foundation (WPF) navazuje na technologii Windows Forms a je podmnožinou platformy .NET Framework od verze 3.0. WPF používá značkovací jazyk XAML pro vytvoření „uživatelsky bohatého rozhraní“ (RUI[2]) – bohatého svou komplexností, vyspělé, obsahující multimédia, vysoce interaktivní. Díky jazyku XAML jsou od sebe odděleny funkčnost a vzhled aplikace. WPF si klade za cíl sjednotit poutavé uživatelské rozhraní, 2D a 3D grafiku, vektorovou a rastrovou grafiku, animace, *data binding* a audio a video.

Vlastnosti technologie WPF[9]

- **Nezávislost na zobrazovacím zařízení** Velikosti všech prvků jsou definovány v nezávislých jednotkách tzv. dip (angl. *device independent pixel*). Velikost jednotky je definována jako

  1 device independent pixel = 1/96 palce

  Pokud bychom tedy chtěli vědět, jakou výšku v pixelech bude mít prvek o výšce 96 dip (tedy 1 palec), musíme tuto hodnotu vynásobit nastavením DPI[3] daného zobrazovacího zařízení (u monitorů většinou 96 DPI, lze se setkat i se 120 DPI, u tiskáren 300 DPI i více). Pro monitor s nastavením 96 DPI bude tedy prvek vysoký 96 dip

[2] *Rich User Interface*
[3] *Dots Per Inch* – počet bodů na palec

zabírat přesně 96 pixelů, u monitoru s nastavením 120 DPI 120 pixelů a na tiskárně s 300 DPI bude vysoký 300 pixelů. Výška prvku v pixelech se bude měnit dle zobrazovacího zařízení tak, aby byla vždy 1 palec.

- **Vylepšená přesnost** Pro pozici bodu v souřadném systému i hodnoty transformací a průhlednosti je použit datový typ `double`. WPF podporuje *wide color gamut* scRGB[4] a má integrovanou podporu vstupu barev v různých formátech barevného prostoru.
- **Pokročilá grafika a podpora animací** WPF zjednodušuje vývoj grafických scén s animacemi. Programátor se nemusí starat o zpracování scény, renderovací smyčky či bilineární interpolaci.
- **Hardwarová akcelerace** Přes vrstvu DirectX se provádí vektorové vykreslování přes grafický adaptér. Pokud schází technické vybavení, provádí se vykreslování procesorem.

1.3.2 XAML – eXtensible Application Markup Language

XAML je deklarativní značkovací jazyk, který popisuje uživatelské rozhraní v aplikacích postavených na technologii společnosti Microsoft, jako strom souvisejících prvků. Tento jazyk mohou využívat různé nástroje pro návrh UI[5], např. nástroj pro tvorbu vizuálních komponent *Microsoft Expression Blend*. Kromě nástrojů od firmy Microsoft také existují další, jako např. *ZAM 3D*, *XAMLPad* nebo *KaXaml*[7].

Jazykem XAML můžeme popsat vzhled aplikace a definovat návaznost na kód, který obsluhuje události. Vše, co lze napsat pomocí XAML lze napsat i poocí standardních .NET jazyků C♯ nebo Visual Basic .NET. Výhodou XAML je jednoduchost. XAML soubor lze také zkompilovat do binární podoby a použít jej jako *resource* v .NET projektu[13].

1.4 Triviální přístup versus Model-View-ViewModel

Pokud tvoříme aplikaci, která je jednoúčelová, nejspíše sáhneme po nějakém triviálním přímočarém postupu, kdy vytvoříme design aplikace v návrhovém okně, pojmenujeme prvky, definujeme *handlery* a implementujeme chování aplikace (*code behind*). Je to nejspíše nejrychlejší přístup, jak dosáhnout požadovaného cíle.

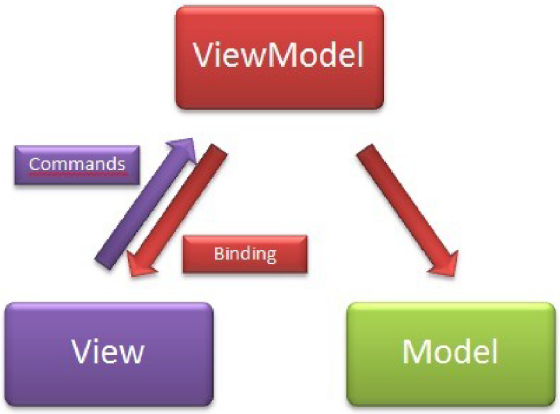
Tento triviální přístup má však následující nevýhody[8]:

- *View* (XAML) a *code behind* jsou silně svázány, tzn. že kód, který popisuje chování okna (*code behind*), odkazuje na pojmenované elementy definované v XAML, a tím je s nimi silně svázán.
- *View* se stává úložištěm dat, ačkoliv by se měl starat pouze o jejich zobrazení.
- Protože *code behind* odkazuje na pojmenované prvky *View* modelu, nelze jej samostatně otestovat při *Unit* testování.
- Velkou nevýhodou je nevyužití možností *data binding*.

Proto pokud pracujeme na složitějším projektu a máme v plánu tento projekt dále rozšiřovat, je vhodné použít některý z návrhových vzorů pro prezentační vrstvu.

[4] viz http://en.wikipedia.org/wiki/ScRGB
[5] *User Interface* – uživatelské rozhraní

Model-View-ViewModel



Obrázek 1.2: MVVM model[3]

Tento návrhový vzor vychází z principu vzorů MVC[6], který navrhl architekt John Gossman. Odděluje vzhled (*View*) a logiku (*ViewModel*) od návaznosti na data (*Model*). Mezi výhody patří dobrá znovupoužitelnost, testovatelnost a udržovatelnost kódu.

Představíme si jednotlivé části modelu[5]:

- **Model** obsahuje reference na zdroj dat. Může existovat i více modelů vedle sebe pro různé logické kontexty.
- **View** reprezentuje grafické rozhraní v jazyce XAML s trochou nezbyteného *code behind*. Zde se používa *Data binding* a obousměrný *Data binding* na *ViewModel*.
- **View-Model** přizpůsobuje *Model* pro *View*. K *Modelu* se neváže přímo, ale přes jeho rozhraní. Obsahuje kód logiky prezentační vrstvy.

[6] *Model View Controller* – http://en.wikipedia.org/wiki/Model_View_Controller

Kapitola 2

Přehled současného stavu ovládacích prvků pro WPF

V současné době je možno k vytváření aplikací použít šablony předchystaných komplexních prvků uživatelského rozhraní, jako jsou např. součásti pro grafy, časové osy, tabulky, formuláře aj.

2.1 Komerční produkty

Z komerčních produktů jsou na trhu dostupné produkty firem *Telerik*[1] a *DevExpress*[2]. Tyto společnosti nabízí bohatou škálu ovládacích prvků jak pro technologii WPF, tak pro starší technologii WinForms. Na obrázku 2.1 na straně 9 je zobrazen ovládací prvek *ChartView* od firmy Telerik. Pokud bychom v naší aplikaci chtěli takový prvek použít, zaplatíme za rozsáhlou knihovnu ovládacích prvků pro WPF $999 (samostatný ovládací prvek si koupit nelze). U konkureční firmy DevExpress bychom za podobnou knihovnu zaplatili o $100 méně[3].

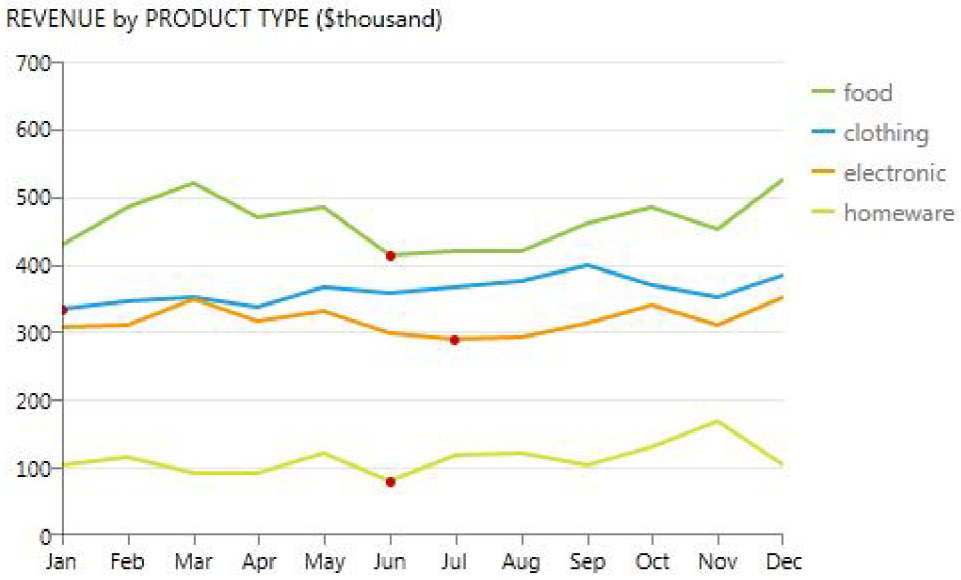
2.2 Open Source a Free produkty

Z OpenSource produktů se můžeme setkat s *Dynamic Data Display* (neaktualizován od 4/2009) nebo *WPF Toolkit* (poslední aktualizace 2/2010). Z *free* produktů můžeme použít *VisiBox Charts*, který ale má ve této verzi vodoznak.

2.3 Zhodnocení

Trh nabízí řadu komerčních produktů, které jsou kvalitní a disponují širokou škálou možností změn vzhledu a chování ovládacího prvku. Jejich cena ale není nízká. Pro vývojáře nekomerční aplikace není myslitelné nakoupit celou sadu ovládacích prvků pro vývoj programu negenerujícího zisk. Jako východisko se nejeví ani *free* produkt, který je často jen „osekanou“ placenou verzí nebo je označen vodoznakem, který lze odstranit po zaplacení určitého poplatku.

[1] viz http://www.telerik.com
[2] viz http://www.devexpress.com
[3] aktuální ceny v květnu 2012

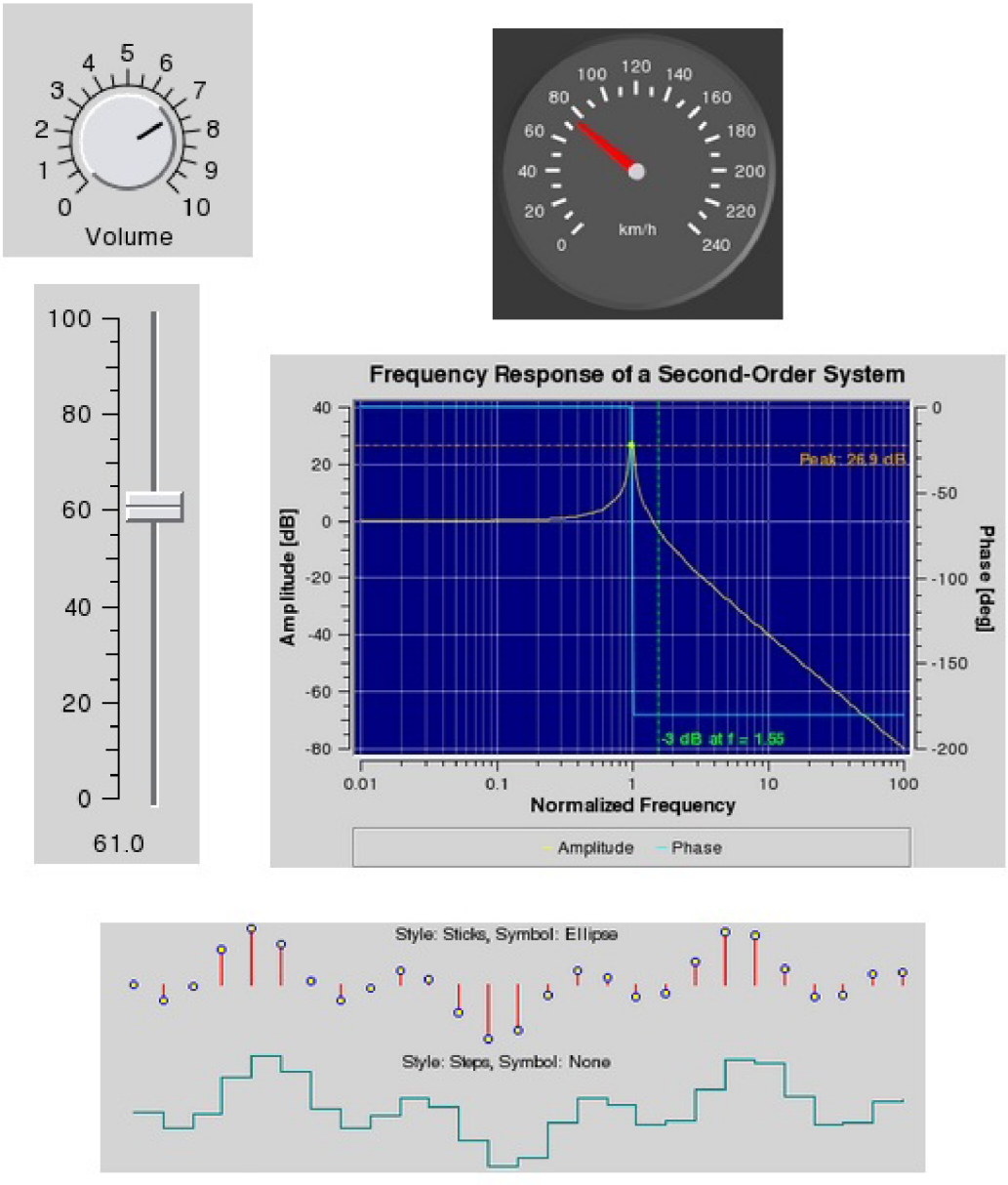


Obrázek 2.1: Telerik *ChartView*

Variantou zůstávají tedy open source produkty nebo vlastní vytvoření ovládacího prvku. Když nyní pomineme tvorbu vlastního, a podíváme se na open source produkty, zjistíme, že vývoj často stagnuje a prvky nenabízí mnoho možností.

Pro technickou aplikaci, cílenou např. na měřící techniku propojenou s počítačem, nenalezneme téměř žádné open source ovládací prvky. Jediným hojně dostupným prvkem majícím využití v technické aplikaci, jsou grafy. Prvky jako otočné knoflíky nebo ručičkové indikátory hodnot nalezneme jen zřídka. Tyto nabízí např. sada *Qwt - Qt Widgets for Technical Applications*[4] (viz obr. 2.2/str. 10). Jak již název napovídá, jedná se o prvky pro aplikace napsané ve frameworku Qt. Pro .NET framework však takové prvky chybí.

[4] viz http://qwt.sourceforge.net/



Obrázek 2.2: Ovládací prvky z knihovny *Qwt*

Kapitola 3

Tech4WPF

V případě, že se rozhodneme vyvinout nekomerční technickou aplikaci, najdeme málo dostupných open source ovládacích prvků, které bychom mohli použít. Pro framework Qt existuje sada ovládacích prvků *Qwt - Qt Widgets for Technical Applications*[1]. K této sadě je dostupný i port na .NET framework – open source projekt *QWT for .NET*. Protože se ale jedná o port, není využito všech předností .NET frameworku a možností WPF a Data Bindingu.

Rozhodl jsem se tedy, že vytvořím open source projekt **Tech4WPF**, který si klade za cíl nabídnout ovládací prvky pro technické aplikace (podobně jako sada prvků Qwt) vyvíjené v .NET framework, postavené na technologii WPF.

Každá technická aplikace, která pracuje s měřícími přístroji, potřebuje pro komunikaci s uživatelem ovládací prvky, které indikují stav nějakého čidla. Tento stav může zprostředkovat buď ručičkový ukazatel, nebo graf. Uživatel také často potřebuje mít možnost nastavovat hodnoty vstupů, které mají přesně dané hranice. K těmto účelům může sloužit např. otočný knoflík. Proto jsem zvolil následující prvky jako základ sady ovládacích prvků **Tech4WPF**:

- Knob Control – Otočný knoflík
- Chart Control – Graf
- Gauge Control – Ručičkový ukazatel

3.1 Architektura

Knihovna ovládacích prvků **Tech4WPF** obsahuje čtyři základní prostory jmen:

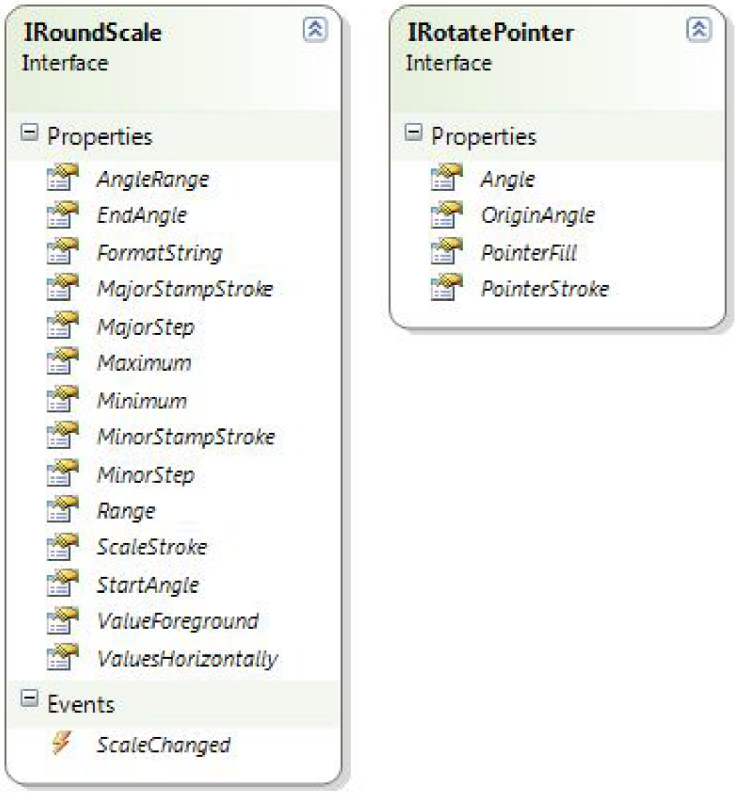
- `Tech4WPF.Common`
- `Tech4WPF.GaugeControl`
- `Tech4WPF.KnobControl`
- `Tech4WPF.ChartControl`

[1] viz http://qwt.sourceforge.net/

3.1.1 Namespace `Tech4WPF.Common`

Tento prostor jmen obsahuje definice rozhraní, společnou implementaci částí pro prvky **Knob Control** a **Gauge Control**, a také pomocné metody, které jsou použity v implementaci jednotlivých ovládacích prvků.

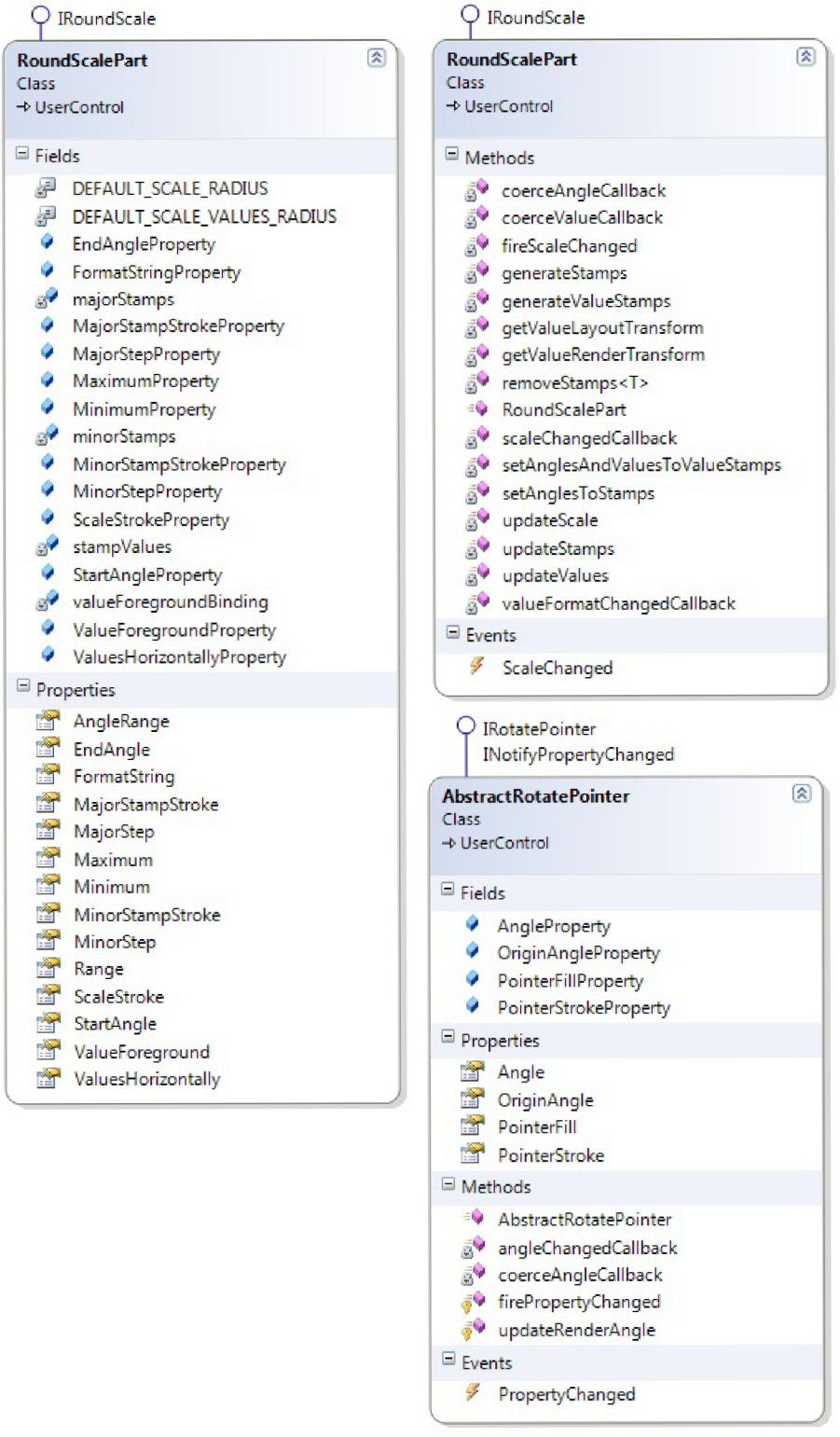
Rozhraní `IRotatePointer` a `IRoundScale` obsahují seznam vlastností, které musí splňovat otočný ukazatel a kruhová stupnice (viz obr. 3.1/str. 12).



Obrázek 3.1: Rozhraní `IRoundScale` a `IRotatePointer`

Ukazatele prvků **Knob Control** a **Gague Control** sdílí implementaci rozhraní ve třídě `AbstractRotatePointer` (viz obr. 3.2/str. 13). Zde se přímo nabízí, aby třída `AbstractRotatePointer` byla deklarována jako třída abstraktní. V případě, že máme několik tříd, které se zavázaly implementovat stejné rozhraní, a tato implementace rozhraní je společná (čili má stejný kód), je vhodné tento kód umístit do třídy nadřazené, ze které ostatní třídy tuto implementaci zdědí. Vyhneme se tak duplicitnímu kódu. Taková nadřazená třída ale zajisté neimplementuje ostatní detaily (ani nemůže), které odlišují jednotlivé třídy od sebe. Je pak ale správné mít možnost vytvořit instanci takové „neúplné“ třídy? Zajisté ne. K těmto účelům je vhodné v C♯ použít právě abstraktní třídu. U abstraktní třídy totiž nelze vytvořit její instanci. Dále nám také dovoluje deklarovat metody bez těla (ekvivalent k čistě virtuálním metodám z C++), jež třída, která dědí, musí implementovat.

Nabízí se tedy deklarovat třídu `AbstractRotatePointer` jako abstraktní, která bude dědit od `UserControl` a ponese společnou implementaci rozhraní `IRotatePointer`. Následně z této abstraktní třídy dědit do tříd implementujících konkrétní prvek, jako je v případě **Tech4WPF** `KnobPart` nebo `PointerPart`. V XAML těchto prvků pak nahra-



Obrázek 3.2: Třídy `AbstractRotatePointer` a `RoundScalePart`

dit root tag `<UserControl>` za tag `<my:AbstractRotatePointer>` (kde `my` je deklarované XML namespace, kde se vyskytuje třída `AbstractRotatePointer`).

Tato architektura využívající abstraktní třídu má však jednu nevýhodu. Designer Visual Studia nám nebude schopen vykreslit náhled XAML zápisu prvku. Přesto, že nikde v kódu nevytváříme instanci abstraktní třídy, dostaneme chybovou hlášku „Nelze vytvořit instanci abstraktní třídy“. Designer Visual Studia totiž při vykreslení prvku potřebuje vytvořit instanci zděděné třídy. K změnám prováděným v XAML bychom tak neměli zpětnou vazbu. Přes to, že designer produkuje tuto chybu, lze aplikaci zkompilovat a spustit. Při běhu aplikace totiž není zapotřebí vytvářet instanci základní třídy (v tomto případě abstraktní).

Nabízí se řešení pomocí preprocesoru[2], které se pokusím nastínit pomocí následujícího úryvku kódu:

```
#if DEBUG
public class MyBaseControl : UserControl
{
#else
public abstract class MyBaseControl : UserControl
{
#endif
    //...
#if DEBUG
    protected virtual void OverrideMe() { }
#else
    protected abstract void OverrideMe();
#endif
    //...
}
```

Při kompilování *debug* verze (a tím i při vykreslování designeru) bude třída zkompilována bez klíčového slova `abstract` a abstraktní metody budou nahrazeny virtuálními. Při kompilování *release* verze bude zkompilována abstraktní třída s abstraktními metodami.

Toto řešení se mi nejevilo jako nejvhodnější, protože je snížena přehlednost kódu (na první pohled není patrné, co je vlastně zamýšleno), a také protože při debugování se pracuje s jiným kódem, než který je zkompilován jako *release* verze. Proto jsem přistoupil k vlastnímu řešení, které nepoužívá makra preprocesoru. Třídu jsem deklaroval bez klíčového slova `abstract`. Bohužel konstruktor této třídy musí zůstat veřejně přístupný (tedy `public`) kvůli designeru, takže je zde riziko vytvoření instance této neúplné třídy. Považuji jej ale za menší nevýhodu, než v případě použití maker preprocesoru, která můžou při hojnějším použití (více abstraktních funkcí) zatemňovat kód a být zdrojem skrytých chyb (aplikace používá jiný kód v *release* než v *debug* verzi).

Metody, které bylo vhodné v případě použití abstraktní třídy deklarovat také jako abstraktní, jsem deklaroval jako virtuální a do jejich těla jsem napsal kód, vyhazující výjimku s popisem, že tato metoda musí být předefinována. Metodu jsem doplnil i dokumentačním XML komentářem, který taktéž upozorňuje na nutnost ji předefinovat. Tím je zajištěna implementace této metody v dědící třídě. Zde je příklad tohoto způsobu náhrady abstraktní metody ve třídě `AbstractRotatePointer`:

[2] viz http://social.msdn.microsoft.com/Forums/en-US/winformsdesigner/thread/b284d11a-1a9d-4ca5-b3cc-d58dd9e4f5cf/

```
/// <summary>
/// Updates the render angle of the pointer. Has to be overridden.
/// </summary>
protected virtual void updateRenderAngle()
{
    throw new NotImplementedException(
        "updateRenderAngle() has to be overriden");
}
```

Kruhová stupnice, kterou představuje třída `RoundScalePart` (viz obr. 3.2/str. 13), se skládá z vizuálních prvků `Arc` a `Path`. Prvky `Path` představují šablony značek stupnice (*minor stamp* a *major stamp*) a `Arc` stupnici samotnou. Šablony značek stupnice mají nastavenou pozici a střed pro rotační transformaci. Jejich vlastnost `Visibility` je však nastavena na hodnotu `Hidden`, protože neplní funkci značky, ale pouze šablony pro její vytvoření. Při skutečném vykreslování stupnice se následně pomocí jejích *properties* `StartAngle`, `EndAngle`, `Minimum`, `Maximum`, `MinorStep` a `MajorStep` vykreslí stupnice a dopočítají se počty značek a jejich umístění. Tyto značky se vytváří právě podle zmiňovaných šablon a nastavuje se jim transformační úhel, o který se pootočí na své místo. Dále jsou stupnici dopočítány a vykresleny hodnoty. Tato implementace je společná pro oba prvky **Knob Control** i **Gauge Control**.

Třídy `AbstractRotatePointer` a `RoundScalePart` mají své veřejné *property* implementovány pomocí *dependency property*, které zajišťují *callback* volání metod pro kontrolu zadaných hodnot a vynucení překreslení prvku.

Způsob vyvolávání události v knihovně **Tech4WPF** se liší od klasického postupu při vyvolání události. Ukážeme si příklad na události `PropertyChanged` třídy `AbstractRoundPointer`. Vyvolávání ostatních událostí v jiných třídách knihovny **Tech4WPF** se děje pak stejným způsobem. Často najdeme vyvolání události kódem podobným tomuto:

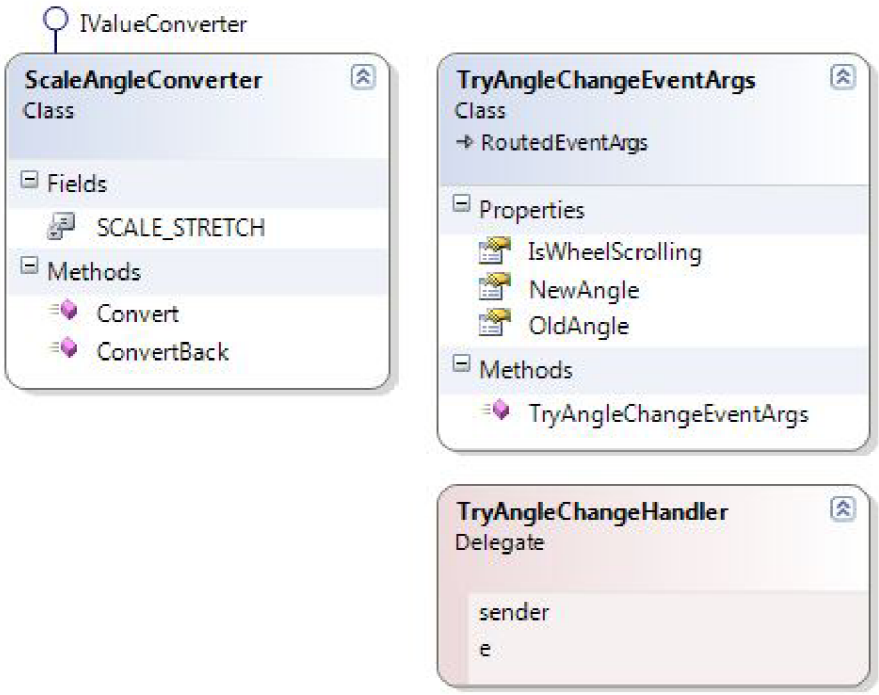
```
protected void firePropertyChanged(string property)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(property));
    }
}
```

Toto řešení je vhodné, ale při použití ve vícevláknové aplikaci může docházet k vyhození výjimky `NullReferenceException`. Pokud totiž po proběhnutí podmínky ověření nerovnosti s `null` dojde v jiném vlákně k odhlášení odběru události, vyvolání delegáta události `PropertyChanged` skončí výše zmíněnou výjimkou.

Pokud si však zachováme lokální kopii delegáta, bude náš kód bezpečný[3]. V případě, že by po otestování lokální kopie delegáta na `null` došlo k odhlášení odběru události `PropertyChanged`, nemá to vliv na tuto lokální kopii, protože ta byla vytvořena ještě před odhlášením odběru. Volání lokální kopie `propertyChanged` je tedy bezpečné a neskončí výjimkou.

Nevýhoda tohoto přístupu je, že *handler* události bude zavolán i po odhlášení jejího odběru. Přesto jsem toto řešení zvolil kvůli jeho bezpečnosti vzhledem k výjimce `NullReferenceException`.

[3] viz http://www.atulverma.com/2010/05/inotifypropertychanging-and_04.html



Obrázek 3.3: Třídy `ScaleAngleConverter` a `TryAngleChangeEventArgs` s delegátem `TryAngleChangeHandler`

Způsob vyvolávání událostí v knihovně **Tech4WPF** ilustruje následující kód:

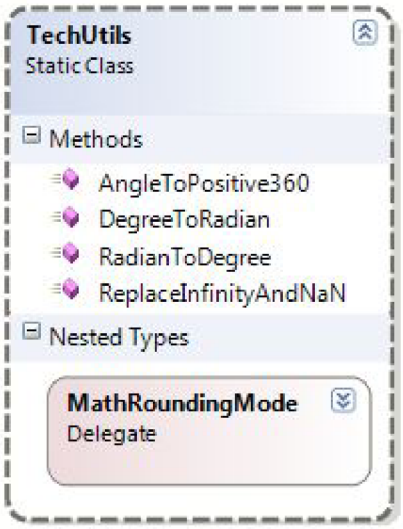
```
protected void firePropertyChanged(string property)
{
    //Maintaining the temporary copy of event to avoid race condition
    propertyChangedEventHandler propertyChanged = PropertyChanged;
    if (propertyChanged != null)
    {
        propertyChanged(this, new PropertyChangedEventArgs(property));
    }
}
```

Třída `ScaleAngleConverter` (viz obr. 3.3/str. 16) implementuje rozhraní `IValueConverter`, tzn. poskytuje funkcionalitu konverze hodnot úhlu pro prvek `RoundScalePart`. Protože značky pro *major stamp* jsou vykreslovány pod vlastní kružnicovou výsečí a mají určitou tloušťku, v krajních hodnotách přesahují polovinou své tloušťky okraj kružnicové výseče. A jelikož ovládací prvek umožňuje nastavit rozdílné barvy kružnicové výseče stupnice (`ScaleStroke` *property*) a hlavních značek (`MajorStampStroke` *property*), bylo by toto přesahování viditelné a vizuálně rušivé. Proto je nutno počáteční a koncový úhel protáhnout o 1,8 ° v obou směrech, aby se kružnicová výseč vykreslila až po okraj značky *major stamp*. V závislosti na tom, zda se jedná o začátek či konec stupnice, je protažení přičítáno nebo odčítáno, v závislosti parametru `parameter`.

Třída `TryAngleChangeEventArgs` dědí od `RoutedEventArgs` a poskytuje funkcionalitu při pokusu změnit úhel natočení. Je využívána pro prvek **Knob Control**, kde je při změně

úhlu natočení otočného knoflíku pomocí myši potřeba validovat, zda se na tento úhel může knoflík natočit (a neukazoval mimo stupnici). *Property* této třídy mají privátní *setter*, čímž je zajištěno bezpečné použití, tzn. že se nastaví pouze při vyvolání události pomocí konstruktoru a z venčí jsou pouze pro čtení. Jak můžeme vidět na obrázku 3.3 na straně 16, tato třída obsahuje pouze tři *property* informující o původním a novém úhlu a o tom, zda natočení na nový úhel bylo realizováno pomocí kolečka myši.

Delegát `TryAngleChangeHandler` 3.3 deklaruje podobu metody, která obsluhuje událost s argumenty objektu `TryAngleChangeEventArgs`.



Obrázek 3.4: Třída `TechUtils`

Statická třída `TechUtils` (viz obr. 3.4/str. 17) poskytuje pomocné metody, které jsou využívané při implementaci jednotlivých ovládacích prvků. Protože se metody v této třídě mohou hodit i vývojářům používajícím ovládací prvky z knihovny **Tech4WPF**, rozhodl jsem se tuto třídu deklarovat jako `public`. Tato třída poskytuje metody pro převod úhlu mezi stupni a radiány, převod libovolného úhlu ve stupních ekvivalentní kladný úhel menší než 360 ° a metodu pro kontrolu hodnoty typu `double` a nahrazení hodnot `Infinity` nebo `NaN` nulou.

3.1.2 Namespace `Tech4WPF.GaugeControl`

Třída `PointerPart` (viz obr. 3.5/str. 19) dědí od třídy `AbstractRotatePointer` implementaci rozhraní a přepisuje „abstraktní“ *callback* metodu `updateRenderAngle()` z nadřazené třídy (tato metoda překresluje natočení prvku). Třída představuje ručičku ukazatele pro stupnici. V XAML je definována vizuální podoba ukazatele, který na stupnici prvku **Gauge Control** ukazuje hodnotu. Design ukazatele byl vytvořen v prostředí *Microsoft Expression Blend*, ve kterém je možné kromě přímé editace XAML zápisu i vizuálně kreslit křivky, elipsy atd.

Třída `GaugeControl` (viz obr. 3.5/str. 19) reprezentuje prvek **Gauge Control**. Skládá dohromady součásti `RoundScalePart`[4] a `PointerPart`, zapouzdřuje jejich *property* a při-

[4] viz oddíl 3.1.1 Namespace `Tech4WPF.Common` na straně 12

dává zobrazení aktuální ukazované hodnoty, *snap mode* a popisek prvku. Při změně ukazované hodnoty vyvolává událost `ValueChanged`. V konstruktoru jsou nastaveny *handlery* pro odběr událostí stupnice a ukazatele, zároveň je pro ukazatel nastaven úhel počátku stupnice (nastavena *property* `OriginAngle`). Třída `GaugeControl` implementuje rozhraní `INotifyPropertyChanged`, což umožňuje upozornit okolní objekty, že se změnila *property* objektu,- v tomto případě se jedná o upozornění změny *property* `Value`.

Při použití v XAML je obsah mezi otevíracím a zavíracím tagem použita pro *property* `Label` (tedy popisek ovládacího prvku) a výchozí událostí je `PropertyChanged`. Toho je docíleno v deklaraci třídy `GaugeControl` pomocí atributů `ContentProperty` a `DefaultEvent`, které určují výchozí událost a použití obsahu mezi tagy.

3.1.3 Namespace `Tech4WPF.KnobControl`

Třída `KnobPart` (viz obr. 3.6/str. 20), podobně jako třída `PointerPart` z *namespace* `Tech4WPF.GaugeControl`, dědí od třídy `AbstractRotatePointer` implementaci rozhraní a přepisuje „abstraktní“ *callback* metodu `updateRenderAngle()` z nadřazené třídy. Tato metoda překresluje natočení prvku. Třída představuje otočný knoflík, který má dané hranice svého natočení. V XAML je definována vizuální podoba knoflíku, který na stupnici prvku **Knob Control** ukazuje hodnotu. Design ukazatele byl vytvořen v prostředí *Microsoft Expression Blend*, jako tomu bylo u prvku `PointerPart`. Při překreslení natočení knoflíku není však otáčeno celým prvkem, ale pouze jeho ryskou `arrow`. Výsledný efekt vytváří dojem otáčení celého knoflíku.

Třída `KnobPart`, narozdíl od třídy `PointerPart`, implementuje navíc obsluhu událostí klepnutí a tažení myši, dvojklik myší, rolování kolečkem a vyvolává událost `TryAngleChange`, která je vyvolána vždy, když dochází k pokusu změnit úhel natočení knoflíku. Obsluha událostí myši byla částečně převzata z [10].

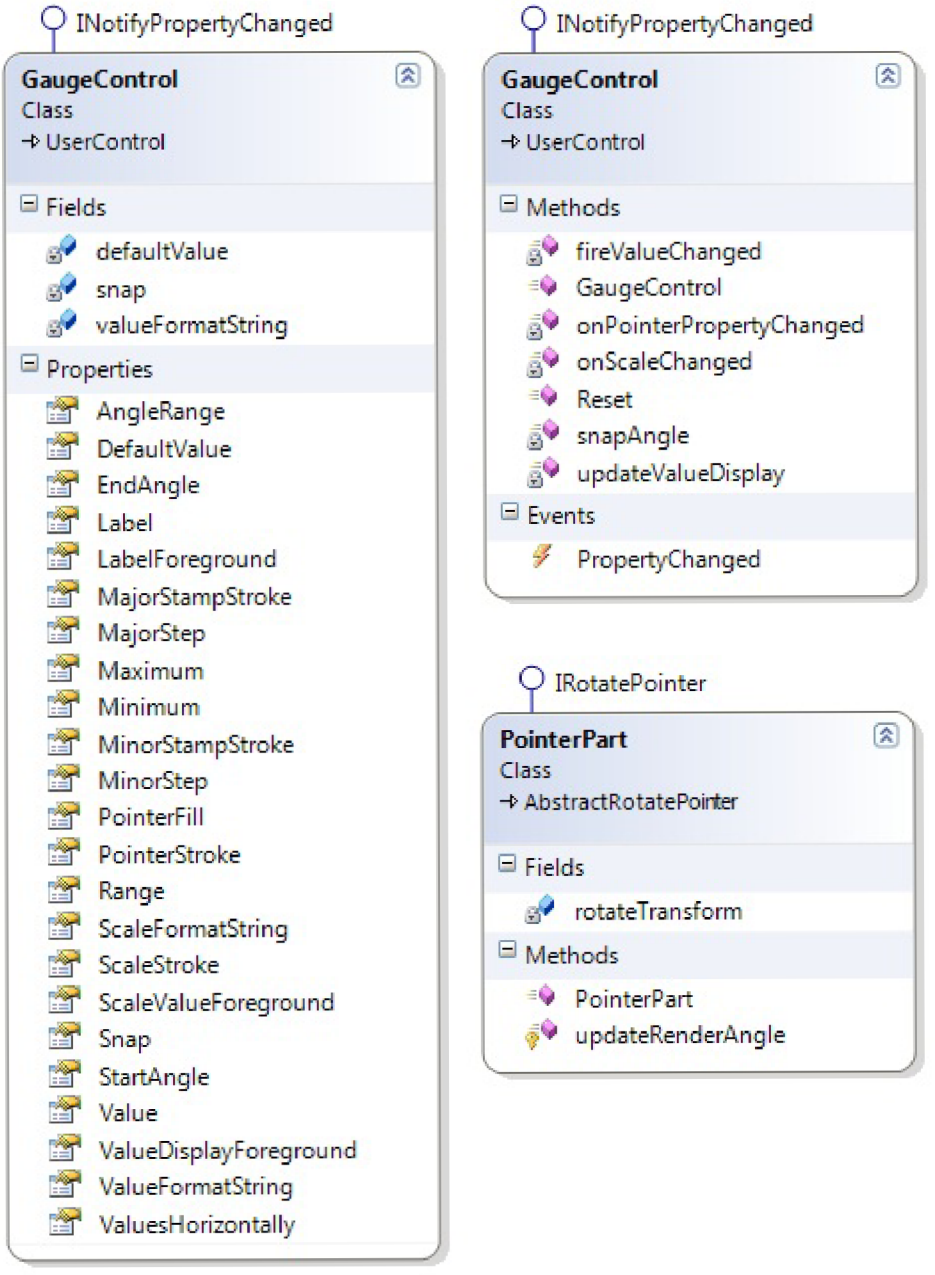
Dalším rozdílem je přítomnost statického konstruktoru. Statický konstruktor se používá pro inicializaci statických dat a volá se automaticky vždy před prvním vytvořením instance třídy. Ve třídě `KnobPart` slouží statický konstruktor k přepsání zděděných statických dat. *Dependency poperty* má pro výplň ukazatele v nadřazené třídě výchozí hodnotu červenou, kdežto u knoflíku jsem požadoval výchozí hodnotu zelenou. Přepsání této výchozí hodnoty se děje právě ve statickém konstruktoru.

Výhodou tohoto řešení je sdílená implementace `dependency property`. Kód se neduplikuje jen z důvodu jiných výchozích hodnot. Menší nevýhodou tohoto přístupu je, že designer Visual Studia nebere v potaz tento statický konstruktor a knoflík zobrazí s červenou výplní, zděděnou ze třídy `AbstractRotatePointer`. Při spuštění aplikace se však vše chová správně a knoflík má ve výchozím stavu zelenou výplň.

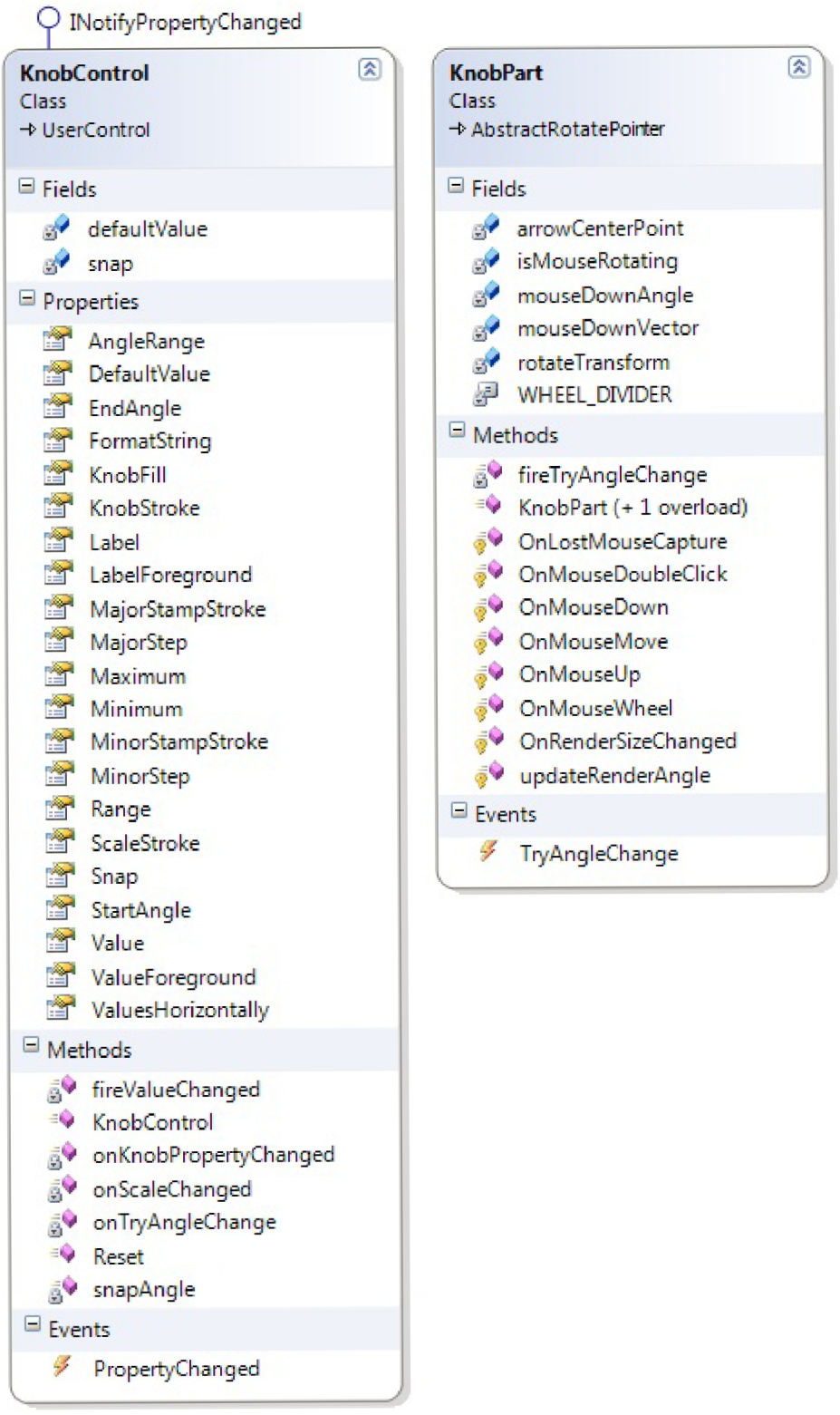
Třída `KnobControl` (viz obr. 3.6/str. 20) reprezentuje prvek **Knob Control**. Podobně jako třída `GaugeControl` u prvku **Gauge Control**, skládá třída `KnobControl` dohromady části `RoundScalePart`[5] a `KnobPart`, zapouzdřuje jejich *property* a přidává *snap mode* a popisek prvku. Při změně hodnoty vyvolává prvek událost `ValueChanged`. V konstruktoru jsou nastaveny *handlery* pro odběr událostí stupnice a knoflíku, zároveň je také pro knoflík nastaven úhel počátku stupnice (nastavena *property* `OriginAngle`). Implementuje taktéž rozhraní `INotifyPropertyChanged`, aby mohla upozornit na změnu *property* `Value`.

Třída `KnobControl` dále odebírá událost `TryAngleChange` třídy `KnobPart`. Při pokusu změnit úhel knoflíku se zjistí, zda hodnoty nepřekračují dané meze, nebo pokud je zapnut

[5] viz oddíl 3.1.1 Namespace `Tech4WPF.Common` na straně 12



Obrázek 3.5: Třídy `PointerPart` a `GaugeControl`



Obrázek 3.6: Třídy `KnobPart` a `KnobControl`

snap mode, zaokrouhlují se hodnoty k nejbližším *major stamps*. V případě překročení mezí je nastaveno minimum nebo maximum v závislosti na pozice myši či směru překročení hodnot.

Při použití v XAML je stejně jako u prvku **Gauge Control** obsah mezi otevíracím a zavíracím tagem použita pro *property* `Label` (tedy popisek ovládacího prvku) a výchozí událostí je `PropertyChanged`. Toho je docíleno v deklaraci třídy `GaugeControl` pomocí atributů `ContentProperty` a `DefaultEvent`, které určují výchozí událost a použití obsahu mezi tagy.

3.1.4 Namespace `Tech4WPF.ChartControl`

Implementace ovládacího prvku **Chart Control** je částečně převzata ze zdrojových kódů ke knize *Practical WPF Charts and Graphics*[17], konkrétně zdrojových kódů pro *LineChartControl*. Převzata je implementace typů symbolů, jejich vykreslení, implementace grafu a legendy.

Kód byl však značně refaktorován, dlouhé nepřehledné metody byly rozděleny do dílčích metod, a byla pozměněna hierarchie tříd. Dále byl přidán vykreslovací typ čáry `Steps`, symbol `Bar` pro vykreslení sloupcového grafu a také možnost nastavit výplň pod křivkou grafu.

Při zkoumání kódu jsem zjistil, že na rozdíl od knihy – která poměrně kvalitně pojednává o teorii transformačních matic, vykreslování 2D a 3D grafů – je zdrojový kód kvalitativně horší. Při použití původního ovládacího prvku `LineChartControl` bylo nutno hlídat změnu rozměrů tohoto prvku a při této změně znova naplnit ovládací prvek daty, protože docházelo ke smazání původních dat. Toto mi přišlo z hlediska vývojáře, který má takový prvek použít, značně nepraktické. Způsob použití takového grafu by dle mého názoru měl být v tom, že se nastaví hodnoty a požadovaný styl vykreslení a graf je zobrazuje i při změnách rozměrů, dokud nejsou zadána jiná data.

Dalším nedostatkem bylo, že po vložení dat (jednotlivých bodů grafu) byly tyto body pozměněny. V důsledku to tedy znamená, že pokud programátor nastavil, že se má vykreslit XY bod $(1, 1)$, po vykreslení a znovu-přečtení tohoto bodu vloženého do ovládacího prvku nebyl tento bod $(1, 1)$, ale úplně jiná hodnota (např. $(110, 329)$) přepočítaná tak, aby dávala smysl pro vnitřní funkci vykreslování grafu. Poroto také hodnoty byly při změně rozměrů prvku vymazány, protože je nebylo možno dále znovu přepočítat pro nový rozměr prvku.

Hodnoty jako `NaN` nebo `Infinity` nebyly ošetřeny, takže vykreslení grafu s těmito hodnotami způsobovalo vyvolání výjimky.

Testování rozměru *canvasu* grafu na hodnotu `NaN` probíhalo způsobem, který ilustruji kódem níže. V nižších verzích .NET frameworku nebo v anglickém prostředí kód nejspíše fungoval, ale v nynější aktuální verzi 4.0 a jiném národním prostředí (česká verze Windows 7) tento kód nedělal to, co bylo zamýšleno.

```
if (canvas.Width.ToString() == "NaN")
{
    canvas.Width = 300;
}
```

Jak můžeme vidět, hodnota šířky *canvasu* (*property* `Width`) byla metodou `ToString()` převedena na řetězec a porovnána s řetězcem `"NaN"`. Pomineme-li tento chybný přístup k testování na hodnotu `NaN`, zdá se, že by kód mohl fungovat. Lokalizované české prostředí však při volání metody `ToString()` na objekt `double` obsahující hodnotu `NaN` nevrací

`"NaN"`, ale řetězec `"Není číslo"`. Rovnost tedy platit nebude a kód, který se v případě hodnoty `NaN` měl vykonat, se přeskočí.

Při opravě tohoto kódu jsem nejspíše narazil na to, co autora vedlo k řešení pomocí metody `ToString()`. Úprava kódu do následující podoby totiž nedávala očekávané výsledky.

```
if (canvas.Width == double.NaN)
{
    canvas.Width = 300;
}
```

Přesto, že hodnota *property* `Width` je `NaN`, podmínka platit nebude. Nejdná se ale o chybu[15]. Konstanta `double.NaN` slouží totiž pouze pro inicializaci výchozích hodnot, nikoliv pro porovnávání. Pro účely testování na hodnotu `NaN` slouží statická metoda `double.IsNaN(double d)`, která vrací `true`, pokud má parametr hodnotu `NaN`.

Správné řešení použité v prvku **Chart Control** tedy vypadá takto:

```
if (double.IsNaN(canvas.Width))
{
    canvas.Width = 300;
}
```

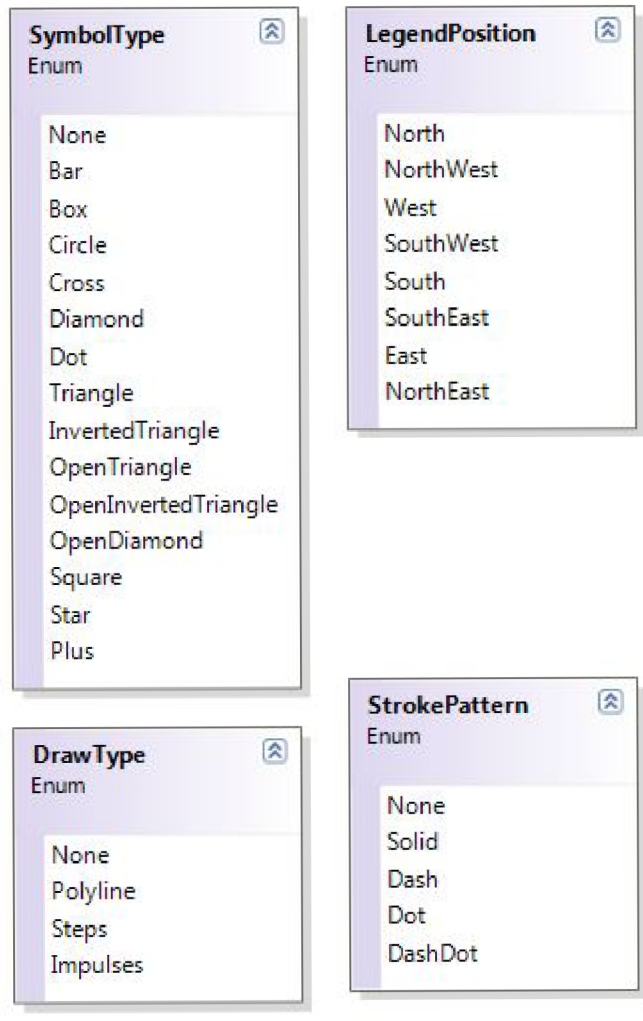
Výše zmíněné nedostatky jsem tedy opravil a přepracoval. S body pro vykreslení grafu v ovládacím prvku **Chart Control** není manipulováno, takže přečtení bodů vložených pro vykreslení vrátí očekávané hodnoty (tedy nezměněné). Přepočítané hodnoty pro vykreslení grafu jsou uloženy interně v ovládacím prvku. To má mimo jiné tu výhodu, že při změně rozměrů grafu je možno z původních hodnot vypočítat nové interní hodnoty a graf znovu vykreslit bez zásahu zvenčí.

Prvek **Chart Control** definuje výčtové typy pro volby stylu vykreslení čáry `DrawType`, typu symbolu `SymbolType`, vzoru čáry `StrokePattern` a pozice legendy `LegendPosition` (viz obr. 3.7/str. 23).

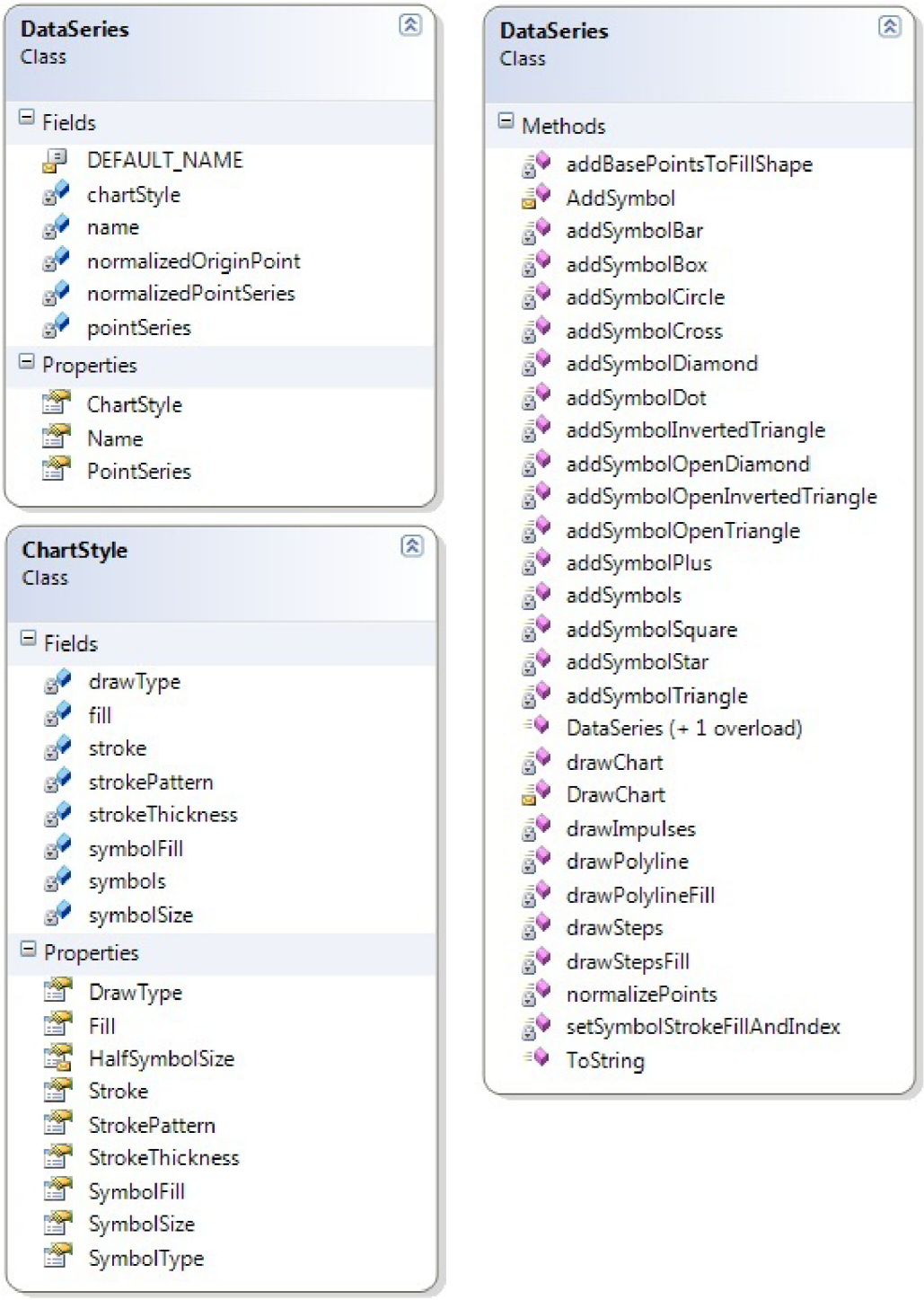
Třída `DataSeries` (viz obr. 3.8/str. 24) reprezentuje data pro vykreslení grafem. Obsahuje *property* `ChartStyle`, která definuje styl vykreslení dat. Ten je reprezentován třídou `ChartStyle`, která zprostředkovává volby vykreslení. Třída `DataSeries` dále zprostředkovává metody pro vykreslení symbolu a křivky dle zvoleného nastaveného stylu.

Třída `Legend` realizuje vykreslení legendy do grafu dle pozice definované v *property* `LegendPosition`.

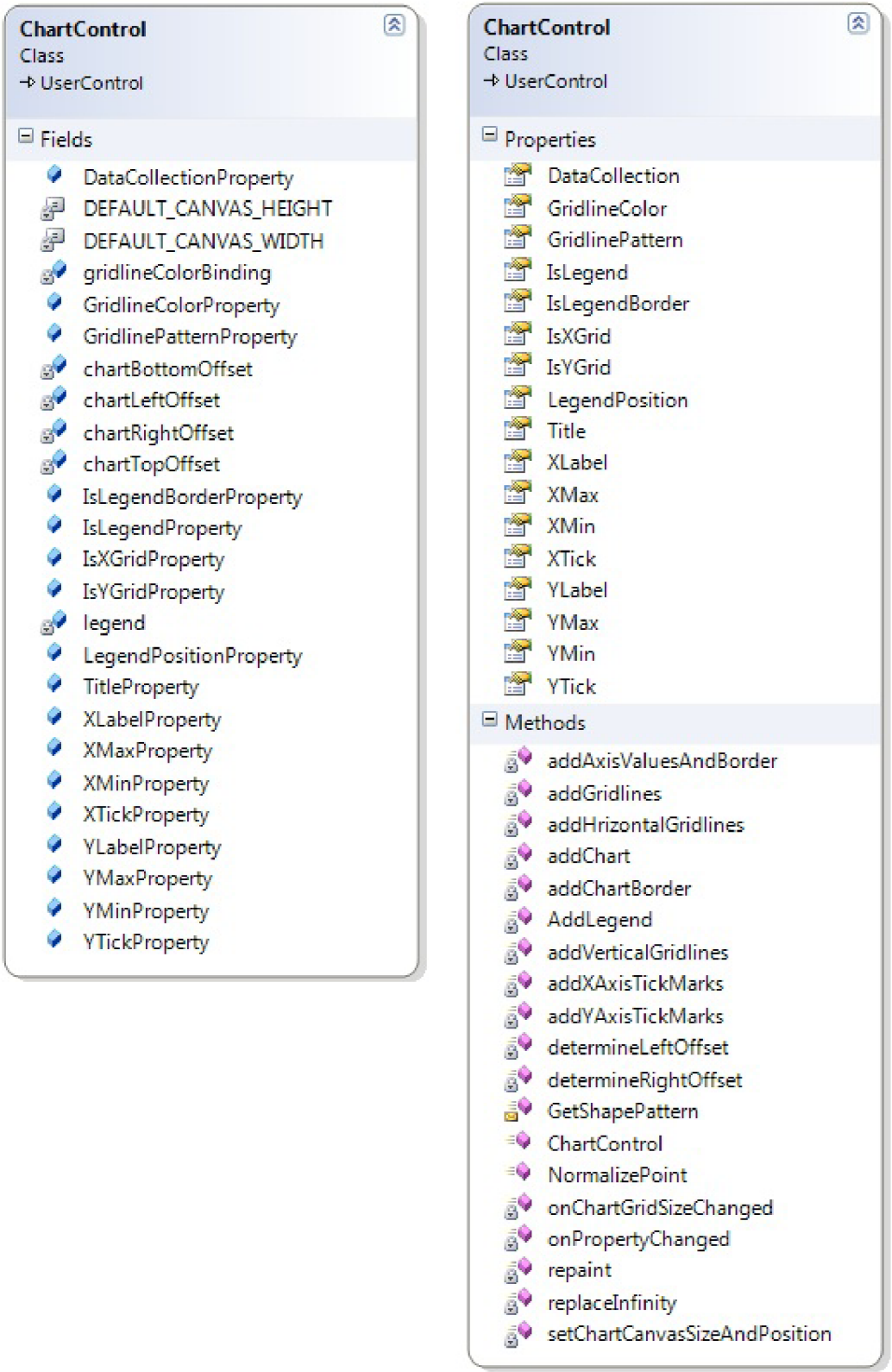
Třída `ChartControl` (viz obr. 3.9/str. 25) představuje ovládací prvek **Chart Control**. Definuje volby pro graf, jako je minimum a maximum os, popis grafu a další. Podobně jako u ostatních prvků knihovny **Tech4WPF** jsou *property* implementovány přes *dependency property*.



Obrázek 3.7: Výčty `SymbolType`, `LegendPosition`, `DrawType` a `StrokePattern`



Obrázek 3.8: Třídy `DataSeries` a `ChartStyle`



Obrázek 3.9: Třída `ChartControl`

3.2 Dokumentace

Zdrojový kód je dokumentován XML komentáři. Protože **Tech4WPF** je open source projekt, jsou dokumentovány nejen veřejné i privátní (případně interní) třídy, rozhraní, *property* a metody. Tato dokumentace je automaticky součástí *assembly*, takže při použití těchto ovládacích prvků bude vývojovým prostředím zobrazována (např. prostřednictvím *IntelliSense* při psaní kódu).

Z těchto komentářů byla vygenerována i samostatná vývojářská dokumentace pomocí nástroje *Sandcastle* resp. *SandCastle Help File Builder*[6] (SHFB). V *solution* **Tech4WPF** je obsažen projekt *Tech4WPF Documentation*, kterým je možno spustit kompilaci dokumentace. Ve Visual Studiu musí však být nainstalováno rozšíření SHFB a jeho součásti. Styl vzhledu použitý pro dokumentaci byl nastaven na styl Visual Studia 2010, známý také ze současného stylu dokumentací knihoven .NET a z dokumentace na webu MSDN.

Dokumentace byla vygenerována v těchto dvou formátech:

- HTML Help v1 (CHM)
- MS Help Viewer (MSHC)

HTML Help v1 jsem zvolil z důvodů kompatibility. Je to nejstarší formát dokumentace a je možné jej zobrazit na většině počítačů[16]. Formát *MS Help v2* nebyl sestavován, protože nástroje pro jeho vytvoření se s verzí Visual Studia 2010 nedistribuují (nástroje je možné ale najít v SDK Visual Studia 2008). Tento formát dokumentace byl ve Visual Studiu 2010 nahrazen novějším formátem *MS Help Viewer*. Soubory této dokumentace je možné do systému nainstalovat buď pomocí nástroje *Help Library Manager*[7], nebo pomocí dávkového souboru, který je přiložen u sestavené dokumentace **Tech4WPF**.

3.3 Demonstrační aplikace

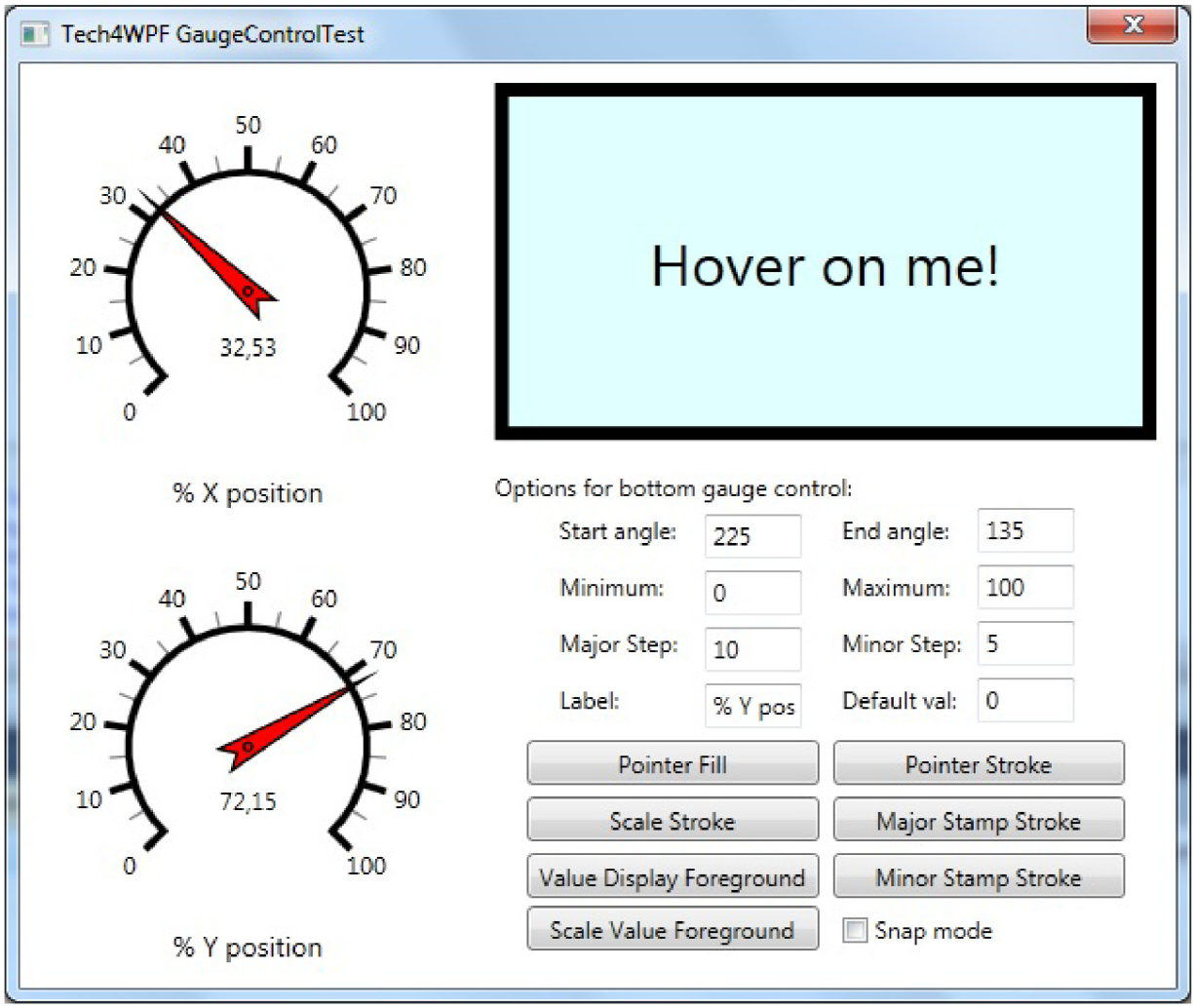
Pro demonstraci možností vytvořených ovládacích prvků v knihovně **Tech4WPF** jsem vytvořil jednoduchou vzorovou aplikaci, která je součástí projektu. Po spuštění aplikace se zobrazí menu, ve kterém je možno vybrat ovládací prvek. Po klepnutí na tlačítko s názvem prvku se zobrazí okno, kde je možno pomocí voleb měnit jeho vzhled, chování a vyzkoušet si jeho činnost. Všechny volby jsou implementovány pomocí *data bindingu*, takže se projeví ihned nebo v momentě, když editační políčko pro zadání hodnoty ztratí *focus*. Okno s prvkem je nemodální, takže je možno spustit další instanci okna s tímto prvkem a porovnat vliv rozdílných nastavení. Všechna otevřená okna se ukončí společně se zavřením okna menu.

Na obrázku 3.10 na straně 27 je předvedení ovládacího prvku **Gauge Control**. Ovládací prvek je v této aplikaci umístěn dvakrát. Jeden představuje procentuální pozici myši ve vyznačeném obdelníku v ose X, druhý v ose Y. Volby ovlivňují pouze chování spodního ovládacího prvku. Horní ovládací prvek zůstává ve výchozím nastavení.

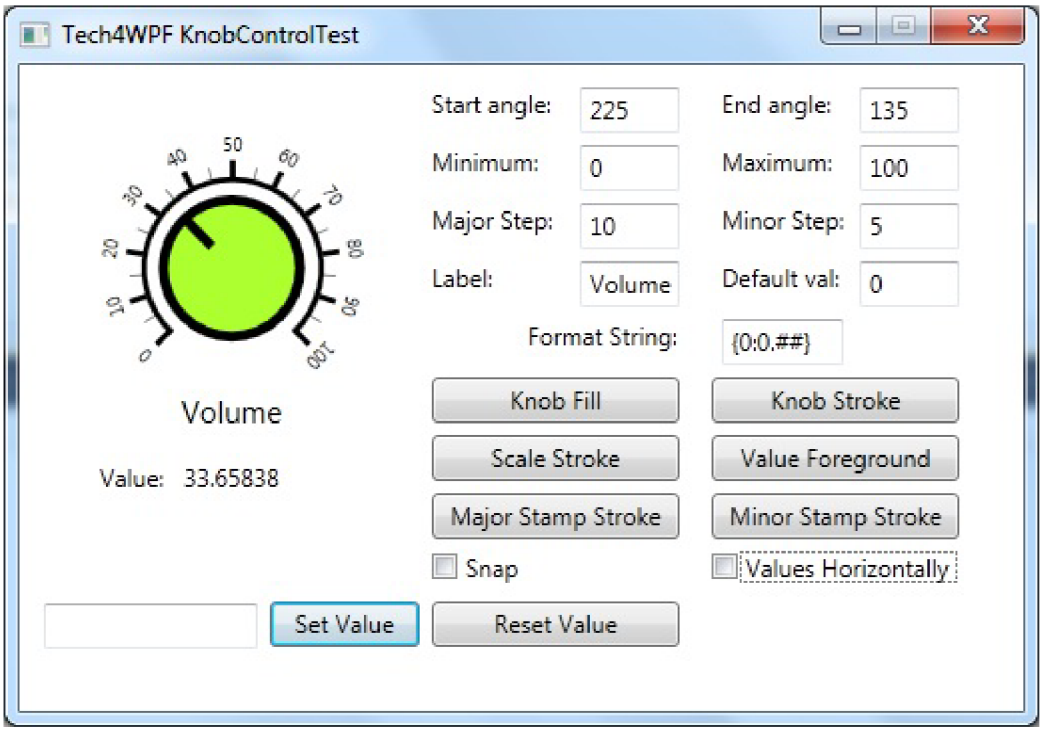
Ovládací prvek **Knob Control** umožňuje v demonstrační aplikaci (viz obr. 3.11/str. 28) nastavit hodnotu, kterou má knoflík ukazovat. Po vyplnění políčka vlevo dole a stisknutí tlačítka *Set Value* se tato hodnota nastaví. Místo stisknutí tlačítka *Set Value* je také možno použít klávesu *enter*.

[6] viz http://shfb.codeplex.com/

[7] ve Visual Studiu 2010 jej lze nalézt v menu Help/Manage Help Settings



Obrázek 3.10: Demonstrační aplikace **Gauge Control**

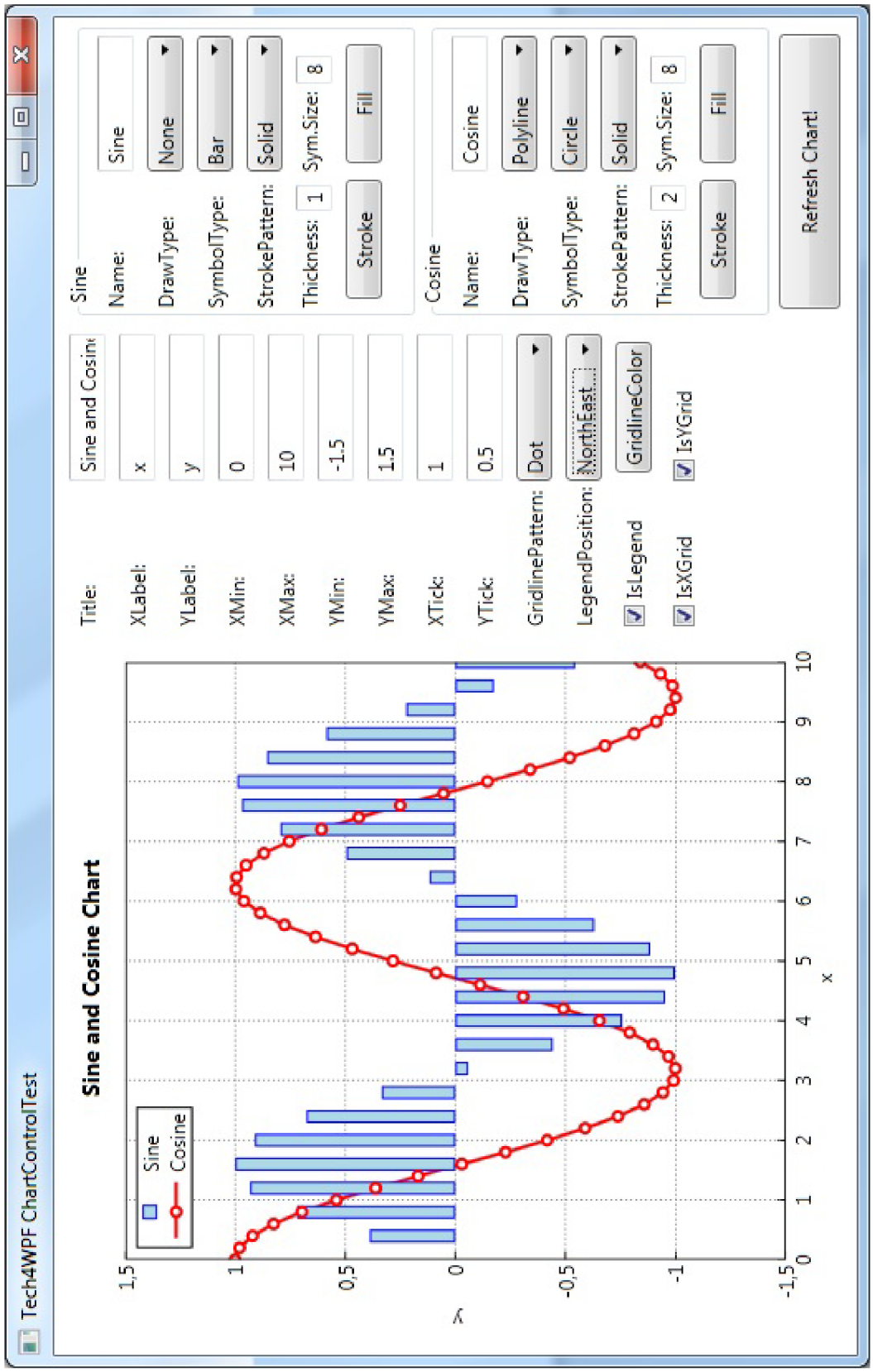


Obrázek 3.11: Demonstrační aplikace **Knob Control**

Na obrázku 3.12 na straně 29 vidíme demonstrační aplikaci ovládacího prvku **Chart Control**. Volby vlastností datových řad (*Sine* a *Cosine*) nepropagují svou změnu okamžitě do vykreslení grafu. Přesto, že jsou implementovány také pomocí *data bindingu*, změny se neprojeví ihned, ale až po stisknutí tlačítka *Refresh Chart* nebo při změně velikosti okna. Důvod tohoto chování vychází z návrhu prvku. *Property* `DataCollection` je implementována přes *dependency property*, takže je ovládací prvek **Chart Control** notifikován o její změně – a protože `DataCollection` je typu `List<DataSeries>` (jedná se tedy o instanci třídy umístěnou na haldě), změnou se rozumí změna ukazatele. Vnitřní změna hodnot datového obsahu *property* `DataCollection` neovlivní nijak ukazatel na tyto data. Pokud tedy chceme, aby se při úpravě datového obsahu `DataCollection` překreslil graf, stačí vytvořit novou instanci objektu pro *property* `DataCollection`. Novou instanci dat inicializujeme původními daty předanými do konstruktoru.

Pro představu uvádím úryvek kódu:

```
\\ DataCollection inner data change...
chartControl.DataCollection =
    new List<DataSeries>(chartControl.DataCollection);
```



Obrázek 3.12: Demonstrační aplikace **Chart Control**

Kapitola 4

Závěr

Cílem této bakalářské práce bylo navrhnout sadu znovupoužitelných ovládacích prvků pro WPF sloužících pro vizualizaci dat, a tyto prvky implementovat. Po prostudování teorie a technologií použitých ve WPF jsem se seznámil s již dostupnými ovládacími prvky. Zjistil jsem, že dostupnost takových prvků pro neziskové aplikace (kde se neuvažuje nákup komerčních knihoven) je špatná, a proto jsem navrhl projekt **Tech4WPF**. Tento projekt nyní obsahuje tři základní ovládací prvky, které cílí na technické aplikace. Pro seznámení se s možnostmi těchto prvků jsem vytvořil demonstrační aplikaci, která zároveň slouží i k jejich otestování.

Projekt **Tech4WPF** byl ke dni odevzdání této práce zveřejněn na portálu *Codeplex.com*, který se zabývá open source projekty a je zastřešován firmou Microsoft. Lze jej najít na adrese `tech4wpf.codeplex.com`. Protože se předpokládá, že projekt budou používat další vývojáři a že se do jeho vývoje zapojí i další lidé, byla k němu vytvořena vývojářská dokumentace.

Možnost pokračování projektu vidím ve zdokonalení již vytvořených ovládacích prvků, rozšíření jejich možností vzhledu (např. přidání nových stylů ručičkových ukazatelů), rozšíření konfigurovatelnosti a postupném přidávání prvků nových. Portál *Codeplex.com* nabízí možnosti vedení projektu a spolupráce s ostatními členy komunity, správu verzí, systém pro zachytávání chyb a přidávání požadavků na novinky s možností hlasování pro jejich implementaci, čímž tvoří dobrou základnu pro další vývoj.

Literatura

[1] Operating Systems [online]. URL http://www.talktoanit.com/A+/aplus-website/lessons-operating-systems.html, 2007-05-24 [cit. 2012-05-10].

[2] Windows Versions Features [online]. URL http://www.emsps.com/oldtools/mswinv.htm, [cit. 2012-05-10].

[3] Crescimanno, G.: Introduction to MVVM in Silverlight [online]. URL http://www.silverlightblog.net/2011/01/introduction-to-mvvm-in-silverlight/, 2011-01-28 [cit. 2012-05-11].

[4] Dostál, M.: *Základy tvorby uživatelského rozhraní*. PřF Univerzita Palackého, 2007.

[5] Holan, T.: MVVM ve WPF a Silverlightu, část 1: Základní třídy [online]. URL http://blog.imp.cz/post/2011/03/07/MVVM-ve-WPF-a-Silverlightu-cast-1-Zakladni-tridy, 2011-03-07 [cit. 2012-01-18].

[6] Isaacson, W.: *Steve Jobs*. Práh, 2011, ISBN 978-80-7252-352-8.

[7] Jirava, J.: XAML jako deklarativní jazyk [online]. URL http://xaml.cz/wpf/xaml-jako-deklarativni-jazyk/, 2010-01-13 [cit. 2012-01-18].

[8] Jirava, J.: Používáme Model-View-ViewModel [online]. URL http://xaml.cz/wpf/pouzivame-model-view-viewmodel-uvod/, 2010-02-02 [cit. 2012-01-18].

[9] Microsoft: Introduction to WPF [online]. URL http://msdn.microsoft.com/cs-cz/library/aa970268.aspx, 2012 [cit. 2012-01-18].

[10] TommySoft: Tutorial: Creating a Lookless WPF Custom Rotate Control [online]. URL http://www.codeproject.com/Articles/69048/Tutorial-Creating-a-Lookless-WPF-Custom-Rotate-Con, 2010-03-28 [cit. 2012-05-10].

[11] Truck, M.: The Real History of the GUI [online]. URL http://www.sitepoint.com/real-history-gui/, 2001-08 [cit. 2012-05-10].

[12] Wikipedia: X Window System [online]. URL http://cs.wikipedia.org/wiki/X_server, 2001-02-08 [cit. 2012-05-10].

[13] Wikipedia: Extensible Application Markup Language [online].
URL http://cs.wikipedia.org/wiki/XAML, 2011-12-28 [cit. 2012-01-18].

[14] Wikipedia: History of the graphical user interface [online]. URL http://en.wikipedia.org/wiki/History_of_the_graphical_user_interface, 2012-04-19 [cit. 2012-05-10].

[15] Wikipedia: NaN [online]. URL http://en.wikipedia.org/wiki/NaN, 2012-04-22 [cit. 2012-05-10].

[16] Wikipedia: Microsoft Compiled HTML Help [online].
URL http://en.wikipedia.org/wiki/Microsoft_Compiled_HTML_Help, 2012-05-12 [cit. 2012-05-13].

[17] Xu, J.: *Practical WPF Charts and Graphics*. Apress, 2009, ISBN 978-1-4302-2481-5.

[18] Zemčík, P.: *Tvorba uživatelských rozhraní – Studijní opora*. FIT VUT Brno, 2006.

Příloha A

Obsah CD

- **bin** – složka se vzorovou aplikací
- **doc** – složka s dokumentací
- **src** – složka se zdrojovými kódy projektu
- **tex** – složka se zdrojovými kódy bakalářské práce v LaTeXu
- **thesis.pdf** – bakalářská práce v elektronické podobě