VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

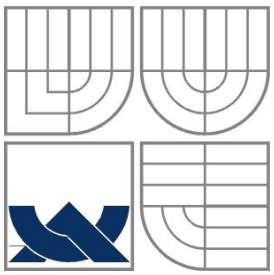
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KOMPONENTA GRAFŮ PRO .NET

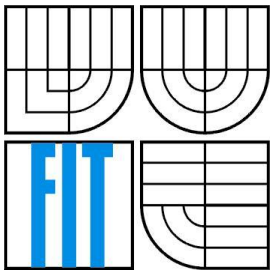
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE PAVEL RIEDL
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KOMPONENTA GRAFŮ PRO .NET

.NET CHART COMPONENT

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — PAVEL RIEDL
AUTHOR

VEDOUCÍ PRÁCE — Ing. MICHAL PAJGRT
SUPERVISOR

BRNO 2008

Abstrakt

V současné době, kdy k osobním počítačům usedá stále více uživatelů, jsou kladeny nároky na jednoduchost a přehlednost vytvořených programů. Jeden ze způsobů, kterým lze jejich práci ulehčit, je tvorba grafických uživatelských rozhraní (GUI). Jestliže program využívá platformu .NET, pak lze pro jejich vytváření využít část její knihovny tříd. Jedním prvkem, který není součástí této knihovny, je i komponenta pro vykreslování grafů. Tvorbou právě takové komponenty se zabývá tato bakalářská práce.

Klíčová slova

.NET, C#, komponenta, user control, GDI+, graf

Abstract

Nowadays, when the number of computer users is still increasing, there are more demands for simplicity and lucidity of created applications. One way, how to make their work easier, is the creation of graphic user interfaces (GUI). If the application uses the .NET Framework, then we can use a part of its class library to create the GUIs more easily. One item that is missing in this library is the chart component. Developing of this kind of component is a topic of this Bachelor's thesis.

Keywords

.NET, C#, component, user control, GDI+, chart

Citace

Pavel Riedl: Komponenta grafů pro .NET. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Komponenta grafů pro .NET

Prohlášení

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

……
Pavel Riedl
17. ledna 2008

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce, Ing. Michalu Pajgrtovi, za jeho pomoc při stanovení cíle a za rady při tvorbě komponenty.

Obsah

Úvod

Platforma .Net byla poprvé představena v roce 2002, jejím hlavním cílem je zjednodušit tvorbu aplikací, poskytnout jednotné prostředí pro běh aplikací napsaných v různých jazycích a nabídnout vývojářům sadu užitečných nástrojů pro tvorbu vlastních aplikací. Jejími hlavními součástmi jsou *.Net Framework class library* a *Common Language Runtime (CLR).*

Knihovna tříd obsahuje veliké množství komponent poskytujících běžné funkce, které se ve stejné podobě vyskytují v mnoha aplikacích. Programátor je může využít zejména pro tvorbu formulářů pro Windows[1] (WinForms), webových formulářů (ASP.NET) a tvorbu databázových aplikací (ADO.NET). Technologie, které jsem zde uvedl, jsou z mého pohledu ty nejpodstatnější, nicméně je jich daleko více.

CLR je virtuální stroj na kterém běží .NET aplikace. Je zde určitá analogie s programy napsanými v jazyce JAVA, avšak našli bychom hodně rozdílů. Společným rysem je fakt, že program není kompilován do nativního kódu pro architekturu, na které poběží, ale do kódu který poběží na virtuálním stroji. U .NET se jedná o jazyk *MSIL (Microsoft Intermediate Language).* U obou těchto technologií se vyskytuje pojem *JIT (Just In Time) Compiler*, jehož úkolem je překládat program do strojového kódu za jeho běhu. Hlavní odlišnost spočívá v důvodu, proč tomu tak je. U jazyka JAVA je tomu kvůli přenositelnosti aplikace, která je napsána v jednom jazyce, avšak spustitelná pod různými operačními systémy, či na různých mobilních zařízeních. U .NET je důvod zcela odlišný. Je jím hlavně jazyková nezávislost, kde CLR definuje sémantiku a datové typy používané v jeho jazycích. Jednotlivé jazyky jsou pak odlišné zejména ve své syntaxi. Jelikož jsou komponenty zkompilovány MSIL, tak je možné, aby program vyžíval komponentu napsanou v jiném jazyce, než je on sám. Avšak aplikace využívající .NET Framework jsou zaměřeny na operační systémy MS Windows. Nejčastějšími jazyky, které jsou s .NET spojovány jsou: C#, Visual Basic .NET a jazyk C++ upravený pro běh na platformě .NET. Nicméně lze využít i jazyky, které se běžně využívají pro skriptování, jako je například Python, nebo JavaScript. Všechny .NET jazyky jsou přibližně stejně výkonné.

Jazyková nezávislost výrazně zvyšuje využitelnost napsaných komponent, což z dělá z .NET ideální nástroj pro jejich tvorbu už i proto, že při jejich programování lze využít i komponent jiných, které mohou být také napsány v různých jazycích.

[1] Existuje také prostředí pro Unixové systémy, které se nazývá *mono*.

1 Tvorba komponent v .NET

Tato kapitola se zabývá tvorbou komponent v prostředí .NET obecně a zatím nijak blíže nespecifikuje využití těchto metod a postupů na komponentu pro vykreslování grafů. Přestože existuje více různých způsobů, které se dají k tvorbě komponent využít, tak specifikuje pouze ty, které byly využity pro tvorbu komponenty v zadání bakalářské práce.

1.1 Provázanost s objektově orientovaným programováním

CLR v .NET je navrženo pro běh objektově orientovaných jazyků. S tím také souvisí způsob, jakým se v praxi programuje.

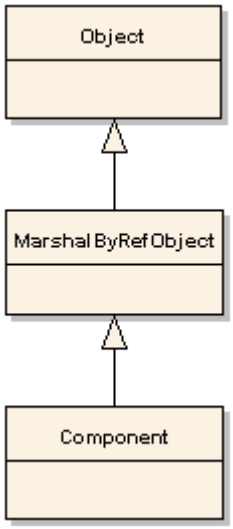
Pro programování v .NET se většinou používá integrované vývojové prostředí Microsoft Visual Studio[2]. Tento program značně zjednodušuje využívání komponent tím, že jeho součást, tzv. *Designer*, umožňuje programátorovi jednoduchým způsobem vložit do programu komponentu, kterou chce programátor využít a vygeneruje za něj část kódu sám.

Automatické generování kódu však není využitelné za všech okolností, a proto je nutné při tvorbě komponenty dodržovat postupy, které ho umožní. V této souvislosti existují 2 druhy komponent, první se nazývají čistě jen komponenty, druhé jsou součástí formuláře a označují se anglickým výrazem control. Pokud chceme těm, kteří budou naši komponentu používat, jejich práci co nejvíce usnadnit, měli bychom navrhnout naši komponentu tak, aby se dala Designerem využít.

1.1.1 Komponenta

Pokud komponenta není prvkem formuláře, pak jediné, co musí implementovat je rozhraní *IComponent*. Ovšem toto rozhraní obsahuje veliké množství různých vlastností, metod, či událostí. A tak zde existuje zjednodušení, při kterém využijeme jedné ze základních charakteristik OOP a to je dědičnost. Komponenta jednoduše zdědí to, co potřebuje z třídy *Component*, jak ukazuje obrázek 1.1. Jak lze z obrázku vidět, tak už i třída Component má své předky.

[2] Existují i podobné IDE, které jsou určené pouze pro vývoj v jednom jazyce. Jsou to např.: Microsoft Visual C#, Microsoft Visual C++ apod.



Obrázek 1.1: dědičnost komponenty

Jedním z nich je *MarshalByRefObject*. Předtím, než si ujasníme jeho význam, je ale potřeba si vysvětlit několik pojmů. CLR může spustit jednotlivé programy v rámci jednoho procesu operačního systému. Nicméně si sám tyto programy rozděluje do „procesů“ a ty se v jeho rámci pak nazývají *aplikační domény*. Zjednodušeně by se dalo říct, že aplikační doména je ekvivalent pro proces, který neběží přímo pod operačním systémem, ale v rámci CLR. Pokud je potřeba, aby objekty mezi aplikačními doménami komunikovali, mají v zásadě dvě možnosti. První z nich je zasílání kopií objektů mezi aplikačními doménami. Druhou možností je zasílání zpráv pomocí proxy. MarshalByRefObject je třídou, od které dědí všechny třídy, které si mezi aplikačními doménami zasílají zprávy pomocí proxy.

Druhým předkem je třída *Object*, od které v .NET dědí všechny třídy.

1.1.2 Control

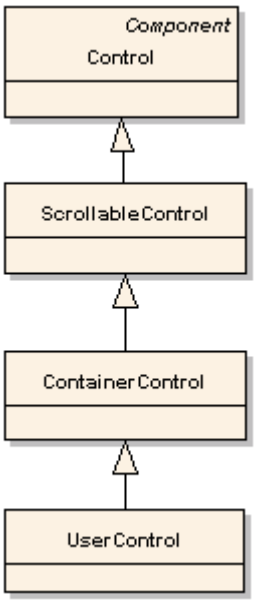
Contol je speciální typ komponenty, který je součástí formuláře ve Windows. K tomu, aby se mohl ve formuláři nacházet, musí obsahovat některé základní vlastnosti jako je například pozice, rozměry nebo události při přejetí myši. Základna těchto metod a událostí je značně široká a bylo by proto neefektivní je implementovat pro každý objekt znovu. Z toho důvodu se v .NET využívají k jejich tvorbě 2 postupy.

1.1.2.1 Dědění z již existující komponenty

Tento způsob je hodně používaný a jednoduchý. Používá se například k tvorbě numerických TextBoxů a podobných prvků, jejichž chování se mění jen mírně.

1.1.2.2 Dědění z abstraktní třídy UserControll

V tomto případě se dědí pouze chování komponenty ve formuláři a ostatní její funkce musí programátor implementovat sám. Hierarchii dědičnosti ukazuje obrázek 1.2. Z obrázku lze vidět, že třída Control už dědí z třídy Component, kterou jsme si již ukázali dříve.



Obrázek 1.2: dědičnost uživatelského prvku formuláře

Při tomto způsobu tvorby se může využít prvků již existujících (např.: ProgressBar a Label) k vytvoření komponenty, která je kombinuje. V tomto případě je dobré si všimnout v obrázku 1.2 třídy ContainerControll, která tento postup umožňuje. Nebo se může komponenta vykreslit uživatelsky. K tomu se v .NET používá metod ze jmenného prostoru Sytem.Drawing, jenž je standardní součástí knihovny tříd v .NET. Ten zaobaluje standardní funkce z GDI+. Tímto způsobem je vytvořena i komponenta pro vykreslování grafů, vytvořená v rámci této bakalářské práce.

1.2 Podpora Designeru

Jak už jsem uvedl výše, k tvorbě aplikací v .NET se většinou využívá pokročilých IDE jako je např. MS Visual Studio, jehož součást (Designer) usnadňuje seskládání aplikace z komponent. Obvykle se komponenty vkládají do programu pouhým jejich přetažením na formulář, při čemž Designer vygeneruje část kódu, kterou by jindy musel programátor psát ručně.

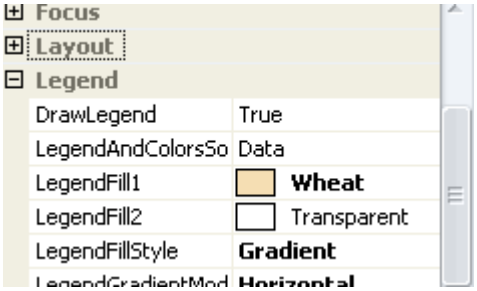
Designer ovšem neumožňuje pouze vkládání komponent, ale také jejich nastavení. V jazyce C# to vypadá tak, že pokud se zdrojový soubor s formulářem jmenuje například Form1.cs, pak vedle něj existuje ještě další soubor, který se jmenuje Form1.Designer.cs. Ten pak obsahuje metodu *InitializeComponent(),* ve které se nachází vygenerovaný kód. S nastavováním některých vlastností ovšem vznikají další nároky na to, co musí komponenta splňovat. Některé vlastnosti nemusejí dávat smysl při vytváření komponenty, ale třeba až za jejího běhu. Nebo může nastat případ, kdy dvě

vlastnosti určují rozsah hodnot, přičemž smysl dává, pouze pokud první je menší než druhá. Dalším nárokem je, že musíme Designer seznámit s tím jaká je výchozí hodnota vlastnosti. V případě že pak za tvorby programu budeme chtít vlastnosti ponechat její výchozí hodnotu, tak se pak tato vlastnost nebude v metodě InitializeComponent() nastavovat. To však stále ještě není vše, komponenta může také reagovat na události od uživatele, které bychom mohli chtít při jejím vytváření potlačit apod.

Designer tedy sice ulehčuje skládání programu z komponent, ale právě naopak tomu může být při jejich tvorbě.

1.2.1 Atributy

Atributy jsou jedním ze způsobů, kterým můžeme podporu Designeru vylepšit. V jazyce C# se zapisují přímo před danou vlastnost do hranatých závorek. U každé vlastnosti je z pravidla nutné použít tři atributy. Těmi jsou: *DefaultValue()* pro nastavení výchozí hodnoty, *Browsable()* pro nastavení, zda má být v době návrhu vlastnost vidět a *Category()*, která určuje do které kategorie má být v tabulce vlastnost zařazena. Tabulka vlastností (*Property Grid*) je zobrazena v obrázku 1.3.



Obrázek 1.3: Property Grid v MS Visual Studio 2005

Například u vlastnosti *DrawLegend* byl nastaven atribut Browsable na hodnotu true, Category na „Legend“ a DefaultValue na true.

1.2.2 Implementace rozhraní ISupportInitialize

Některé vlastnosti na sobě mohou záviset, například minimum a maximum z nějakého rozsahu. Pokud v takovém případě programujeme defensivně, tak vždy kontrolujeme hodnotu, která se do vlastnosti nastavuje a pokud má být hodnota maxima menší než minimum tak vyhozením výjimky dáme vědět, že daný případ nemá smysl a že se má nejprve snížit minimum pod hodnotu, kterou chceme maximu nastavit.

V takovém případě ale může nastat problém, pokud mají být hodnoty nastaveny pomocí Designeru. Uvažujme scénář, při kterém je potřeba nastavit minimum na -12 a maximum na -1 a výchozí hodnota těchto vlastností je 0 a 1. V metodě InitializeComponent() se nastavování vlastností řadí podle abecedy. Čili vlastnost jménem Maximum, se bude vždy nastavovat před vlastností

jménem Minimum. Pokud ji tedy zkusíme nastavit zápornou hodnotu, tak hodnota minima bude stále 0 a to i přesto, že později budeme jeho vlastnost nastavovat na -12.

Tento problém se řeší pomocí implementace rozhraní *ISupportInitialize*, které obsahuje dvě metody. První z nich se jmenuje *BeginInit(),* druhá *EndInit().* První je zavolána z metody InitializeComponent() před tím, než se začnou nastavovat vlastnosti a jejím úkolem je vypnout ověřování, zda vlastnost může danou hodnotu obsahovat, či nikoliv. Druhá je zavolána po ukončení a jejím úkolem je nejprve ověřit, jestli jsou vlastnosti nastaveny správně a pak jejich ověřování zase zapnout.

Tímto způsobem pak bude možno po zavolání BeginInit() nastavit maximum na -1, pak minimum na -12 a při zavolání EndInit() se ověří, zda platí, že maximum je větší, než minimum. Je to běžný způsob používaný při řešení podobných problémů.

1.2.3 Vlastnost DesignMode

V případě že z nějakého důvodu potřebujeme změnit chování komponenty v době jejího návrhu, můžeme využít vlastnosti *DesignMode* z třídy Component, dále implementované v třídě Control.

Tato booleovská vlastnost je pravdivá (Má hodnotu true.) v případě, že je komponenta ve stavu inicializace Designerem. V závislosti na její hodnotě lze pak nastavit různé chování komponenty. Například inicializovat grafovou komponentu vzorovými hodnotami apod.

2 Návrh komponenty grafů pro .NET

Tato kapitola stanovuje cíle a představuje návrh komponenty, jejího vzhledu a její vnitřní organizace.

2.1 Formulace cíle

Úkolem je vytvořit standardní komponentu umožňující zobrazení různých typů grafu. Těmi jsou např.: sloupcový, koláčový atd. To znamená, že všechny typy grafu nejsou přesně dány. Čili jako další grafy můžeme vytvořit spojnicový graf a graf závislosti os X a Y. Dále máme přidat možnosti formátování grafu a operace s grafem. Jazyk který se má použít k implementaci není striktně zadán, čili si můžeme zvolit kterýkoliv z jazyků, který podporuje .NET. Jelikož jazyků, které tato platforma podporuje je hodně, tak máme hodně možností, pro výběr toho ideálního.

Chceme, aby komponenta vypadala pro všechny grafy uceleně. Dále pak, aby pracovala se stejným typem dat, aby mohl uživatel měnit typ grafu i za běhu programu.

2.2 Návrh typu grafů

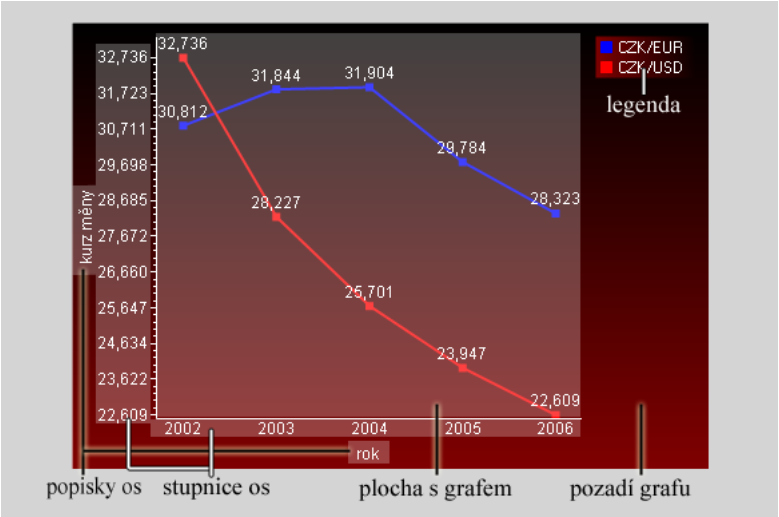
Budeme tedy implementovat 4 typy grafů a to:

- Sloupcový
- Spojnicový (někdy též označován jako čárový)
- Koláčový
- Graf závislosti os X a Y (nebo také bodový graf či graf XY)

Ačkoliv se může zdát, že jednotlivé grafy nemají moc společného, opak je pravdou. Všechny vykreslují legendu. Jsou, s výjimkou koláčového, stejným způsobem rozmístěny na ploše, ve které se vykresluje. Všechny mají stejné možnosti pro nastavení pozadí grafu, použití barev, stylu legendy apod.

Proto je tedy cílem si určit co mají všechny typy grafu společného a to pak sjednotit do společné abstraktní třídy. Od této třídy pak budou všechny grafy dědit a implementovat pouze své rozdíly.

Obrázek 2.1 demonstruje, jak může vypadat spojnicový graf a označuje důležité objekty, co se rozložení grafu týče.



Obrázek 2.1: spojnicový graf

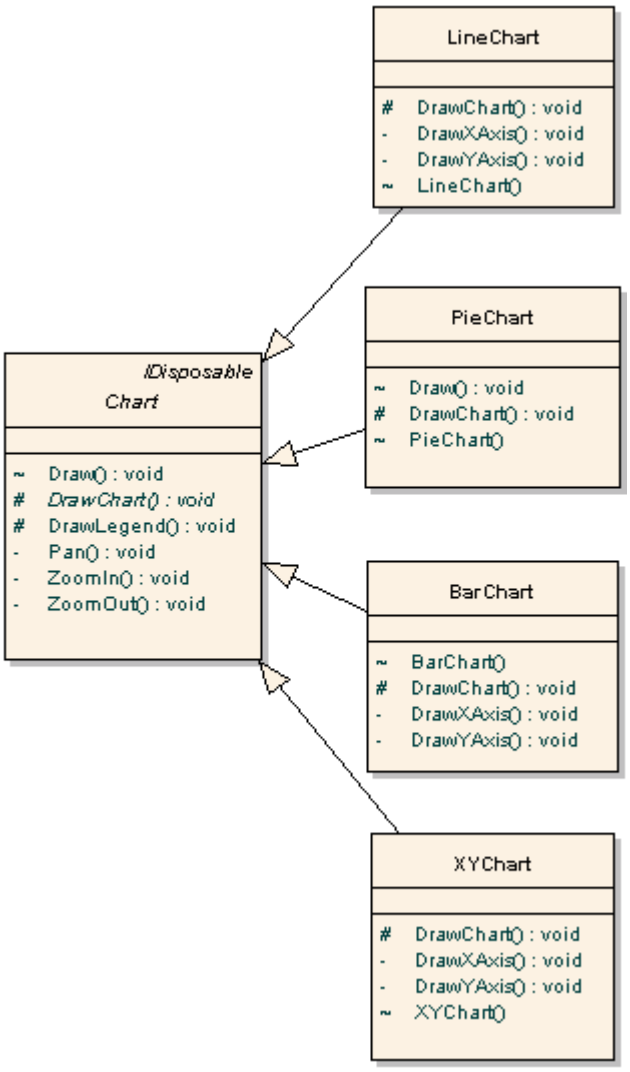
Na obrázku 2.1 jsou vidět podstatné věci, které sdílejí jednotlivé typy grafů. Jednou z nich je legenda. Každý graf ji vykresluje, a proto by měla být metoda pro její vykreslení součástí již zmíněné abstraktní třídy. Dalším shodným prvkem pro ostatní typy grafů je umístění plochy s grafem. Protože je u všech grafů toto rozmístění stejné pak můžeme v abstraktní třídě určit i rozložení grafu na ploše. To má však jedno omezení a tím je koláčový graf, který bude rozložen jinak, proto bude tato metoda označena jako virtuální, aby mohl mít koláčový graf její vlastní implementaci.

U grafu spojnicového a grafu závislosti os X a Y bude navíc vhodné přidat možnost přiblížení a posouvání přiblíženého grafu. To se bude u obou těchto grafů chovat stejně, proto můžeme vše, co k tomu bude potřeba zahrnout do abstraktní třídy.

Mohlo by se zdát, že stejný způsob budeme moci použít i na stupnice os, ale tento případ je zcela odlišný, protože každý graf používá jiný typ stupnice. Spojnicový graf z obrázku 2.1 má na ose X popsány roky, ale právě tak tam mohou být i třeba měsíce např. leden, únor atd., nebo jakýkoliv jiný textový řetězec. Na rozdíl od něj má graf závislosti os X a Y stupnici zcela odlišnou, kde popisuje osu podle jejího rozsahu. Kvůli těmto odlišnostem si tedy musí každý graf vykreslit stupnici sám.

A nakonec pozadí grafu a jeho způsob vykreslení již není na typu grafu závislý a na rozdíl od legendy je vykresleno vždy. Proto metoda pro jeho vykreslení bude součástí třídy komponenty.

Všechno co jsme zatím navrhli, ukazuje diagram na obrázku 2.2.



Obrázek 2.2: Diagram Tříd, zobrazující nejpodstatnější věci pří návrhu grafu.

Při každém překreslení komponenty bude zavolána metoda Draw(), z třídy Chart. Ta bude mít za úkol z velikosti komponenty vypočítat velikosti regionů, do kterých se bude graf vykreslovat. Následně pak zavolá metodu DrawLegend() pro vykreslení legendy a DrawChart() pro vykreslení grafu. Ta pak převezme řízení a udělá vše potřebné a s výjimkou koláčového zavolá i metody pro vykreslení stupnice os. Metoda Draw() bude navíc označena jako virtuální, aby mohl koláčový graf její chování změnit.

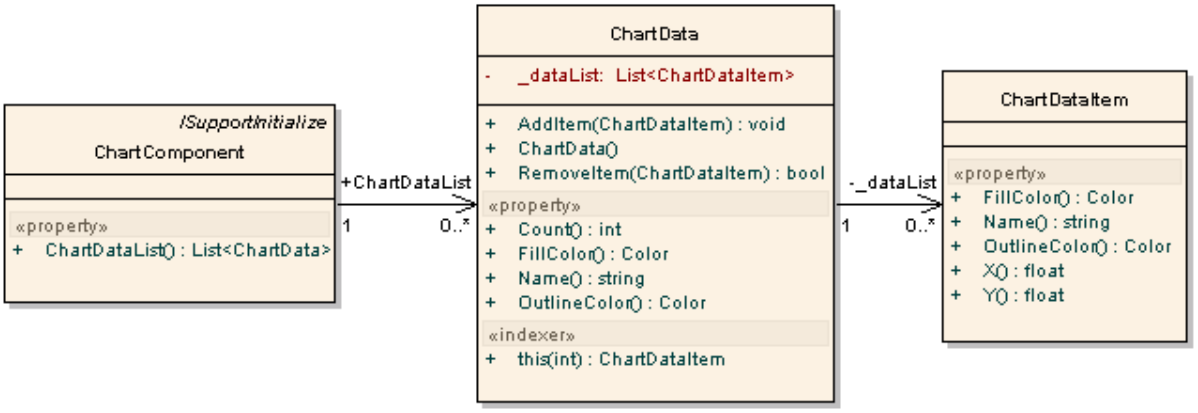
2.3 Zadávání dat to grafu

Máme navrženo, jaké typy grafů bude komponenta nabízet. Tak nám ještě zbývá navrhnout, jakým způsobem se budou do grafu zadávat jednotlivé hodnoty. Na obrázku 2.1 je vidět spojnicový graf. Všimněme si, že zobrazuje 2 datové řady do jednoho obrázku, jednou je kurz koruny vůči euru a

druhou vůči americkému dolaru. Každá z těchto řad obsahuje více položek (hodnot), má svoji barvu a jméno[3].

Právě tak, jako u spojnicového grafu hrají hlavní roli datové řady, tak u koláčového grafu ji zase mají jejich položky. Je tedy nutné, aby každá z těchto položek také mohla nést informaci o své barvě, jménu a hlavně svoji hodnotu.

Když teď seskládáme všechno dohromady, pak zjistíme, že budeme potřebovat kolekci jednotlivých položek grafu. Tato kolekce bude součástí datové řady, která navíc ponese ještě další informace. Nakonec, abychom mohli do grafu vložit více datových řad, tak bude muset komponenta nést jejich kolekci. To vše ukazuje diagram na obrázku 2.3.



Obrázek 2.3: diagram tříd zobrazující reprezentaci dat v grafu

Třída *ChartDataItem* znázorňuje jednu položku. Třída *ChartData* zase jednu datovou řadu. Všimněme si, že obsahuje kolekci jednotlivých položek, které je možné vkládat a odebírat dvojicí metod *AddItem()* a *RemoveItem()*. A nakonec komponenta obsahuje kolekci datových řad. Některé grafy však mohou mít ještě nějaké další požadavky, kvůli kterým bude potřeba tyty třídy dále rozšířit.

2.4 Další požadavky grafu

V části 2.3 jsme navrhli, jakým způsobem se budou do grafu vkládat data, které má graf zobrazit. Ale co když to vše ještě nestačí? Co když potřebujeme v grafu zobrazit více než jen sloupce, nebo body spojené čarou? Typem grafu, který potřebuje víc je graf XY. Nezobrazuje totiž jen stejně vzdálené body propojené čarou, ale bere v úvahu obě jejich souřadnice. Díky tomu je možné jej využít k zobrazení různých matematických či fyzikálních funkcí. To ovšem klade další požadavek na to, co vše graf bude muset zobrazit. Pokud bude grafem funkce hyperbola, pak by měl graf umět zobrazit její asymptoty. Právě tak můžeme narazit na nespojité funkce, kde budeme potřebovat zobrazit, zda bod do intervalu ještě patří, či už ne. U některých funkcí jsou grafem pouze isolované body, u jiných

[3] Není nutné, aby se jednalo o jméno, ale o něco popisujícího danou řadu. Právě tak se může jednat i o popis funkce např.: y=ln(x). V obrázku 2.1 se jedná o popisek CZK/EUR a CZK/USD.

zase spojitá čára. Body mohou být spojeny čarou, nebo také křivkou. U různých statistických měření, je možné, že bude potřeba navíc zvýraznit závislost, kterou se měření ubírá. To znamená, že bude potřeba body proložit přímkou, křivkou, či funkcí. V tom případě taky bude nutné ověřit, zda lze pro zadané body danou metodu použít.

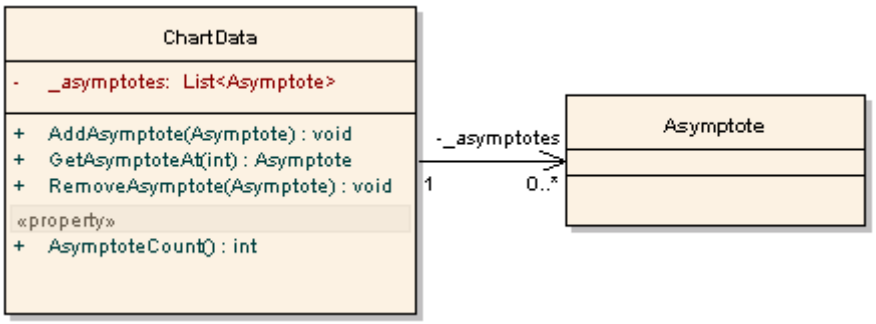
K tomu, abychom mohli tyto možnosti nabídnout, je potřeba také rozšířit třídy, které slouží pro vkládání dat do grafu.

2.4.1 Asymptoty

Existuje veliké množství funkcí, pro které existuje přímka, k níž se její grafy nekonečně blíží. Jedná se především o funkce exponenciální, logaritmické, goniometrické (např. tangens) a také funkce, jejichž grafem je třeba hyperbola. S těmito typy funkcí se v matematice a ostatních přírodních vědách setkáváme velmi často. Z toho důvodu musí graf, jenž znázorňuje závislost os X a Y, umět asymptoty zobrazit.

Vycházíme z toho, že asymptota je přímka. A víme, že každá přímka je zadána dvěma body. Také ale může být zadána pomocí úhlu a posunutí, což můžeme pro její reprezentaci v našem grafu využít.

To znamená, že do třídy, která reprezentuje datovou řadu, můžeme přidat asymptotu. Grafy jako je třeba hyperbola mají asymptoty dvě, a tak bude nejvhodnější přidat datové řadě rovnou celou kolekci asymptot, které bude tvořit třída pro ně určená.



Obrázek 2.4: asymptoty

Obrázek 2.4 zobrazuje změny, které nastanou v třídě ChartData. Byly přidány metody pro přidání a odebrání asymptot. Vlastnost pro zjištění jejich počtu. A pak taky metoda GetAsymptoteAt(int index), která umožňuje výběr asymptoty podle indexu.

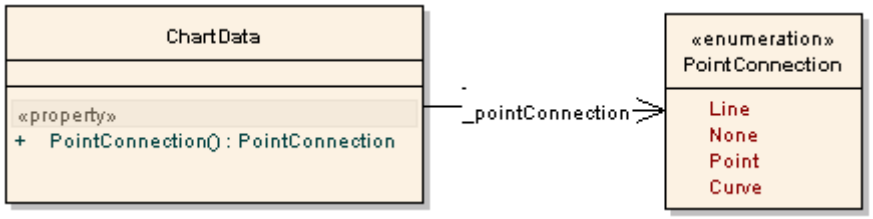
2.4.2 Propojení bodů

Propojení bodů rozlišuje, jakým způsobem budou body v grafu propojeny. Po povolení změny budeme potřebovat výčet těchto typů a přidat jej jako další vlastnost do třídy s datovou řadou. U tohoto propojení budeme rozlišovat 4 typy:

- propojení přímkou

- propojení křivkou
- žádné propojení
- zvýraznění zadaných bodů

Někoho může nyní napadnout jaký je důvod k tomu, aby se body vůbec nijak nezobrazovali. Důvod zde ale je. Může totiž nastat případ, kdy zadáme body, které budeme chtít aproximovat funkcí a budeme chtít zobrazit pouze aproximační křivku. V takovém případě přijde i tato možnost vhod.



Obrázek 2.5: možnosti propojení bodů

Obrázek 2.5 zobrazuje změny, které nastanou, při přidání možnosti propojení bodů.

2.4.3 Náležitost bodu k intervalu

Náležitost určuje, zda bod do daného intervalu patří, nebo ne. Příkladem funkce, která tuto vlastnost využívá je funkce *celá část*[4].

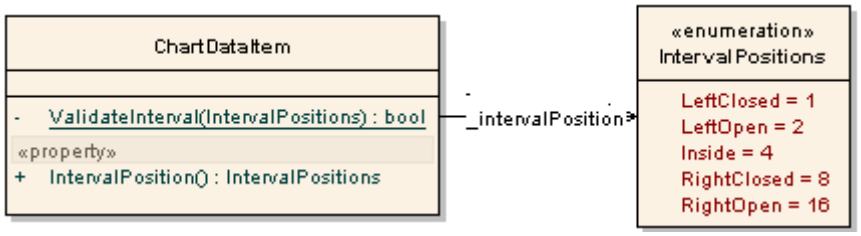
Náležitost k intervalu se nevztahuje k datové řadě, ale k bodům samotným. Proto je potřeba tuto vlastnost implementovat do třídy, která nese údaje o bodech.

Pokud se na tuto vlastnost podíváme z pohledu pozice bodu v intervalu, tak nám nastávají následující možnosti:

- Bod uzavírá interval zleva a nepatří do něj
- Bod uzavírá interval zleva a patří do něj
- Bod leží uvnitř intervalu
- Bod uzavírá interval zprava a patří do něj
- Bod uzavírá interval zprava a nepatří do něj
- Bod je osamocen. Jinak by se dalo říci, že uzavírá interval zleva a zároveň zprava. A v obou případech do nich patří.

Vidíme tedy, že budeme muset vytvořit tuto vlastnost tak, aby bylo možné zadat více těchto možností zároveň. Ale jediný případ, kdy dává kombinace možností smysl, je ten, kdy bod uzavírá interval z obou stran a zároveň do něj patří. Což vede k tomu, že při zadávání náležitosti k intervalu budeme muset ověřovat, zda má daná kombinace smysl.

[4] y=[x]. Y vyjadřuje největší celé číslo, které je menší, než číslo na ose X. Například pro x=3 je y rovno třem. Ale jestliže je x o málo menší než 3, pak už je hodnota y rovna dvěma.



Obrázek 2.6: náležitost k intervalu

Obrázek 2.6 znázorňuje změny v třídě ChartDataItem, která representuje jednu položku (či jeden bod). Výčet *IntervalPositions* má hodnoty, které lze kombinovat. V třídě položky navíc ještě přibila statická metoda *ValidateInterval()*, která vrátí false, pokud daná kombinace nemá smysl a na základě toho nastane při nastavování vlastnosti výjimka.

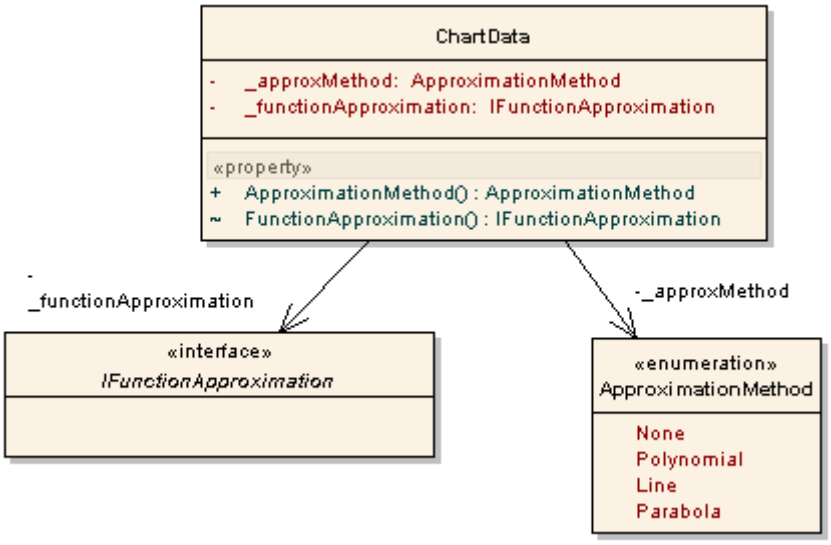
2.4.4 Aproximace funkcí

V praxi se nejčastěji používají pro funkce, které nemají žádný předpis a její hodnoty byly získány nějakým typem měření. V tom případě se užívají k výpočtu funkce v neměřených bodech. Další způsob jejich využití je v grafech pro zvýraznění závislosti funkce. Někdy mohou body ležet (alespoň přibližně) v přímce. Jindy mohou tvořit část paraboly, nebo být součástí jiné funkce. Pokud tedy chceme jejich závislost v grafu nějakým způsobem vyjádřit, pak určíme jejich aproximační funkci a s její pomocí pak vypočítáme další body, které leží blíže[5] u sebe a jejich propojením nám vznikne křivka, která znázorňuje tuto závislost.

Aby mohl graf podporovat více aproximačních metod, je nutné vytvořit rozhraní, jež budou všechny metody implementovat. Daný objekt, který bude aproximační metodu implementovat, pak bude součástí datové řady. Pro zjednodušení použitelnosti komponenty pak ještě vytvoříme výčet těchto metod, pomocí kterého se pak bude daná metoda volit.

Obrázek 2.7 zobrazuje změny, které nastanou při začlenění aproximační metody do třídy, která representuje datovou řadu.

[5] S výjimkou přímky. U ní stačí znát souřadnice pouze dvou bodů.



Obrázek 2.7: začlenění aproximační metody do grafu

Obrázek 2.7 ukazuje rozhraní *IFunctionApproximation*, které implementují všechny metody aproximace funkcí. Uživatel komponenty je však nezadává sám, ale pouze volí jednu z metod obsaženou ve výčtu *ApproximationMethod.* Z toho důvodu pro něj není vlastnost *FunctionApproximation* přístupná, ale je viditelná pouze z komponenty[6], protože ji bude využívat třída vykreslující graf XY.

[6] Přesněji pouze z assembly s komponentou. Použit modifikátor *internal*, v některých jazycích bývá také označován jako *friend.*

3 Implementace komponenty

Tato kapitola popisuje způsob implementace navržené komponenty. Blíže přibližuje některá specifika vybraného jazyka a použitých metod. Také se zabývá jejich limitacemi a problémy, které při jejich použití vznikly.

3.1 Výběr jazyka

Se způsobem kompilace do jazyka MSIL, který .NET používá souvisí také jazyková nezávislost vytvořených komponent. Z toho důvodu je pak možné využít komponentu napsanou v např. v jazyce C# jazykem C++, VB .NET apod. K tomu, aby bylo možné tuto vlastnost použít, musí být stanovena určitá sada sémantických pravidel, která bude stejná pro všechny tyto jazyky. Tím se ovšem ztrácí výhody výkonnosti některých jazyků. Rozdíly mezi jejich výkonem se snižují, a tak bez ohledu na to, jaký zvolíme jazyk, bude výkon výsledného programu přibližně stejný.

Výkonnost jazyka tedy můžeme jako jeden z aspektů výběru vyřadit. Jestliže se všechny jazyky v .NET řídí stejnou sadou základních sémantických pravidel, tak už se dá jen těžko hovořit o tom, který jazyk je pro tento konkrétní případ vhodnější. V tom případě už zbývá to poslední, podle čeho si můžeme jazyk vybrat a to je jeho syntaxe.

Jazyk tedy volíme tak, aby se nám co nejlépe četl. Také podle toho, jestli jsme se s ním už někdy setkali a jak jej známe.

Já jsem si v tomto případě pro implementaci komponenty zvolil jazyk C#. Jedním z důvodů této volby byla jeho syntaxe připomínající jazyk C, se kterou jsem se již několikrát setkal ať už v rámci studia, nebo ve svých osobních aktivitách. Druhým důvodem je má předchozí positivní zkušenost s tímto jazykem. Oproti C++ navíc disponuje možností rozdělení třídy do více souborů, což při vhodném a logickém rozdělení také přispívá k jeho čitelnosti.

Kromě výběru jazyka je tu další nutnost určit si verzi .NET, která bude použita pro implementaci. Zvolil jsem .NET 2.0 i přesto že v době psaní této práce existuje i verze 3.5. Důvodem pro volbu této starší verze je kompatibilita se staršími systémy MS Windows[7].

3.2 Členění kódu

Po vytvoření projektu komponenty s názvem ChartComponent se vytvoří její třída rozdělena do dvou zdrojových souborů a to *ChartComponent.cs a ChartComponent.Designer.cs*. Soubor ChartComponet.Designer.cs má využití především v případě, kdy skládáme tento prvek uživatelského

[7] Verze .NET Framework 3.0 a 3.5 jsou navrženy hlavně pro spolupráci s operačním systémem Windows Vista. Ze starších verzí podporují pouze Windows XP a Windows Server 2003.

rozhraní z prvků již existujících. Soubor ChartComponent.cs je určen pro vlastní editaci. Tato třída je ovšem velmi velká a tak má cenu ji rozdělit dále. Soubor ChartComponent.cs bude obsahovat metody a všechny vlastnosti budou v novém souboru *ChartComponent.Properties.cs*.

3.3 Jmenný prostor System.Drawing a jeho vhodnost pro vykreslování grafů

Jmenný prostor *System.Drawing* obsahuje třídy a struktury pro vykreslování základních obrazců, tvarů a textu. Jde jej využít nejen pro vykreslování na displej, ale i pro tiskárnu. Svým způsobem vytváří rozhraní pro snadnější užívání GDI+. Pro pokročilejší funkce je nutné, využít i některý jmenný prostor, který jej rozšiřuje.

Pro tento projekt je nejdůležitější *System.Drawing.Drawing2D*, který poskytuje především větší množství různých štětců, které lze využít k tvorbě různých typů výplní.

Vykreslování se provádí přes metody třídy *Graphics*. Mimo jiné obsahuje i prostředky pro omezení prostoru, do kterého se má vykreslovat (clip) a také pro transformaci souřadnic. Transformace přijdou vhod právě pro vykreslování grafů, pomohou umístit jeho počátek a usnadní tak některé výpočty.

3.3.1 Průběh vykreslování

Vykreslování v .NET probíhá ve 2 fázích:

- Vykreslení pozadí
- Vykreslení popředí

Jinak řečeno, pokud se má komponenta překreslit tak reaguje na dvě události. První z nich je *OnPaintBackgroud*, druhá *OnPaint*. V tomto smyslu také rozlišuje třída UserControl v .NET dvě stejnojmenné metody, které na tyto události reagují. Jejich druhým argumentem je objekt typu *PaintEventArgs*, ze kterého se pak přistoupí k objektu třídy Graphics, který se využije pro kreslení.

Pokud se nějaký objekt překresluje velmi často, tak dochází k nepříjemnému problikávání. Proto poskytuje .NET možnost využít optimalizovaný *double buffer*[8]. Který problikávání zredukuje. Nicméně pokud chceme, aby komponenta poskytla získání obrázku s grafem o jakýchkoliv rozměrech, tak vestavěný double buffer nevyužijeme a vytvoříme si vlastní bitmapu, do které budeme vykreslovat. To vše provedeme v metodě OnPaint. V takovém případě ale může dojít k dalšímu probliknutí, které způsobuje chování metody OnPaintBackgroud, která smaže předchozí obrázek dříve, než je k dispozici bitmapa, která ho má nahradit. Proto musíme základní chování metody

[8] Vykreslení se provede nejprve do bitmapy v paměti. Ta se pak celá překreslí na zobrazovanou plochu.

OnPaintBackgroud přepsat tak aby nic neprováděla a pozadí vykreslit v reakci na událost OnPaint. To sebou nese několik nevýhod, ale také několik výhod.

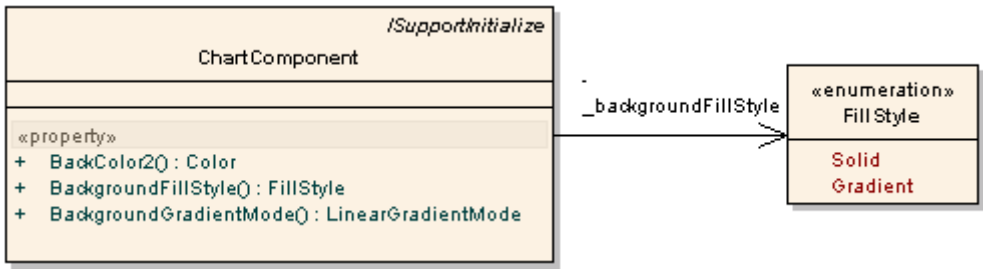
Tím že jsme potlačili výchozí vykreslování pozadí, tak musíme znovu implementovat vyplnění pozadí stejnou barvou či obrázkem. Obrázek navíc může být na pozadí vykreslen dlaždicově, uprostřed, v levém horním rohu, roztažený, nebo roztažený se zachováním poměru stran. To vše budeme muset vytvořit znova.

Ale přepsání metody pro vykreslování pozadí má i své výhody. Můžeme například komponentě přidat možnost použití přechodové výplně na pozadí.

3.3.2 Vykreslování pozadí

Úkolem je tedy vyřešit problém vykreslování pozadí jak byl popsán v části 3.3.1. Na to komponenta využívá metodu *FillBackground(Graphics, int, int),* v souboru ChartComponent.cs.

Vlastnosti *BackColor*, *BackgroundImage* a *BackgroundImageLayout* jsou zděděny z třídy UserControl. Pro rozšíření zavádí komponenta další vlastnosti, kterými jsou *BackColor2*, *BackgroudFillStyle* a *BackgroundGradientMode*. Ty jsou použity v případě, kdy má být pozadí vyplněno přechodovou výplní. Pokud je obrázek pozadí nastaven tak jej použije a vyplní jím pozadí dle stylu, který je vybrán v BackgroundImageLayout. Pokud není nastaven žádný obrázek, pak je pozadí vybarveno. Styl určuje vlastnost BackgroudFillStyle, ze které se vybere, jestli se má použít jedna barva, nebo přechod. V případě, že se použije přechod, tak se využijí barvy z vlastností BackColor a BackColor2. Další vlastností je BackgroundGradientMode, která určuje směr přechodu.



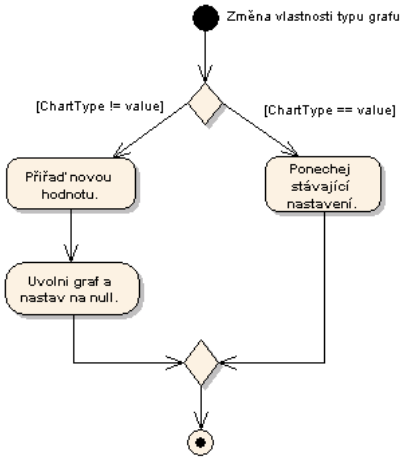
Obrázek 3.1: Vlastnosti, které přibudou pro styl pozadí.

3.4 Uchování grafu

3.4.1 Změna typu grafu

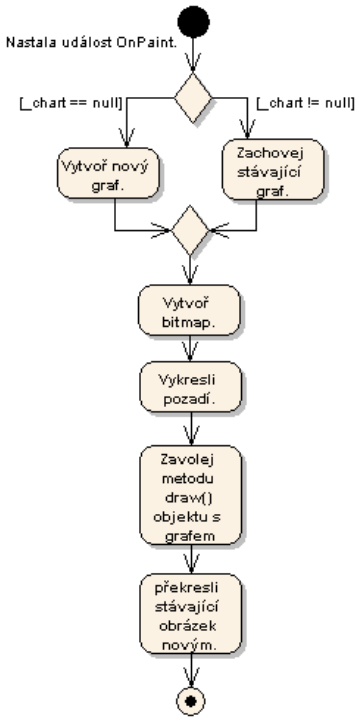
Návrhem typu grafů jsme se zabývali v části 2.2. Pro samotný objekt s grafem je tedy vytvořena proměnná, která objekt s grafem obsahuje a výčet pro změnu typu grafu. Graf se tedy mění přes vlastnost *ChartType.* Pro getter této vlastnosti stačí pouze vrátit její hodnotu. Což ovšem již neplatí pro její setter.

Při změně typu grafu je totiž nutné změnit obsah proměnné s grafem. Jeden ze způsobů, jak toho dosáhnout je nastavit ji prázdnou referenci a objekt s grafem vytvořit při požadavku na překreslení komponenty. Průběh změny typu grafu zobrazuje diagram na obrázku 3.2.



Obrázek 3.2: změna typu grafu

Objekt s grafem se tedy vytváří při překreslení grafu a to pouze pokud ještě není vytvořen. Respektive se testuje, zda je null. Pokud ano vytvoří se nový v závislosti na vybraném typu grafu. Jakmile je graf vytvořen, tak už nic nebrání tomu, aby byla zavolána metoda Draw()[9], která je součástí třídy tohoto grafu. Aktivitu, která pak s vykreslením grafu souvisí, znázorňuje diagram na obrázku 3.3. Z diagramu lze i vyčíst vlastní implementaci double bufferu.



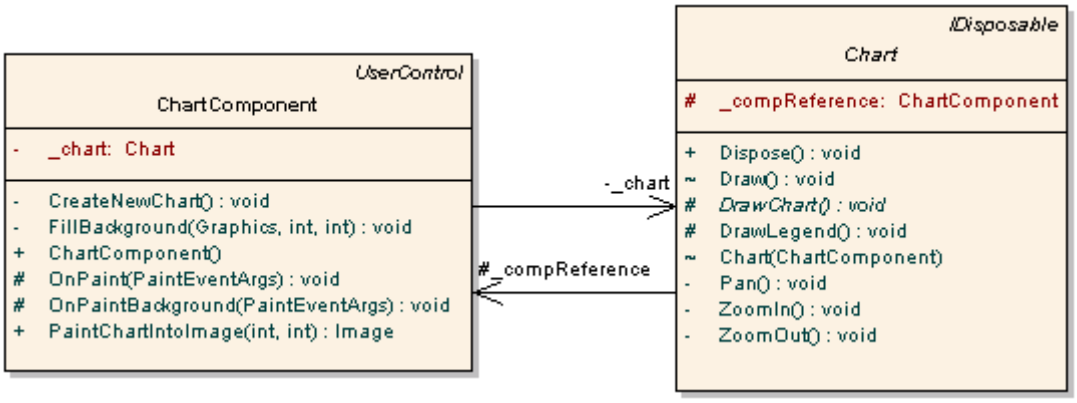
Obrázek 3.3: vytvoření grafu

[9] viz. Návrh typu grafů.

3.4.2 Přístup grafu k vloženým datům a nastavením

Část 2.3 se věnovala problematice zadávání dat grafové komponentě. Máme tedy vytvořené třídy, které je obsahují a ty jsou pak dále obsaženy v kolekci, která je součástí třídy komponenty. Otázkou je, jakým způsobem se k nim objekt s konkrétním typem grafu dostane. Nejedná se pouze o vložená data, ale také o všechny parametry[10], které lze nastavit komponentě pomocí Designeru.

Tento problém je vyřešen předáním grafu reference na komponentu. Při vytváření objektu s grafem z třídy komponenty se jednoduše předá reference jeho konstruktoru použitím klíčového slova *this*[11]. Objekt s grafem následně získá přístup nejen k datům, ale i ke všem ostatním nastavením komponenty. Vzájemné propojení demonstruje obrázek 3.4.



Obrázek 3.4: Vzájemné reference mezi třídy grafu a komponenty

3.4.3 Přibližování grafu

Komponentu jsme navrhli tak, aby se s některými typy grafů dali provádět operace přiblížení a posunování přiblíženého grafu. Nyní se tedy musí vyřešit, jakým způsobem bude uživatel požadavky na tyto operace zadávat.

Jako nejlepší řešení mi přišel výběr nástroje z kontextového menu, které se objeví po klepnutí myši pravým tlačítkem na komponentu. Z něj si uživatel vybere nástroj pro přiblížení, oddálení, nebo posunutí grafu. Pro každý nástroj je také potřeba vytvořit ukazatel myši.

Protože používáme prvky, jako je kontextové menu a kurzor, které mají přístup k systémovým zdrojům, tak musíme implementovat rozhraní *IDisposable*. Toto rozhraní obsahuje metodu *Dispose()*, ve které tyto prvky uvolníme.

O přibližování grafu bude ještě řeč později, nejdříve se musí vyřešit jeho rozložení.

[10] Ne všechna tato nastavení musí konkrétní graf nutně využívat. O některých jsme si něco pověděli již dříve v části věnované Vykreslování pozadí, na další přijde řeč později.
[11] V některých jazycích, se místo *this* používá *me*.

3.5 Vykreslování grafu

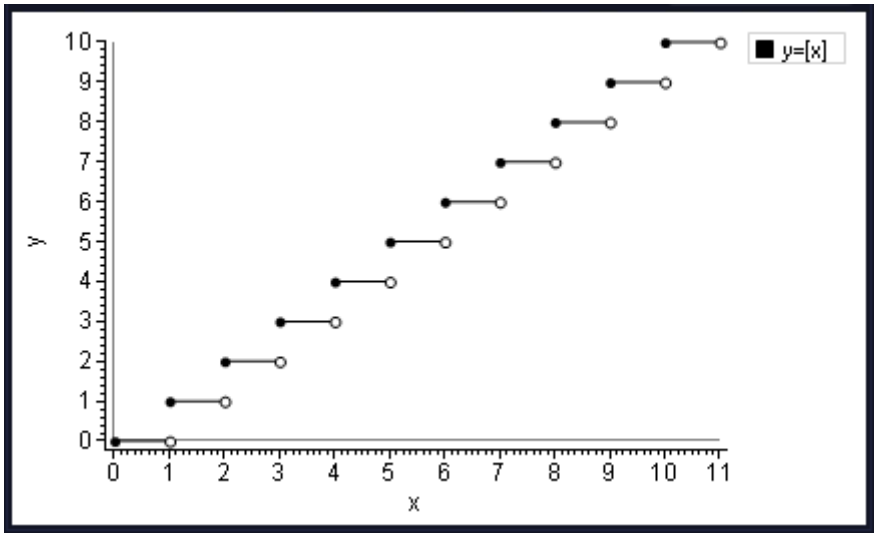
Už víme jakým způsobem je graf v komponentě uchován a jak se volá metoda pro jeho vykreslení. Nyní přichází řada na to, abychom si pověděli, jakým způsobem se graf vykresluje do předané bitmapy.

3.5.1 Rozložení grafu v předané ploše

První věcí, kterou je potřeba udělat při vykreslování grafu je rozložit si plochu, do které budeme kreslit. Je tedy nutné určit místo, kde bude legenda, kde bude obdélník s grafem apod.

V grafických odvětvích se při rozdělování takovýchto částí používá nejčastěji metoda třetin, protože je toto rozdělení pro lidský zrak nejpřirozenější. V našem případě by to znamenalo rozdělit si plochu jak horizontálně, tak i vertikálně na třetiny. Plocha s grafem by pak zabírala dvě třetiny obrázku z levé strany. Oblast s legendou zase jednu třetinu z pravé strany.

Nicméně při použití takovéto metody bychom neušetřili moc místa. Nejvíce by to bylo poznat při malých velikostech komponenty. Našim úkolem je tedy rozložit vše na ploše tak aby byl graf co největší. Na obrázku 3.5 je vidět, co všechno musí komponenta zobrazovat. Nalevo od obdélníku vymezující graf se nachází popisek osy Y a také její stupnice. Dole je to samé pro osu X. Napravo leží legenda. Chceme tedy, aby tyto prvky byly co nejmenší a přitom se nepřekrývali.



Obrázek 3.5: Graf funkce celá část

3.5.1.1 Stupnice a popisky os

Výška plochy pod osou X se vypočítá celkem jednoduše. .NET nabízí metodu *MeasureString()*, kterou pro tento účel využijeme. Výsledná výška je tedy dvojnásobkem výšky textu při použití fontu, který je pro popisky os zvolen.

To už však není tak jednoduché pro osu Y. Pro její popisek získáme velikost tak, že změříme jeho velikost a použijeme výšku. U stupnice ale nemůžeme změřit jen tak jakoukoliv hodnotu. Záleží

zde totiž na rozsahu osy. Číslo 0 bude zabírat méně místa, než 100. A i to bude kratší než -100. Bereme tedy v úvahu nejmenší a největší hodnotu osy a podle širší z nich určíme šířku stupnice. Tu pak připočítáme k výšce popisku a tím jsme získali odsazení grafu z levé strany.

3.5.1.2 Legenda

Pro legendu je situace opět poněkud jiná. Jsou zde totiž 2 důvody, kvůli nimž bude možná nutné změnit velikost fontu, který je pro ni nastaven:

- Nechceme, aby legenda zabírala zbytečně moc místa. Mohlo by to mít špatný vliv na velikost grafu, který by se příliš zúžil.
- Pokud bude mít legenda hodně položek, tak potřebujeme, aby se do grafu vešla celá.

Musíme tedy změřit velikost všech řetězců, které mají být v legendě zapsané. Pro určení šířky legendy spočítáme maximum ze všech šířek řetězců, které obsahuje, a k němu připočteme velikost dalších prvků[12], které obsahuje její jeden řádek. Vypočítané maximum se pak porovná s její maximální šířkou, kterou určíme tak, aby tvořila jednu třetinu šířky celé komponenty.

3.5.1.3 Prvky ohraničující graf

Rozložení máme tedy určeno, ještě zbývá si určit obdélník, do kterého bude graf vměstnán. Vůči poloze a velikosti tohoto obdélníku budou transformovány souřadnice tak, aby se do něj graf přesně vešel. Protože chceme, aby některé typy grafu podporovali funkci přiblížení, tak je nejvhodnější si tyto obdélníky určit dva.

První z nich bude přesně vymezovat ohraničení oblasti s grafem. To znamená, že se jeho poloha a velikost spočítají z odsazení vůči celé ploše komponenty.

Druhý z nich bude určovat plochu, na kterou se bude graf vykreslovat. Jeho velikost bude násobkem[13] velikosti již zmíněného obdélníku. A jeho poloha bude určena jako posunutí oproti prvnímu obdélníku. Vůči tomuto obdélníku budou také transformovány souřadnice tak, aby nejnižší hodnota na ose X byla jeho nejlevější okraj. Největší zase pravý okraj a podobě pro osu Y.

3.5.2 Transformace souřadnic

Máme tedy určeno kam se má graf vykreslit a máme rozmístěny i ostatní prvky na ploše komponenty. V tomto stádiu by jako výchozí bod pro kreslení byl levý horní roh komponenty. Stejně tak by jednotka v grafu odpovídala výšce či šířce jednoho pixelu. Proto musíme provést dvě transformace souřadnic. První je potřeba pro posunutí bodu s počátkem. Druhá pro změnu měřítka grafu a k jeho vertikálnímu převrácení.

[12] V jednom řádku legendy jsou navíc vykresleny čtverečky, které určují barvu toho, co legenda popisuje. Navíc je ještě potřeba brát v úvahu i odsazení.
[13] Závisí na velikosti přiblížení. Čím větší přiblížení, tím větší tento obdélník bude.

3.5.2.1 Posunutí bodu s počátkem

Tato transformace je vestavěná v GDI+. V .NET se použije zavoláním metody *TranslateTransform(),* která je součástí třídy Graphics. Jedním jejím parametrem je posunutí na ose X, druhým posunutí na ose Y. Zbývá tedy vyřešit jak posunout tento bod tak, aby byl tam, kde chceme mít počátek grafu.
V této době zatím ještě není změněno měřítko grafu, takže nemůžeme jednoznačně určit, kde se tento bod nachází a tak budeme muset tuto transformaci provádět dvakrát. Postup je následující:

1. Posuneme bod s počátkem tak, aby odpovídal souřadnici levého horního rohu obdélníku, v jehož rozsahu se graf vykresluje[14].
2. Aplikujeme změnu měřítka.
3. Podle maximálních, nebo minimálních hodnot, kterých graf dosahuje, posuneme počátek soustavy souřadnic na své místo.

Třetí krok se liší v závislosti na typu grafu. Například sloupcový, nebo spojnicový graf může dosáhnout záporných hod pouze na ose Y. A tak se posune pouze jeho souřadnice osy Y. U grafu závislosti X a Y se počátek posune o maximální hodnotu shora a minimální z leva. Pro koláčový graf se mohou první 2 body úplně vynechat a bod se může posunout zrovna do jeho středu.

3.5.2.2 Změna měřítka soustavy souřadnic

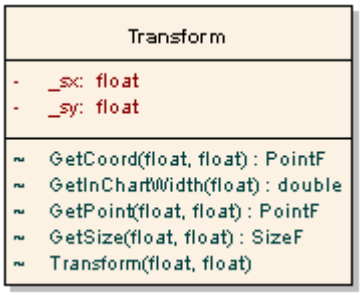
Jejím úkolem je změnit měřítko grafu tak, aby se rozsah hodnot na obou osách vešel do velikosti obdélníku, do kterého se graf vykresluje. Dále taky změnit směr osy Y, která je při běžném vykreslování orientována dolů. To znamená, že pokud bod leží níže, tak je jeho souřadnice na ose Y vyšší. Což ovšem neodpovídá klasickému zobrazení grafů, kdy při umísťování bodu výš jeho souřadnice na ose Y roste.

Pro tento úkol můžeme v .NET použít metodu *ScaleTransform()*, která si bere 2 parametry. Prvním je měřítko osy X. Druhým měřítko osy Y. Dosazením záporného čísla zároveň změníme orientaci osy.

Jenže tato metoda neprovádí transformaci přesně tak, jak bychom si ji přáli. To proto, že aplikuje zvětšení až po vykreslení požadovaných tvarů. Což znamená, že vepsaný řetězec může být zvětšen či zmenšen. Co je ale nejhorší, tak tato transformace ovlivní také šířku pera. Následkem toho pak například čára, která spojuje body spojnicového grafu, může být hodně široká, nebo by obtažení sloupců ve sloupcovém grafu mohlo být příliš tlusté apod. To co my potřebujeme je pouze přepočítat souřadnice bodů z výchozí do cílové soustavy souřadnic. Takovou transformaci si ale musíme vytvořit sami.

[14] Nejedná se tedy o obdélník, který pouze ohraničuje viditelnou oblast grafu, ale o ten, který je vzhledem k němu zvětšen, či posunut.

Řešením je tedy vytvořit třídu, která bude tyto transformace provádět. Měla by tedy nést informace o měřítku os. A obsahovat metody pro přepočet souřadnic z výchozí do cílové soustavy souřadnic a naopak.

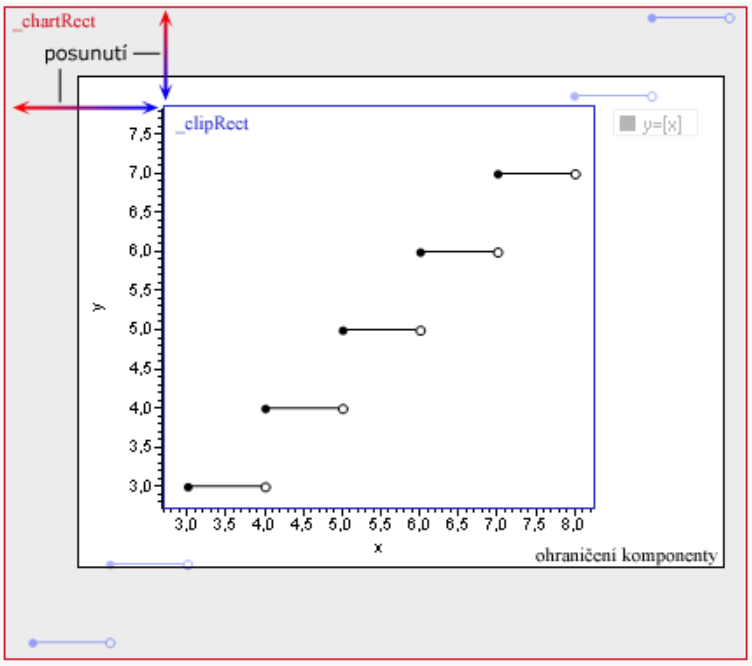


Obrázek 3.6: Třída, která provádí přepočty mezi soustavami souřadnic

Tato třída je znázorněna na obrázku 3.6. V konstruktoru se jí předá poměr zvětšení pro obě osy. Pokud chceme přepočítat souřadnice bodu do souřadnic plochy, na kterou vykreslujeme, tak využijeme vzniklého objektu a zavoláme metodu *GetPoint()*. Metoda *GetCoord()* provádí to samé opačným směrem. Ostatní metody jsou popsány v dokumentaci.

3.5.3 Přibližování

V části 3.5.1.3 bylo nastíněno využití dvou obdélníků pro vykreslování grafu. Nyní je čas se na něj podívat blíže.



Obrázek 3.7: přiblížení grafu

První obdélník, který ořezává graf je v třídě Chart označen jako *_clipRect*. Druhý z nich, vůči kterému se graf kreslí je označen jako *_chartRect*. Jeho rozměry jsou vůči obdélníku _clipRect zvětšeny násobkem proměnné *_zoom* a je posunut o *_pan*. Velikost a umístění obdélníku _chartRect

si určuje třída implementující konkrétní typ grafu. A podle jeho pozice a rozměrů pak provede transformace souřadnic. Ještě než se začne vykreslovat, tak se podle obdélníku _clipRect omezí plocha, do které se bude vykreslovat. To se provede použitím metody *SetClip()* třídy Graphics.

Samotné operace s grafem se pak provádějí v závislosti na tom, který nástroj si uživatel vybral. O způsobu výběru nástroje je zmínka v části 3.4.3. Metody přiblížení a oddálení reagují na klepnutí levého tlačítka myši. Kromě zvětšení proměnné zoom se musí také přepočítat, jakým způsobem zvětšení ovlivní posunutí grafu a vycentrovat bod, na který uživatel klepnul. To samé platí i pro oddálení. Posunutí se provádí při pohybu myši a současného stisku levého tlačítka. Tyto operace provádí v třídě Chart metody *ZoomIn(), ZoomOut()* a *Pan().*

3.6 Vykreslování jednotlivých typů grafu

Nyní už můžeme přejít k vykreslování konkrétních grafů. Každému z nich bude věnována některá podkapitola. Některé postupy jsou shodné či podobné u více typů grafů, proto budou vždy vysvětleny pouze v jedné podkapitole. Pokročilejším možnostem některých grafů se pak dále věnuje kapitola 3.7.

3.6.1 Sloupcový graf

U dat tohoto typu grafu se nebere v úvahu souřadnice osy X, ale jen souřadnice osy Y. Je jediný, u kterého má smysl vkládat data do legendy oběma způsoby[15]. Tzn. vypsat do legendy popis jednotlivých sloupců, nebo celých datových řad. Metody pro vykreslování sloupcového grafu obsahuje třída *BarChart.*

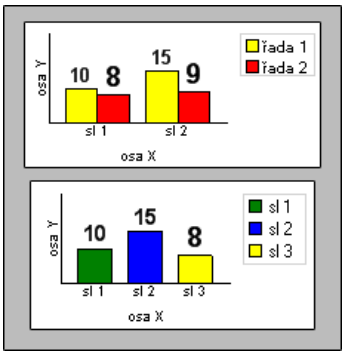
3.6.1.1 Vykreslování

Vykreslování řídí metoda *DrawChart().* Ta nejdříve transformuje souřadnice, pak zavolá metody pro vykreslení vnitřní části grafu a os. Rozsah osy Y se určí dle rozsahu hodnot grafu. Pokud jsou všechny hodnoty větší než nula, tak se za nejmenší hodnotu bere nula. Stejně je tomu tak i pro největší hodnotu, pokud jsou všechny hodnoty menší než 0. Rozsah osy X se určí podle počtu sloupců a počtu řad. Výška[16] obdélníku _chartRect se pak vydělí rozsahem osy Y, šířka zase rozsahem X. Tím se pro sloupcový graf získá poměr zvětšení pro obě osy. Tyto hodnoty se pak použijí pro vytvoření objektu třídy Transform, jehož metody pak budou používány pro přepočet souřadnic potřebných bodů. Jeho první využití je už pro vypočítání souřadnic počátku.

Vykreslování vnitřní části grafu provádí metoda *DrawBars(),* která kromě toho, že vykresluje sloupce a číslo s jejich hodnotou taky vytváří kolekci, která je později použita pro vyplnění údajů na stupnici osy X. Které údaje budou použity pro vyplnění stupnice osy X záleží na zvoleném zdroji

[15] Viz. vlastnost *LegendAndColorsSource* z hlavní třídy komponenty.
[16] Protože se nad sloupci vykresluje ještě řetězec, který udává jeho hodnotu, je nejdříve nutné výšku tohoto obdélníku snížit o výšku tohoto textu. I to je jeden z důvodů proč se tento obdélník nastavuje až v dědící třídě.

popisků legendy. V případě, že vkládáme do legendy popis celé datové řady, pak jsou sloupce se stejným pořadím v řadě, popsány dohromady jmény položek z první datové řady. V případě, že vkládáme do legendy popisky jednotlivých položek, pak je každý sloupec popsán zvlášť dle jména příslušné položky.



Obrázek 3.8: způsoby popisu osy X u koláčového grafu

Rozdíl mezi těmito dvěma způsoby je patrný z obrázku 3.8. Graf nahoře obsahuje dvě datové řady o 2 položkách. V první řadě jsou položky se jmény sl1 a sl2, v druhé sl3 a sl4. Jako zdroj barev a legendy mu byla vybrána datová řada, a protože se pro popis osy používají pouze položky z první datové řady, tak popisky sl3 a sl4 nejsou zobrazeny. Dolní graf obsahuje jednu datovou řadu, ve které jsou položky sl1 až sl3 a jako zdroj barev a legendy mu byla zvolena datová položka.

3.6.1.2 Možnosti formátování

Kromě vlastností, které jsou pro všechny typy grafů společné, umožňuje určit font a formát řetězce, který zobrazuje hodnoty nad[17] každým sloupcem. Font se nastavuje ve vlastností *YValuesFont* a formát řetězce vlastností *YValuesStringFormat*.

3.6.2 Spojnicový graf

Spojnicový, nebo taktéž čárový graf má konstantní posunutí osy X. Co se týče polohy bodu, tak neznázorňuje závislost osy Y na ose X, ale pouze hodnoty, které má osa Y. Jako zdroj dat do legendy pro něj dává smysl pouze datová řada. Metody pro jeho vykreslování obsahuje třída *LineChart*.

3.6.2.1 Vykreslování

Vykreslování řídí metoda *DrawChart()*. Rozsah osy Y se určí dle rozsahu hodnot grafu. Rozsah osy X je dán počtem položek v nejdelší datové řadě. Další transformace se pak provádějí podobným způsobem jako u sloupcového grafu. Protože spojnicový graf podporuje přiblížení, tak je ještě před transformacemi nutné změnit velikost obdélníku _chartRect. To se provede vynásobením jeho šířky a výšky obsahem proměnné _zoom a posunutím jeho polohy o obsah horizontální a vertikální složky proměnné _pan.

[17] Nebo pod sloupcem, pokud je jeho hodnota záporná.

Vykreslení vnitřní části grafu je na metodě *DrawLines()*. Body jsou vždy vzdáleny o jednotku na ose X. Pro osu Y se použije její hodnota z položky.

Stupnice osy X je vykreslena stejným způsobem, jako u sloupcového grafu, který má vybrán jako zdroj barev a legendy datovou řadu. U osy Y je způsob zcela odlišný. Stupnice osy Y je popsána čísly od nejmenší hodnoty, která se v datech grafu nachází, po největší hodnotu. Musí se tedy dopočítat ostatní hodnoty. Počet hodnot, které budou v jednom okamžiku viditelné, určuje vlastnost komponenty s názvem *YAxisNumbersCount*. Pokud nastavíme její hodnotu například na 5, potom bude ve stupnici osy při jakémkoliv přiblížení vidět pět číslic. Stupnice jde mezi jednotlivými číslicemi ještě dále rozdělit na jednotlivé segmenty. Jejich počet určuje vlastnost *YAxisSegmentsPerNumber*. Vykreslování osy probíhá v cyklu, před kterým se určí krok z těchto dvou vlastností. Od nejmenší hodnoty se krok postupně zvětšuje a při každé iteraci se vykreslí na osu čárka. Pokud platí podmínka, že zbytek po celočíselném dělení počtu proběhlých iterací počtem segmentů mezi čísly je roven nule, pak je vykreslena i číslice s aktuální hodnotou.

Tento postup je však ještě o něco upraven. Bylo by totiž zbytečné plýtvání časem procesoru, pokud by se muselo vykreslovat od nejnižší hodnoty až po hodnotu nejvyšší a spoléhat se na to, že co neleží v oblasti, pro kterou je nastaven clip, nepůjde vidět. Vykreslení grafu by pak trvalo příliš dlouho, obzvláště pak při velikých hodnotách přiblížení. Proto tento cyklus obsahuje ještě jednu úpravu, kdy testuje, zda daný bod leží v obdélníku _clipRect. Pokud v něm neleží, pak se stupnice nevykresluje. Tato úprava výrazně urychluje vykreslování os.

3.6.2.2 Možnosti formátování

Kromě možností, které má společné s ostatními typy grafů, umožňuje stejným způsobem formátovat popisky hodnot přímo v grafu.

Mimo to ještě dovoluje nastavit formát řetězce, kterým je zobrazeno číslo na ose Y. Což se kvůli počítání čísel, které leží v rozsahu, stává nutností. To hlavně proto, že toto číslo může mít mnoho desetinných míst a tak by se mohlo v grafu zobrazit v nepřehledném tvaru. Formátování těchto řetězců se nastavuje vlastností *YAxisStringFormat*, která se nachází ve třídě komponenty.

Tento ty grafu dále nabízí možnost změnit odsazení bodů v grafu zleva, zprava a ze spodu pomocí vlastnosti *FirstPointOffset*, *LastPointOffset* a *BottomOffset*.

Vlastností *MarkSize* se nastavuje velikost čtverečku, který zvýrazňuje jednotlivé body. Toto zvýraznění jde vypnout pomocí vlastnosti *ShowPointMarks*.

3.6.3 Koláčový graf

U koláčového grafu odpadá potřeba vykreslování stupnice os a z části také transformace souřadnic. Naopak zase přibývají další výpočty pro rozmístění a otočení jednotlivých částí koláče. Tento typ grafu neznázorňuje datové řady, a tak by měla být při jeho použití nastavena jako zdroj barev a legendy položka. Koláčový graf representuje třída *PieChart*.

3.6.3.1 Vykreslování

Protože tento typ grafu využívá jiné rozložení na ploše, tak má vlastní implementaci metody *Draw()*. Ta ihned po spočítání rozložení zavolá metodu *DrawChart()*, která řídí vykreslování grafu. Pro vykreslení jednotlivých výsečí koláče můžeme využít metody *DrawPie()* a *FillPie()* třídy Graphics. Kromě pera či štětce je potřeba pro jejich vykreslení znát ještě následující údaje:

- Obdélník, kterému je vepsána elipsa, jejíž výseč vykreslujeme.
- Úhel, který udává natočení výseče.
- Úhel, který udává šířku výseče

Jako obdélník pro vepisování můžeme využít obdélníka _chartRect. Jednotlivé úhly budeme muset před vykreslením každé výseče spočítat. Nejprve k tomu budeme potřebovat součet všech hodnot všech částí koláče, ten si označíme písmenem *l*. Hodnotu aktuální výseče jako y_k. Úhel natočení označíme jako φ_k a úhel, který udává šířku jako α_k. Poslední věcí, kterou budeme muset znát je pozice aktuální výseče, kterou označíme jako p_k. Úhly ve stupních se pak vypočítají následujícím způsobem:

$$l = \sum_{k=1}^{n} y_k$$

$$p_1 = 0$$

$$p_k = p_{k-1} + y_{k-1}$$

$$\varphi_k = 360\frac{p_k}{l}$$

$$\alpha_k = 360\frac{y_k}{l}$$

Kde n je počet všech výsečí. Tímto způsobem vypočítané úhly se pak použijí v uvedených metodách.

3.6.3.2 Vykreslování popisků do grafu

Tato komponenta umí vykreslit popisky grafu přímo do jednotlivých výsečí koláče. Jeden způsob, kterým by se toho dalo dosáhnout je otočení[18] soustavy souřadnic. Nicméně při použití tohoto typu transformace se otočí i směr textu. V takovém případě by pak text ležící v úhlu 180° byl svisle i vodorovně převrácen a nebyl by čitelný. Proto je potřeba spočítat souřadnice bodů, na nichž bude text umístěn. Určíme si tedy kružnici, na níž budou body ležet. Nejprve se vypočítá úhel φ, pod kterým jde bod ze středu kružnice vidět. A po té se podle poloměru s využitím goniometrických funkcí spočítají souřadnice bodu. Výpočty se provedou následujícím způsobem (l, p_0, p_k, y_k,jsou stejné, jako v předchozím případě):

$$\alpha_k = 2\pi\frac{y_k}{l}$$

[18] Funkce *RotateTransform()* třídy Graphics.

$$\varphi_k = 2\pi\frac{p_k}{l} + \frac{1}{2}\alpha$$

$$x_k = \cos\varphi_k * r$$

$$y_k = \sin\varphi_k * r$$

Kde r je poloměr kružnice, a úhly jsou udány v radiánech. Tyto předpisy využijeme pro vypočítání souřadnic bodů, na kterých se text nachází.

3.6.3.3 Možnosti formátování

Stejně jako graf sloupcový a spojnicový umožňuje i koláčový graf formátovat vepsaný text. Dále umí přepočítat hodnoty na procenta, či zobrazit v grafu pouze jméno položky, nebo v něm nezobrazit nic. Jedná se o vlastnosti *DisplayNames, DisplayValues* a *DisplayValuesAsPercents*. Všechny jsou obsaženy v třídě ChartComponent.

3.6.4 Graf XY

Graf XY je ze všech grafů této komponenty nejkomplexnější. Obsahuje totiž nejvíce pokročilých funkcí. O těchto funkcích pojednává kapitola 3.7 a každé z nich je věnována samostatná podkapitola. Tento graf znázorňuje závislost os X a Y a tak je vhodný pro zobrazení funkcí z přírodních věd jako je například matematika, nebo fyzika.

3.6.4.1 Vykreslování

Jeho vykreslování začíná metodou DrawChart(). Pro osu X se rozsah hodnot určí z její nejnižší a největší hodnoty, stejně je tomu i pro osu Y. Ještě než se začne s transformováním souřadnic, tak je potřeba změnit rozměry a umístění obdélníku _chartRect v závislosti na proměnných _zoom a _pan.

Vnitřní část grafu vykresluje metoda *DrawXY()*. Protože tento graf podporuje různé nastavení propojení[19] bodů, musí se tato metoda dále dělit. V případě, že si uživatel vybral propojení bodů úsečkami, je zavolána metoda *DrawPointsConnectedByLine(),* pro propojení křivkou *DrawPointsConnectedByCurve(),* pro isolované body *DrawPoints()* a v případě nezobrazování bodů nezavolá nic.

Stupnice obou os jsou vykresleny stejným způsobem jako osa Y u spojnicového grafu.

3.6.4.2 Možnosti formátování

První možností formátování je způsob propojení bodů. Tuto možnost lze využít z třídy ChartData, která representuje datovou řadu, nastavením vlastnosti *PointConnection*.

Další možností je manuální rozsah os. Komponenta pro něj definuje vlastnost s názvem *ManualAxisInterval*. Ten se pak nastavuje pomocí vlastností *MaxX, MaxY, MinX, MinY*.

[19]Propojení bodů už bylo zmíněno v kapitole 2.4.2.

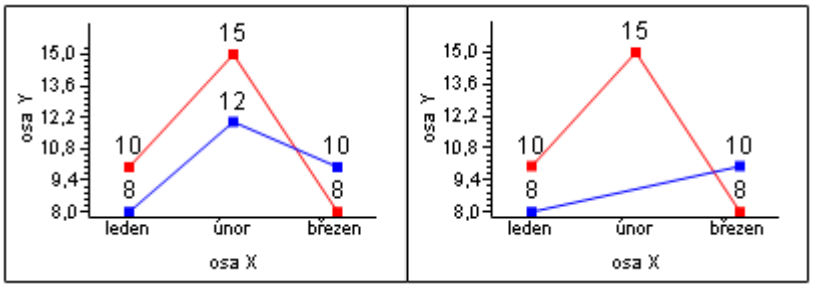
Pak je zde ještě možnost vypnutí zobrazování značek, které znázorňují náležitost bodu k intervalu. Této funkci grafu je dále věnována kapitola 3.7.3. Pro zapnutí, či vypnutí těchto značek definuje komponenta vlastnost *DrawIntervalMarks.*

3.7 Pokročilé možnosti některých grafů

Kapitola 3.7 popisuje další možnosti, které jsou některými grafy nabízeny a nebyly zařazeny do kapitoly 3.6 buď proto, že se jedná o vlastnost, kterou sdílí více grafů zároveň. Nebo proto, že je jejich implementace složitější.

3.7.1 Vynechání položky

Možnost vynechání položky implementuje spojnicový a sloupcový graf. Protože se jednotlivé položky datové řady vykreslují ihned za sebe, tak by bez této možnosti nebylo možné zobrazit graf, pro který bude některá z položek chybět.



Obrázek 3.9: vynechání položky

Vynechání demonstruje obrázek 3.9. V grafu vlevo není vynechána žádná z položek. V grafu vpravo je v datové řadě, označené modrou barvou, vynechána položka pro únor.

Její vynechání se provede nastavením vlastnosti *Use*, třídy ChartDataItem na hodnotu false.

3.7.2 Asymptoty

U asymptot je potřeba si určit, jakým způsobem je budeme uchovávat. Jedná se o přímku a ta je sice jednoznačně určena dvěma body, ale my nechceme, aby byly spojeny pouze tyto 2 body. V takovém případě bychom viděli pouze úsečku, která bude ležet někde uprostřed grafu. Co tedy musíme provést je zjistit úhel, který svírá asymptota s některou s os. Dále je pak potřeba znát posunutí[20] na některé z os. Poté je potřeba si zvolit jednu ze souřadnic dvou bodů, které leží v minimální a maximální hodnotě grafu. Podle úhlu dále dopočítat druhé souřadnice těchto bodů. Aby se přepočet na úhel neprováděl vždy, když je potřeba asymptotu znovu vykreslit, tak je lepší uchovávat přímo daný úhel a posunutí.

[20] Buď y souřadnici průsečíku s osou Y, nebo x souřadnici průsečíku s osou X.

Pro asymptotu, která je rovnoběžná s osu X, musíme posunutí určit na ose Y. To naopak platí i pro asymptotu rovnoběžnou s osou Y. Proto je potřeba si určit, kdy se které posunutí použije. V této komponentě platí, že se vždy použije posunutí na ose X a pouze v případě, že je asymptota rovnoběžná s osou X, tak se použije posunutí na ose Y.

3.7.2.1 Zadávání asymptoty

Asymptotu representuje třída *Asymptote*, která v sobě uchovává úhel a obě posunutí. Takto je možné asymptotu zadat pomocí jejích dvou konstruktorů. Jeden z nich si bere úhel, druhý k němu ještě přidává posunutí na ose X. To ovšem nejsou jediné 2 konstruktory, které tato třída obsahuje.

Ještě je zde jeden další, který umožňuje asymptotu zadat pomocí dvou bodů, jimiž prochází. Ten však musí spočítat úhel i posunutí. Pokud si body označíme jako $P[p_1;p_2]$ a $Q[q_1;q_2]$, pak se úhel φ, který asymptota svírá s osou x určí následujícím způsobem:

$$\vec{u} = (q_1 - p_1; q_2 - p_2)$$

$$\vec{v} = (0; 1)$$

$$\cos\varphi = \frac{\vec{u}\vec{v}}{|\vec{u}||\vec{v}|}$$

Vektor **v** je směrovým vektorem osy X. Dále ještě musíme určit posunutí. Pokud je asymptota rovnoběžná s osou X, pak můžeme její posunutí na ose Y snadno určit z y souřadnice jednoho ze zadaných bodů. V takovém případě totiž platí, že $p_2=q_2$. Jestliže tomu tak není, vypočítáme průsečík asymptoty s osou X z jejich parametrických rovnic. Pro osu x platí:

$$x = t$$

$$y = 0$$

Pro asymptotu platí:

$$x = p_1 + su_1$$

$$y = p_2 + su_2$$

Řešíme tedy soustavu rovnic a po několika úpravách se dostaneme na tvar:

$$s = -1\frac{p_2}{u_2}$$

$$x = p_1 + su_1$$

Kde x je hledaná hodnota posunutí. Objekty s asymptotou se do grafu vkládají pomocí metody *AddAsymptote()* třídy, která representuje datovou řadu.

3.7.2.2 Vykreslování

K jejímu vykreslení si musíme nejprve určit 2 body, které později spojíme úsečkou. Jejich určení je závislé na směru asymptoty.

Pokud je asymptota rovnoběžná[21] s některou z os. Pak jednu souřadnici těchto bodů tvoří minimální a maximální hodnota této osy. Druhá souřadnice je u obou bodů stejná a je jí hodnota posunutí na druhé ose.

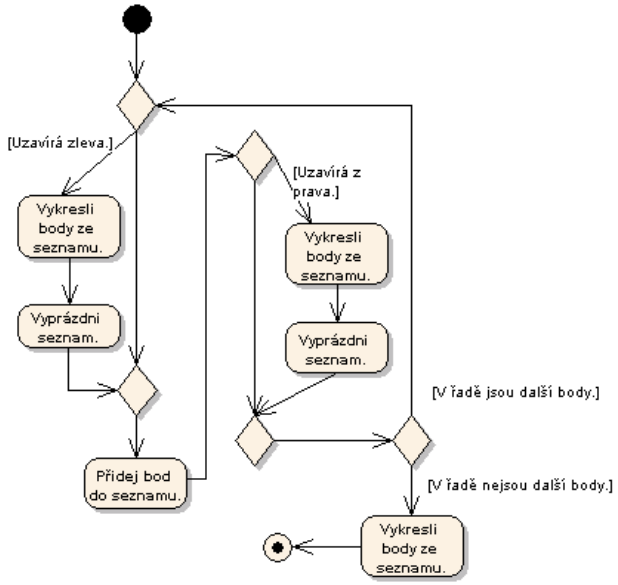
Pokud je asymptota s oběma osami různoběžná, tak určíme oběma bodům souřadnici osy Y. Pro první to bude minimální hodnota grafu, pro druhou zase maximální hodnota. Souřadnici osy X pak vypočítáme ze vzorce $x = \frac{y}{\tan \varphi}$, kde y je souřadnice daného bodu pro osu Y a φ je úhel, který asymptota svírá s osou X. Na závěr pak u obou bodů ještě k souřadnici x přičteme posunutí na ose X.

Asymptotu pak vykreslíme použitím metody *DrawLine()* třídy Graphics. O vykreslování asymptot v třídě XYChart se stará metoda *DrawAsymptotes()*.

3.7.3 Náležitost bodu k intervalu

Jejím návrhem se zabývala kapitola 2.4.3, její implementace spočívá ve změně funkcí, které vykreslují čáry XY Graf. První z nich je funkce DrawPointsConnectedByLine(), druhou je funkce DrawPointsConnectedByCurve(). Kromě vykreslování značek, které zvýrazňují, jestli daný bod do intervalu patří či nikoliv, musí tato funkce také rozdělovat spojnici bodů v závislosti na tom, ze které strany bod interval uzavírá.

Musíme si tedy postupně shromažďovat body, které mají být spolu propojeny a ty spolu vykreslovat. K tomu, abychom tak mohli učinit, si musíme nejprve určit, za jaké podmínky se má bod přidat k předchozím bodům a za jaké podmínky k následujícím. Pokud se má bod přidat k předchozím, pak nesmí uzavírat interval zleva. Pokud se má přidat k následujícím, pak nesmí uzavírat interval zprava.



Obrázek 3.10: Vykreslování bodů po intervalech

[21] Pokud je rovnoběžná s osou X, pak je její úhel roven $0 + k\pi, k \in Z$. Pokud je rovnoběžná s osou Y, pak je její úhel roven $\frac{\pi}{2} + k\pi, k \in Z$.

Postup, kterým je tato situace řešena, je znázorněn diagramem v obrázku 3.10, který pro jednoduchost ukazuje pouze řešení situace, ze které strany bod interval uzavírá, ale už se nezabývá tím, jestli do něj patří.

K tomu je potřeba vytvořit další 2 seznamy. Do prvního budeme vkládat body, u kterých má být zvýrazněno, že do intervalu nepatří. Do druhého zase ty, u kterých má být zvýrazněno, že do něj patří. Pak vykreslíme do grafu kružnici, pro nepatřící body, či vyplněný kruh pro patřící. Tuto akci provádí v třídě XYChart metoda *DrawIntervalMarks*.

3.7.4 Aproximace funkcí

Návrhem začlenění aproximačních metod do grafu se zabývala kapitola 2.4.4. Každá z metod je representována samostatnou třídou, která implementuje rozhraní *IFunctionApproximation*. Toto rozhrání obsahuje pouze jednu metodu s názvem *GetData()*. Úkolem této metody je vypočítat a vrátit datovou řadu, která obsahuje takové body, jejichž spojením vznikne křivka (přímka) aproximační funkce. Tato metoda má tři parametry. Dva udávají rozsah, pro který se bude aproximace počítat. Třetí udává krok, což znamená rozestup dvou sousedních bodů na ose X. Čím tedy bude krok menší, tím bude hodnota aproximace spočítána pro více bodů.

Body, které se mají aproximovat, se předávají konstruktoru. Ten tedy jako svůj parametr přijímá třídu, která representuje datovou řadu. Tímto způsobem se v něm vypočítají pomocné prvky, které zůstanou v objektu aproximační metody uchovány.

Tyto třídy využívá graf XY, který vždy zadá krok s rozsahem a z daného objektu pak dostane zpět data, která už jen vykreslí pospojovaná pomocí úseček. Vždy počítá jen viditelný rozsah. Využije tedy metody *GetCoord()* z třídy Transform a jako souřadnice ji zadá hraniční body obdélníku _clipRect. Tímto způsobem dostane viditelný rozsah osy X, který pak použije při volání metody GetData(). Krok je zadán ve třídě reprezentující datovou řadu a to sice jako vlastnost s názvem *ApproximationStep*. Ten svým způsobem určuje kvalitu křiky a je zadán v počtu obrazových bodů. Je tomu tak z důvodu, aby se při velkém přiblížení tato křivka nejevila příliš hranatá. Zbývající akce, která se tedy musí udělat před zavoláním GetData(), je přepočítání tohoto kroku do jednotek grafu.

Každá metoda však má jistá omezení a tak je při jejím nastavení nutné ověřit, zda je pro zadané body použitelná. V případě, že nelze dané body aproximovat pomocí vybrané funkce, je při jejich přidávání do grafu vrácena výjimka.

Princip získávání dat je tedy znám a tak už můžeme přejít k implementaci jednotlivých metod.

3.7.4.1 Newtonův interpolační polynom

Má tu vlastnost, že zadanými body prochází. Tyto body se pak nazývají jako *uzlové body*. Podmínkou jeho výpočtu je, že všechny jeho uzlové body musejí být navzájem různé. Jestliže platí, že:

$$n_n(x) = \prod_{k=0}^{n}(x - x_k)$$

Kde n_n je báze polynomu. Pak se Newtonův interpolační polynom, který je zadán n uzlovými body určí jako:

$$P_n(x) = a_0 + \sum_{k=1}^{n} a_k n_{k-1}$$

Což lze také zapsat ve tvaru:

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0)(x - x_1)\dots(x - x_{n-1})$$

Kde a_0 je funkční hodnotou pro x_0. Zbylé koeficienty se pak vypočítají pomocí tzv. poměrných diferencí, pro které platí následující vztahy:

$$f[x_0] = f(x_0)$$

$$f[x_i, x_{i+1}, \dots, x_{i+k}] = \frac{f[x_i, x_{i+1}, \dots, x_{i+k-1}] - f[x_{i+1}, x_{i+2}, \dots, x_{i+k}]}{x_i - x_{i+k}}$$

Už víme, že $a_0 = f(x_0) = f[x_0]$. Pro ostatní koeficienty tedy platí: $a_1 = f[x_0, x_1]$, $a_2 = f[x_0, x_1, x_2]$, ..., $a_n = f[x_0, x_1, \dots, x_n]$. Pro zjednodušení výpočtu poměrných diferencí se často využívá tabulka:

i	x_i	$f(x_i)$	$f[x_i,x_{i+1}]$	...	$f[x_i,x_{i+1},\dots x_{i+k}]$
0	x_0	$f(x_0)$	$f[x_0,x_1]$		$f[x_0,x_1,\dots x_n]$
1	x_1	$f(x_1)$	$f[x_1,x_2]$		
...	...	...	$f[x_{n-1},x_n]$		
n	x_n	$f(x_n)$			

Koeficienty $a_0 \dots a_n$ jsou pak obsaženy v prvním řádku tabulky. V komponentě provádí výpočet poměrných diferencí konstruktor třídy *NewtonsPolynomial*, přičemž pro tabulku využívá dvojrozměrného pole. Po ukončení výpočtu jsou koeficienty obsaženy v prvním řádku tohoto pole. Metoda GetData() pak využije těchto koeficientů a bodů $x_0 \dots x_n$ pro výpočet hodnot v jednotlivých bodech.

3.7.4.2 Metoda nejmenších čtverců pro přímku

Tato metoda proloží zadané body přímkou. Pro tyto body však musí platit podmínka, že $\exists k \leq n$, $pro\ které\ platí$: $x_k \neq x_i$, $kde\ i = \{0,1 \dots, k-1, k+1, \dots n\}$. Což znamená, že alespoň jedno x

musí být různé od ostatních. Vypočítaná přímka pak danými body neprochází, pouze se snaží je co nejtěsněji minout. Pro každý bod se chyba[22] vypočítá vztahem:

$$e_i = y_i - f(x_i)$$

Pro přímku, která je určena jako $y = c_0 + c_1 x$ se tento vztah dále upraví:

$$e_i = y_i - c_0 - c_1 x_i$$

Tato metoda používá jako kritérium pro nejlepší proložení co nejmenší součet druhých mocnin chyb v daných bodech. Odtud se tedy bere název metoda nejmenších čtverců. Toto kritérium lze vyjádřit následujícím matematickým vztahem:

$$\rho^2 = \sum_{i=0}^{n} {e_i}^2 = \sum_{i=0}^{n} [y_i - f(x_i)]^2 = \sum_{i=0}^{n} (y_i - c_0 - c_1 x_i)^2$$

Kde ρ^2, které někdy bývá také označováno jako π se nazývá minimální kvadratická odchylka. V rovnici jsou y_i a x_i dány, c_0 a c_1 jsou koeficienty, které musíme vypočítat. Pro jejich výpočet je nutné položit parciální derivace ρ^2, dle těchto koeficientů rovny 0.

$$\frac{\partial(\rho^2)}{\partial c_0} = 2\sum_{i=0}^{n} (y_i - c_0 - c_1 x_i) = 0$$

$$\frac{\partial(\rho^2)}{\partial c_1} = 2\sum_{i=0}^{n} x_i (y_i - c_0 - c_1 x_i) = 0$$

Úpravou těchto rovnic se pak dostaneme na tvar, který se nazývá normální rovnice.

$$c_0(n+1) + c_1 \sum_{i=0}^{n} x_i = \sum_{i=0}^{n} y_i$$

$$c_0 \sum_{i=0}^{n} x_i + c_1 \sum_{i=0}^{n} {x_i}^2 = \sum_{i=0}^{n} x_i y_i$$

Řešení této soustavy rovnic provádí konstruktor třídy *LeastSquaresLine*. Nejprve si určí všechny sumy. Soustavu rovnic pak vyřeší pomocí Cramerova pravidla a následně si uloží kořeny c_0 a c_1. Metoda GetData() pak vypočítá ze vzorce $y = c_0 + c_1 x$ dva body, které vrátí v datové řadě.

3.7.4.3 Metoda nejmenších čtverců pro parabolu

Jak její název napovídá, tak se tato metoda velmi podobá metodě nejmenších čtverců pro přímku. Jediný rozdíl je v tom, že se pro dosazení do obecné rovnice pro chybu aproximace použije rovnice pro parabolu. Jestliže je parabola dána vztahem $y = c_0 + c_1 x + c_2 x^2$, pak je její minimální kvadratická odchylka rovna:

$$\rho^2 = \sum_{i=0}^{n} {e_i}^2 = \sum_{i=0}^{n} [y_i - f(x_i)]^2 = \sum_{i=0}^{n} [y_i - (c_0 + c_1 x_i + c_2 {x_i}^2)]^2$$

Její parciální derivace podle c_1, c_2, c_3 pak opět položíme rovny nule:

[22] Vzdálenost v bodě x od funkce, která body prokládá. V tomto případě od přímky.

$$\frac{\partial(\rho^2)}{\partial c_0} = 2\sum_{i=0}^{n}[y_i - (c_0 + c_1 x_i + c_2 {x_i}^2)] = 0$$

$$\frac{\partial(\rho^2)}{\partial c_1} = 2\sum_{i=0}^{n} x_i[y_i - (c_0 + c_1 x_i + c_2 {x_i}^2)] = 0$$

$$\frac{\partial(\rho^2)}{\partial c_1} = 2\sum_{i=0}^{n} {x_i}^2[y_i - (c_0 + c_1 x_i + c_2 {x_i}^2)] = 0$$

Úprava těchto rovnic vede na normální rovnice pro výpočet koeficientů paraboly:

$$c_0(n+1) + c_1\sum_{i=0}^{n} x_i + c_2\sum_{i=0}^{n} {x_i}^2 = \sum_{i=0}^{n} y_i$$

$$c_0\sum_{i=0}^{n} x_i + c_1\sum_{i=0}^{n} {x_i}^2 + c_2\sum_{i=0}^{n} {x_i}^3 = \sum_{i=0}^{n} x_i y_i$$

$$c_0\sum_{i=0}^{n} {x_i}^2 + c_1\sum_{i=0}^{n} {x_i}^3 + c_2\sum_{i=0}^{n} {x_i}^4 = \sum_{i=0}^{n} {x_i}^2 y_i$$

Soustavu těchto rovnic řeší konstruktor třídy *LeastSquaresParabola.* Ten nejprve určí všechny sumy a pak pokračuje řešením soustavy rovnic pomocí Cramerova pravidla. Funkce GetData() počítá dle vzorce $y = c_0 + c_1 x + c_2 x^2$ funkční hodnoty bodů daných rozsahem a krokem.

Aby bylo možné určit aproximaci pomocí metody nejmenších čtverců pro parabolu, je nutné, aby v původní funkci existovaly alespoň tři navzájem různé body.

4 Závěr

Výsledkem práce je kompletní grafová komponenta, která nabízí jednoduchý a flexibilní přístup k tvorbě grafů na platformě .NET. Obsahuje 4 typy grafů a to graf sloupcový, spojnicový, koláčový a graf XY. Většina jejich vlastností je nastavitelná pomocí Designeru a tak je její využití jednoduché. Komponenta podporuje i pokročilejší funkce, jako je zobrazení náležitosti bodu k intervalu, asymptoty či aproximace zadaných bodů funkcí. Využití těchto funkcí není složité, většinou je potřeba jen změnit nastavení některých vlastností, na hodnotu z příslušného výčtového typu. Funkce přiblížení navíc pomáhá uživateli zjistit přesnější informace o hodnotách zobrazených bodů. Dále komponenta umožňuje získání obrázku s grafem. Na ten se pak dají dále použít různé transformace a díky tomu jej pak lze začlenit i do pokročilejšího grafického rozhraní. K tomu přispívá i podpora různých grafických nastavení. Může se jednat například o změny fontů pro popisky na osách, v legendě nebo přímo pro popisky v grafu. Dále je tu také možnost umístění obrázku na pozadí, nebo vyplnění pozadí, či legendy přechodovou výplní. Pro rozlišitelnost a dobrou orientaci v grafu je navíc možné označit datové řady, nebo jejich položky různými barvami.

To vše z ní dělá prostředek pro využití v jednoduchých programech, které zobrazují statistické, či podobné informace. K tomu přispívá i možnost aproximace daných bodů funkcí, která pomůže zvýraznit trend, který graf zobrazuje.

Z předmětů, které jsem absolvoval na FIT, mi k tvorbě této komponenty nejvíce pomohly: Programování v .NET a C#, Tvorba uživatelských rozhraní, Základy počítačové grafiky, Principy programovacích jazyků a OOP, Numerická matematika a pravděpodobnost.

V budoucnu by bylo možné tuto komponentu dále rozšířit o další typy grafů. Dále by bylo možné upravit některé již existující typy. Například by se mohlo jednat o rozložený koláčový graf, nebo by se mohly přidat další aproximační metody. Dalším zajímavým rozšířením by mohl být i automatický formát řetězce pro číslice popisující stupnici os.

Literatura

[1] CHAPPEL, David. *Understanding .NET, Second Edition.* Addison Wesley Professional, 2006, 336 s. ISBN 0-321-19404-7.

[2] LOWY, Juval. *Programming .NET Components*. O'Reilly, 2005, 648 s. ISBN 0-596-10207-0

[3] MACDONALD, Matthew. *Pro .NET 2.0 Windows Forms and Custom Controls in C#*. Apress, 2006, ISBN 1-59059-439-8.

[4] SERBAN, I.; BREZOI, D.; RADU, T.; WARD, A. *GDI+ Custom Controls with Visual C# 2005*. Birmingham: Packt Publishing, 2006, ISBN 1-904811-60-4.

[5] HAMMILTON, Kim; MILES, Russell. *Learning UML 2.0.* O'Reilly, 2006, 286 s. ISBN 0-596-00982-8

[6] FAJMON, B.; RŮŽIČKOVÁ, I. *Matematika 3*. Brno: Skriptum FEKT VUT, 61-82 s.

[7] *.NET Framework Developer Center* [Online; navštíveno 20.12.2007].
URL http://msdn2.microsoft.com/cs-cz/netframework/default(en-us).aspx

[8] *MathWorld* [Online; navštíveno 8.1.2008]
URL http://mathworld.wolfram.com

[10] Hildyard, Alex. *Build a Reusable Graphical Charting Engine with C#*. [Online; navštíveno 3.11.2007]
URL http://www.devx.com/dotnet/Article/20077/0/page/1

Seznam příloh

Příloha A

Příklady použití komponenty

Tyto příklady demonstrují některá užití. Jsou uvedeny v jazyce C#. V příkladech se předpokládá užití jmenného prostoru *PavelRiedl.BP*.

1. VÝBĚR TYPU GRAFU

Výběr typu grafu se provádí nastavením vlastnosti *ChartType* na příslušnou hodnotu. Následující příklad ukazuje výběr spojnicového grafu:

```
ChartComponent chart = new ChartComponent();
Chart.ChartType = ChartType.Line;
```

Toto nastavení je přístupné i přes Designer, kde se nachází v sekci Behavior.

2. ZADÁVÁNÍ DAT DO GRAFU

Následující příklad ukazuje způsob, jakým lze komponentě *chart* z předchozího příkladu přidat dvě datové řady, z nichž první bude zvýrazněna červenou barvou, druhá modrou. Řada 1 bude mít pro leden hodnotu 12, řada 2 hodnotu 10. Pro únor budou mít hodnoty 15 a 13.

```
ChartData rada1 = new ChartData(Color.Red, "rada1");
rada1.AddItem(new ChartDataItem(12, "leden"));
rada1.AddItem(new ChartDataItem(15, "unor"));
ChartData rada2 = new ChartData(Color.Blue, "rada2");
rada2.AddItem(new ChartDataItem(10));
rada2.AddItem(new ChartDataItem(13));
chart.ChartDataList.Add(rada1);
chart.ChartDataList.Add(rada2);
```

3. VÝBĚR ZDROJE PRO LEGENDU

Následující příklad ukazuje styl výběru pro legendu u koláčového, či sloupcového grafu. Ostatní typy grafu využijí výchozí hodnotu. V Designeru je tato vlastnost obsažena v kategorii Legend.

```
chart.LegendAndColorsSource = LegendAndColorsSource.DataItem;
```

4. OMEZENÍ FORMÁTU ČÍSEL NA PEVNÝ POČET DESETINNÝCH MÍST

Lze jej využít pro stupnici os a také pro popisky přímo v grafu. Jedná se o vlastnosti *XAxisStringFormat, YAxisStringFormat* a *YValuesStringFormat*. Všechny je lze nastavit využitím Designeru. Následující příklad ukazuje omezení číslic osy X na 3 desetinná místa:

```
chart.XAxisStringFormat = "{0:0.000}";
```

5. PŘIDÁNÍ ASYMPTOTY DO XY GRAFU

Asymptota se přidává k objektu datové řady. V následujícím příkladě je jako asymptota přidána přímka *x=1*.

```
Asymptote asymptote = new Asymptote(new PointF(1,0), new PointF(1,1));
rada1.AddAsymptote(asymptote);
```

6. VÝBĚR TYPU PROPOJENÍ U XY GRAFU

Propojení se nastavuje pomocí vlastnosti PointConnection. Následující příklad nastaví Řadě 1 propojení bodů křivkou.

```
rada1.PointConnection = PointConnection.Curve;
```

7. NASTAVENÍ POZICE BODU V INTERVALU

Nyní příklad demonstruje pokus o nastavení neplatné hodnoty pozice v intervalu, při které nastane výjimka.

```
ChartDataItem item = new ChartDataItem(12, 32);
try{
    item.IntervalPosition = IntervalPositions.LeftOpen |
                            IntervalPositions.LeftClosed;
}
catch (ArgumentException e)
{
    ...;
}
```

8. NASTAVENÍ APROXIMACE FUNKCÍ

Pro nastavení aproximace se zadá řadě hodnota příslušného výčtu. Pokud řada obsahuje takové body, které vybranou metodou aproximovat nelze, nastane výjimka. Následující příklad ukazuje, jak se řadě nastaví aproximace polynomem.

```
try
{
    rada1.ApproximationMethod = ApproximationMethod.Polynomial;
}
catch (InvalidDataException e)
{
    ...;
}
```

9. PŘEPOČÍTÁNÍ APROXIMACE

Pokud dojde k tomu, že se změní pozice některého bodu, nebo se přidá bod další, pak je nutné aproximaci přepočítat. To se provede nastavením vlastnosti *RecalculateApproximation* na hodnotu true. Po přepočítání se tato vlastnost nastaví zpět na false. Následující příklad předpokládá, že je některá aproximační metoda už vybrána.

```
rada1.AddItem(new ChartDataItem(11, 10));
```

```
rada1.RecalculateApproximation = true;
```

Příloha B

Diagram tříd