Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Implementace modulu pro zpracování datových struktur

Jakub Klíma

Vedoucí práce: Ing. Ondřej Malík

6. května 2015

Poděkování

Děkuji svému vedoucímu práce Ing. Ondřeji Malíkovi za přínosné konzultace návrhu implementace a za cenné rady a kritiku textu této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 6. května 2015 .....

Odkaz na tuto práci

Klíma, Jakub. *Implementace modulu pro zpracování datových struktur*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato bakalářská práce se zabývá návrhem, implementací a testováním modulu pro stávající produkční aplikaci, napsaného v jazyce C#. Modul provádí zpracování, třídění a další operace s rozsáhlými datovými strukturami, které obsahují sloupce dat. Modul umožňuje nad těmito strukturami vytvářet datové pohledy, které slouží jako zdroje dat pro grafy a tabulky. Datové pohledy umožňují ze vstupních datových struktur sloupce vybírat, seskupovat a řadit je a provádět nad nimi různé agregace. Datové pohledy lze definovat pomocí uživatelsky čitelného zápisu ve formátu XML. Dále je v modulu zajištěna notifikace grafů a tabulek o změnách ve zdrojových datových strukturách. V bakalářské práci bylo dbáno na srozumitelnost a rozšiřitelnost implementace. Součástí práce je dokumentace veškerých veřejných rozhraní implementovaných tříd. V současné době s modulem pracují a dále ho rozšiřují ostatní programátoři.

Klíčová slova Implementace modulu, zpracování dat, stromové struktury, datové rozhraní pro grafy a tabulky, C#, .NET, XML

Abstract

This thesis deals with design, implementation and testing of a module for an existing production application written in C# language. The module performs processing, categorizing and other operations on large data structures that contain columns of data. The module allows creating data views on these structures that are used as data sources for charts and tables. The data views allow to select, group, sort and perform various aggregations on the columns of input data structures. The data views can be defined in user-readable notation in XML format. The module also notifies charts and tables of changes in source data structures. In the implementation, emphasis was placed on its clarity and extensibility. Documentation of public interfaces of all implemented classes is included. The module is being routinely used and extended by other programmers.

Keywords Module implementation, data processing, tree structures, data interfaces for charts and tables, C#, .NET, XML

Obsah

Seznam obrázků

Úvod

Zpracování datových struktur a jejich zpřístupnění ve vhodné formě uživateli je často řešená problematika mnoha aplikací. Tato práce se zabývá návrhem a implementací modulu, který řeší tuto problematiku pro stávající aplikaci.

Tato aplikace se zabývá výpočty v oblasti energetiky. Datové struktury aplikace jsou rozsáhlé a je tedy potřeba data vhodně zpracovat a následně je zobrazit a zpřístupnit uživateli. Pro zobrazení dat uživateli slouží zobrazovací komponenty aplikace, kterými jsou grafy a tabulky.

Zpracování dat je v této práci řešeno pomocí tzv. datových pohledů, přičemž každý z těchto pohledů nad daty provádí některou z operací jako je výběr podmnožiny dat, agregace nad daty, jejich seskupení a řazení apod. Datové pohledy mohou být řetězeny za sebe a výsledné složené operace nad daty pak lze reprezentovat jako stromové struktury těchto pohledů. Vybrané datové pohledy z těchto struktur pak mohou být použity jako zdroje dat pro tabulky a grafy aplikace. Stromové struktury datových pohledů je možné definovat pomocí uživatelsky čitelného, vhodně strukturovaného předpisu v souboru.

V práci je dále kladen důraz na kvalitu implementace a jsou zde diskutovány některé techniky jak takového cíle dosáhnout, neboť výsledný modul bude používán a dále rozšiřován ostatními programátory aplikace. Modul je implementován v jazyce C#.

Kapitola 1

Bližší specifikace

Tato kapitola se zabývá bližší specifikací dat, způsobu jejich zpracování a zobrazení.

1.1 Zobrazovací komponenty

Zobrazovací komponenty aplikace jsou – tabulkové zobrazení dat, tzv. datagrid a grafová komponenta zobrazující spojnicové grafy.

Grafy i datagridy mají v aplikaci vlastní oblast, kde se zobrazují – vlastní panel. Grafový panel i panel datagridu může obsahovat několik těchto komponent, které jsou zobrazeny na jednotlivých kartách daného panelu.

1.2 Data a jejich zpracování

Data mají povahu sloupců hodnot stejné délky – tabulek, které bývají zpravidla rozsáhlé jak do počtu sloupců, tak do počtu řádků. Tabulky mohou být indexovány časovým vektorem. Počet ani pořadí sloupců v tabulce nemusí být dopředu známé.

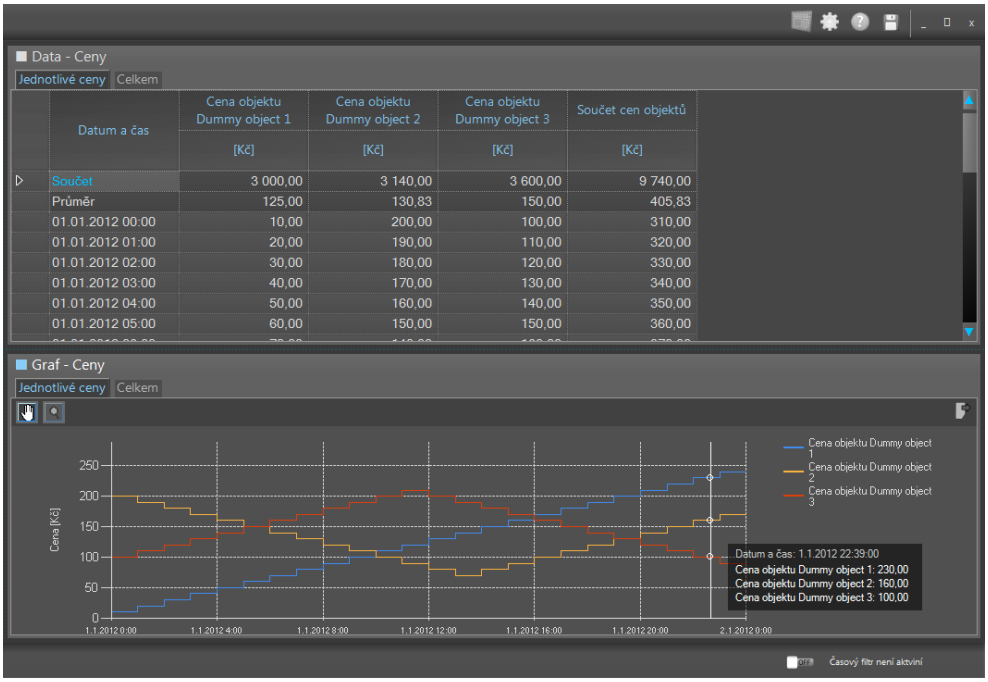
Data je potřeba určitým způsobem zpracovat, než se zobrazí uživateli. Například vybrat jaké ze sloupců zobrazit, provést nad nimi aritmetické operace, seskupovat je, či je řadit.

Zpracování dat by mělo být prováděno jednotně, nezávisle na tom, pro jakou zobrazovací komponentu budou použity.

Sloupce mají kromě hodnot i určité parametry, na základě kterých je možné sloupce identifikovat a provádět s nimi příslušné operace.

V některých sloupcích dat je možno měnit hodnoty. Ze zobrazovacích komponent toto umožňuje datagrid.

V datech tedy mohou nastat změny a to buď prostřednictvím zobrazovacích komponent, nebo jinak. Zobrazovací komponenty se musí o změně dozvědět, aby jejich obsah mohl být udržován aktuální. Například, pokud pro-



Obrázek 1.1: Snímek obrazovky zachycující zobrazení ukázkových dat ve stávající aplikaci

střednictvím datagridu změním hodnoty nějakého sloupce, tak graf, pokud tento sloupec také zobrazuje, se musí o změně dozvědět, aby mohl svůj obsah aktualizovat.

1.3 Definice toho, co se zobrazí

Jednotlivým datovým strukturám aplikace nebo jejich skupinám je potřeba definovat, co se pro ně má v datagridech a grafech zobrazit.

Tyto definice by měly být odděleny od zdrojového kódu aplikace, tedy v souboru a to především ze dvou důvodů. Mělo by je být možné jednoduše editovat a to bez nutnosti zásahu programátora. A dále také proto, že do budoucna se plánuje pro tvorbu těchto definic vytvořit uživatelský nástroj, který by mohl nad tímto typem souborů s definicemi operovat. Tento nástroj by umožnil uživateli vytvářet vlastní pohledy na data.

V těchto definicích bude možno určit kromě toho, co se v jednotlivých grafech a datagridech zobrazí i další potřebné nastavení těchto komponent.

1.4 Modul pro zpracování datových struktur

Pro výše uvedené účely byl implementován modul jako součást stávající aplikace, jehož vývoj je obsahem této práce.

Součástí modulu pak bude také definice rozhraní, skrze které budou poskytována zpracovaná data grafům a datagridům.

Modul je, stejně jako samotná aplikace, implementován v jazyce C#.

Kapitola 2

Cíl práce

Cílem práce je tedy navrhnout, implementovat, otestovat a zdokumentovat modul psaný v jazyce C#, který umožní z datových struktur aplikace připravit obsah zobrazovacím komponentám aplikace.

Součástí modulu bude:

1. Jednotné zpracování dat pro zobrazovací komponenty pomocí tzv. datových pohledů, které nad nimi umožní provádět operace výběru sloupců dat, provádění aritmetických operací nad sloupci dat a jejich seskupování a řazení.

2. Systém notifikace zobrazovacích komponent o změně v podkladových datech.

3. Možnost mimo kód předepsat tzv. schémata, která budou obsahovat to, co se má pro dané datové struktury zobrazit. Předpisy schémat budou v souboru o vhodně navrženém, uživatelsky čitelném formátu.

4. Rozhraní, které budou implementovat datové struktury aplikace, aby na ně mohly být aplikovány datové pohledy.

5. Rozhraní, skrze které budou zpracovaná data poskytována zobrazovacím komponentám.

Kapitola 3

Analýza, návrh a implementace modulu

Následující kapitola popisuje analýzu, návrh a implementaci modulu.

Vzhledem k tomu, že modul řeší několik různých problémů, je podle nich jeho popis rozdělen do několika částí. Toto rozčlenění je znázorněno na obr. 3.1. Čísla na obrázku korespondují s očíslováním jednotlivých částí.

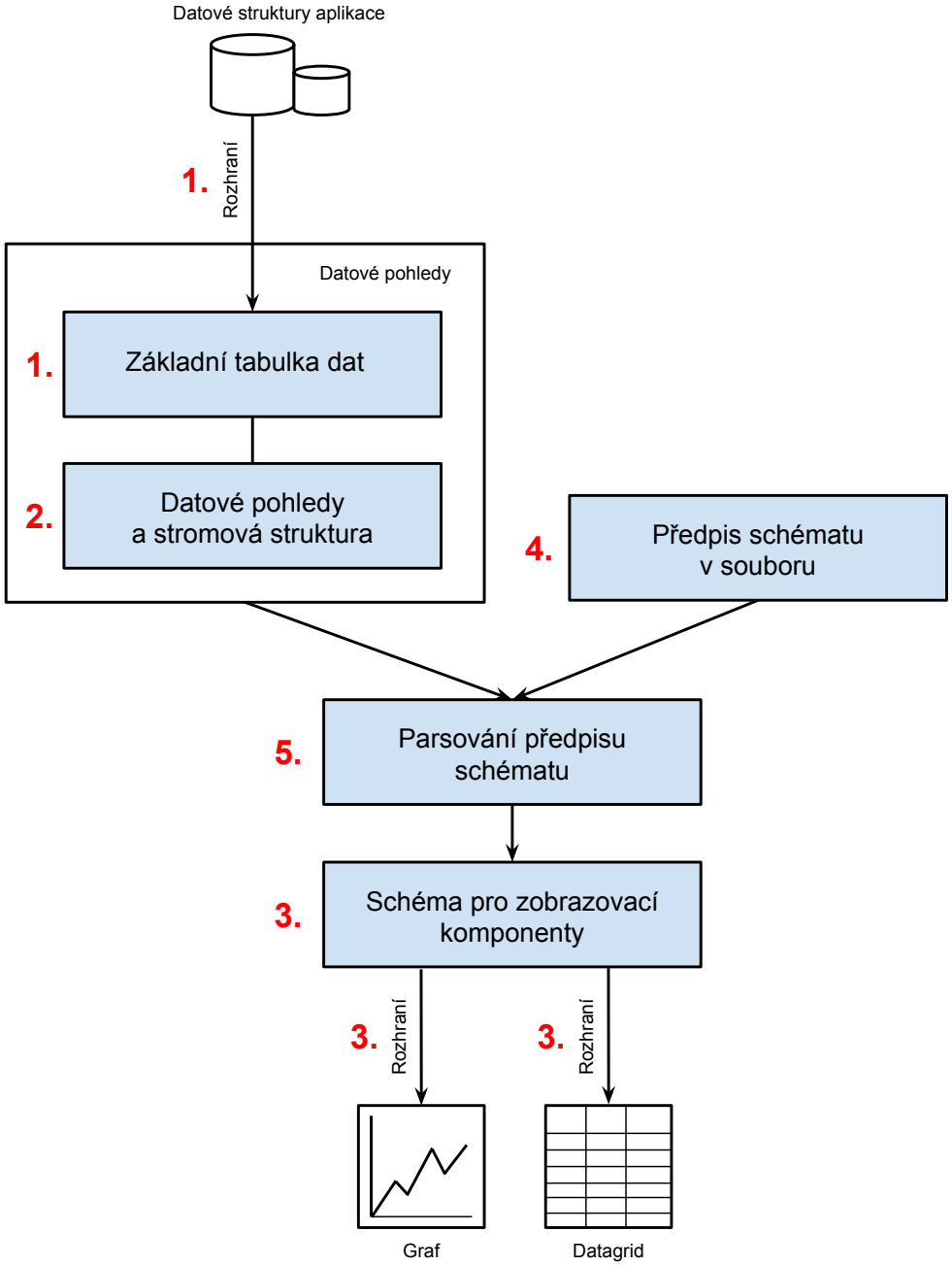
První část modulu řeší prvotní načtení dat z datových struktur aplikace do tzv. základní tabulky dat (viz sekce 3.1). Ta funguje jako zprostředkovatel dat pro následné provádění operací, pomocí již zmíněných datových pohledů. Sama totiž rozhraní datových pohledů implementuje. V této části je také probíráno rozhraní, ze kterého základní tabulka data čerpá – to je právě již zmíněné rozhraní, které musí implementovat datové struktury aplikace, aby na ně mohly být datové pohledy aplikovány.

Druhá část řeší jednotlivé datové pohledy, které mají zodpovědnost za provádění operací nad daty (viz sekce 3.2). Všechny tyto pohledy implementují jednotné rozhraní (to implementuje i základní tabulka dat). Jednotlivé pohledy se za sebe mohou navazovat, což umožňuje definovat posloupnost operací s daty a tím utváří stromovou strukturu. Je zde také diskutován systém notifikace zobrazovacích komponent pomocí událostí a jejich šíření v těchto stromových strukturách.

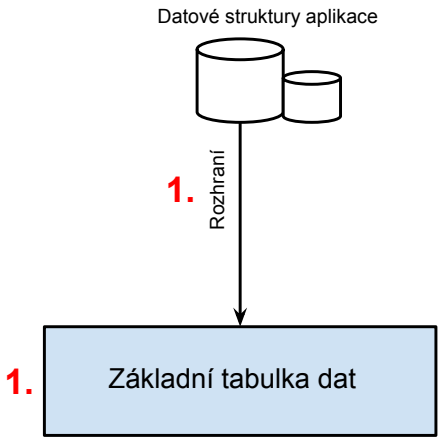
První i druhá část tedy popisují, jak je umožněno zpracovávat data pomocí datových pohledů.

Třetí část popisuje schéma, které poskytuje zpracovaná data zobrazovacím komponentám (viz sekce 3.3). Ve schématu jsou sestavené stromy datových pohledů a z nich vybrané uzly – datové pohledy sloužící jako zdroje dat pro grafy a datagridy. Dále zde jsou některá nastavení těchto zobrazovacích komponent, které jim jsou potřeba předat – např. styly hlaviček sloupců, popisky os grafu apod. Tato část zahrnuje i rozhraní, skrze která jsou data a nastavení zobrazovacím komponentám poskytovány.

Čtvrtá část se zabývá předpisem těchto schémat v souboru (viz sekce 3.4).



Obrázek 3.1: Diagram znázorňující rozčlenění modulu



Obrázek 3.2: Diagram znázorňující rozčlenění modulu – část 1, základní tabulka dat

Diskutuje se zde zvolený formát souboru a především to, jak je v něm předpis strukturován.

Závěrečná pátá část potom řeší, jak se tyto předpisy ze souboru zpracovávají (viz sekce 3.5). To znamená, jak se soubory parsují, jak se z nich sestavují stromy datových pohledů a jak se z nich vytváří příslušná schémata.

Následující podkapitoly se detailněji věnují analýze, návrhu a implementaci jednotlivých částí modulu. Na závěr, po bližším seznámení čtenáře s částmi modulu, je popsáno typické použití, které se o všechny probírané části opírá.

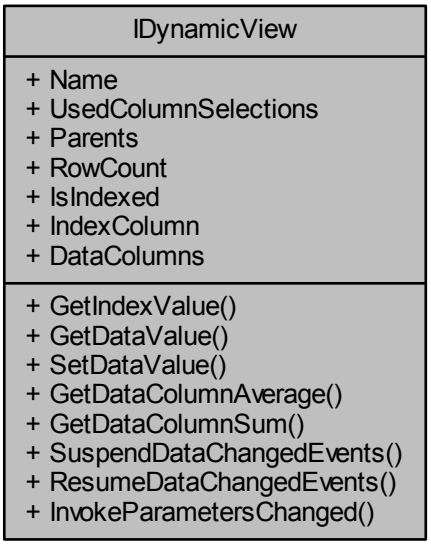
3.1 Základní tabulka dat a rozhraní s datovými strukturami aplikace

Tato část modulu zajišťuje prvotní načtení dat z datových struktur aplikace a připravuje z nich základní tabulku dat (viz obr. 3.2). Tato tabulka funguje jako zprostředkovatel dat pro následné provádění operací nad nimi. Operace nad daty budou prováděny pomocí datových pohledů, které sdílí společné rozhraní. Základní tabulka dat toto rozhraní již implementuje, tudíž se jedná také o datový pohled.

3.1.1 Zásady použité při popisu tříd a jejich diagramů

V následujících diagramech tříd je vždy popsáno především jejich veřejné rozhraní. Pomocné neveřejné metody a členské proměnné, jimiž je realizována implementace, jsou diskutovány jen ve vybraných případech.

Kromě metod se ve veřejném rozhraní tříd vyskytují i tzv. vlastnosti – C# Properties. Nejedná o veřejné členské proměnné tříd, které by třídy, z hlediska



Obrázek 3.3: Diagram třídy IDynamicView

zapouzdření, neměly obsahovat [1]. Jde v podstatě o zapouzdřené metody pro čtení nebo zápis. Co se týče jejich použití, chovají se podobně, jako kdybychom četli nebo zapisovali do proměnné. Jejich čtení a zápis je však implementováno pomocí getter a setter metod a nemusí korespondovat s implementací soukromých členských proměnných uvnitř třídy. Tudíž je zachována zapouzdřenost tříd oddělením jejich veřejného rozhraní a vnitřní implementace. Tyto vlastnosti pak mohou mít buď pouze getter – vlastnosti pouze pro čtení, nebo pouze setter – vlastnosti pouze pro zápis, nebo obojí [2].

3.1.2 Rozhraní datových pohledů

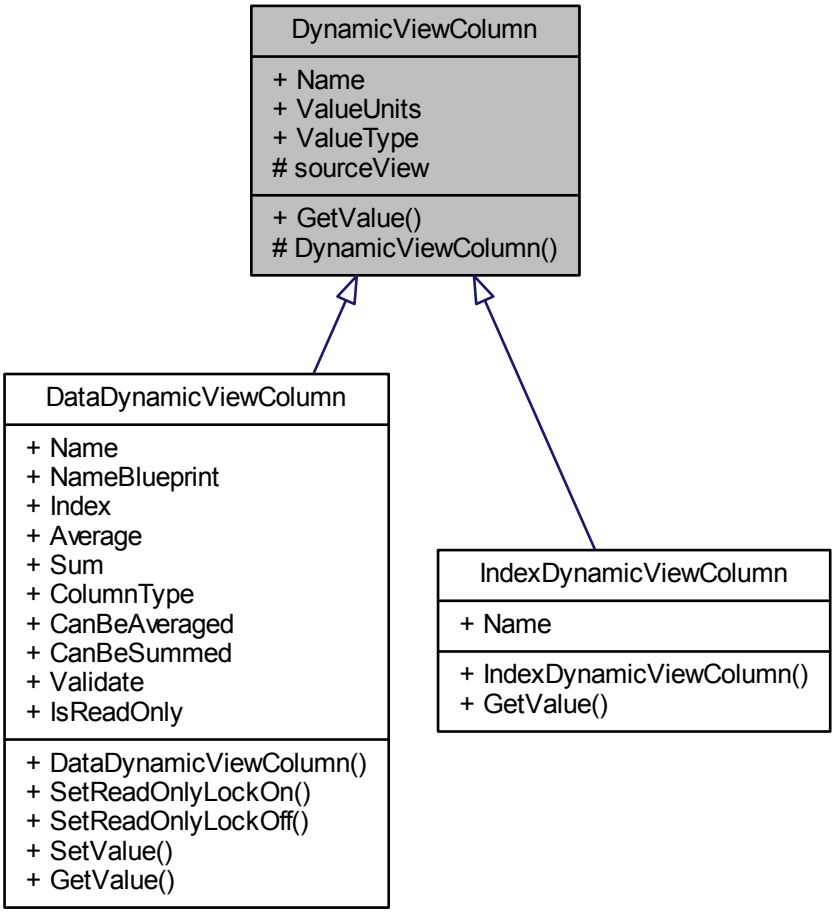
Hlavní částí rozhraní datových pohledů je poskytování dat v podobě tabulky. Zpravidla však pohledy data přímo neuchovávají a jen na ně poskytují jiný pohled. Proto je také rozhrání nazváno **IDynamicView**.

V následující části je popis základních rysů z rozhraní datových pohledů (viz obr. 3.3).

Každý datový pohled má své jméno – vlastnost **Name**, kterým může být pohled označen.

Přístup k datům je zajištěn metodou **GetDataValue(columnIndex, rowIndex)**, kde přes index sloupce a řádku lze získat hodnotu dané buňky aktuálního pohledu. Metodou **SetDataValue(value, columnIndex, rowIndex)** lze pak obdobně novou hodnotu buňky zapsat.

Informace o velikosti tabulky – jaké jsou povolené rozsahy indexů, lze získat skrze vlastnost **RowCount** pro počet řádků a počet sloupců dotázáním se na počet prvků vlastnosti **DataColumns**, která je reprezentována kolekcí.



Obrázek 3.4: Hierarchický diagram tříd DynamicViewColumn

Dále jsou v rozhraní poskytovány agregátní hodnoty jednotlivých sloupců, konkrétně součet a průměr. Ty jsou poskytovány metodami **GetDataColumnAverage(columnIndex)** a **GetDataColumnSum(columnIndex)**.

3.1.2.1 Datové sloupce pohledů

Veškeré parametry datových sloupců jsou obsaženy ve zmíněné kolekci **DataColumns**. Jednotlivé datové sloupce jsou v ní zde reprezentovány instancemi třídy **DataDynamicViewColumn** (viz obr. 3.4). Datové sloupce obsahují parametry sloupce a přístup k jeho hodnotám – datové sloupce (o kterých se píše i dále) tedy **hodnoty neobsahují, pouze k nim poskytují přístup**.

Datový sloupec obsahuje název – vlastnost **Name** a své pořadí v datovém pohledu – vlastnost **Index**.

Ve třídě je dále držena reference na daný datový pohled – členská proměnná **sourceView**. Pomocí ní poskytuje sloupec přístup k datům obdobně

jako samotný pohled, ale již bez nutnosti určení sloupce – pomocí metod **GetValue(rowIndex)**, **SetValue(value, rowIndex)** a vlastností **Sum** a **Average**.

Datový sloupec také poskytuje pomocné vlastnosti **CanBeSummed** a **CanBeAveraged**, které indikují, zdali agregace dává smysl nad daným sloupcem provést. Jejich hodnoty pouze odráží, jestli jsou agregované hodnoty rovny *null*.

Součet a průměr lze provést pouze pro sloupce s číselnými hodnotami, ale i u nich je potřeba určit, jestli dává smysl tuto hodnotu zobrazovat – např. součet nad hodnotami rychlosti v daném čase asi nebude žádoucí zobrazovat. Konkrétní chování těchto prvků pak může být například následující: součet hodnot daného sloupce nebude dávat smysl zobrazovat, tudíž bude vlastnost *Sum* vracet *null* a vlastnost *CanBeSummed* na základě toho bude indikovat *false*.

Hodnoty některých sloupců mohou být pouze pro čtení, to indikuje vlastnost **IsReadOnly**. Pokud se i přes to pokusíme do sloupce hodnotu zapsat, je vyhozena výjimka.

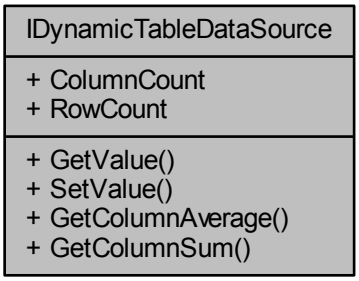
Sloupec může mít zadané jednotky svých hodnot – vlastnost typu textového řetězce **ValueUnits**.

Hodnoty daného sloupce mají vždy určeny datový typ. Ten je uložen ve vlastnosti **ValueType**. Všechny datové typy v jazyce C# dědí od třídy *Object*. Právě s ním pracují metody *GetValue* a *SetValue*. *GetValue* vrací hodnoty přetypované na *Object* a metoda *SetValue* přijímá zadanou hodnotu jako parametr typu *Object*. Pokud se skutečný datový typ zadaný metodě *SetValue* neshoduje s datovým typem sloupce, je vyhozena výjimka.

3.1.2.2 Další mechanismy datových sloupců

Protože některé pohledy v závislosti na jejich aktuálním nastavení mohou dočasně znemožnit zápis do svých sloupců, je zde možnost sloupce uzamknout. Pro tyto účely mají sloupce metody **SetReadOnlyLockOn()** a **SetReadOnlyLockOff()**. Tímto zámkem lze z read-write sloupce udělat dočasně read-only sloupec, nicméně ze sloupce, který je implicitně read-only, read-write odemknutím udělat nelze.

Kromě kontroly datového typu je potřeba u některých sloupců provést další validaci zadané hodnoty. Pro tyto účely má sloupec vlastnost **ValidateDelegate**, která poskytuje validační funkci, která zjistí, jestli je zadaná hodnota validní a případně poskytne chybovou zprávu s popisem problému, pokud hodnota validní není. Validační funkce může například kontrolovat, že do sloupce reprezentujícího cenu není zadána záporná hodnota nebo zda je zadaná hodnota ve stanoveném rozsahu.



Obrázek 3.5: Diagram třídy IDynamicTableDataSource

3.1.2.3 Indexace datových pohledů

Tabulky dat mohou být indexovány celočíselnými indexy nebo datem. Je-li tabulka indexována, pak obsahuje indexační sloupec. Mechanismus reprezentace indexačního sloupce a přístupu k jeho hodnotám je podobný tomu, jak je to u datových sloupců. Hodnoty buňky indexačního sloupce lze získat metodou **GetIndexValue(rowIndex)**. Nastavovat hodnoty indexačního sloupce nelze. Indexační sloupec je reprezentován vlastností **IndexColumn**, která obsahuje instanci třídy **IndexDynamicViewColumn**, nebo hodnotou *null* v případě, že tabulka není indexovaná. To, že tabulka není indexovaná, také indikuje vlastnost **IsIndexed** na základě toho, že datový pohled nemá indexační sloupec.

Třída indexačního sloupce poskytuje jen podmnožinu funkcionality oproti datovým sloupcům. Tato podmnožina je definována v jejich společné nadtřídě – **DynamicViewColumn**. Jedná se o název sloupce – vlastnost **Name**, jeho jednotky – vlastnost **ValueUnits**, datový typ hodnot – vlastnost **ValueType** a metodu pro získání hodnot – **GetValue**.

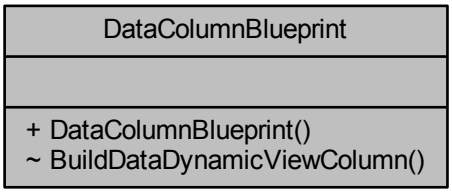
3.1.3 Rozhraní datových struktur aplikace

Součástí modulu je i rozhraní, které musí implementovat datové struktury aplikace, aby k nim mohla být základní tabulka sestavena. Toto rozhraní je realizováno dvěma dílčími rozhraními, rozhraním pro přístup k hodnotám a rozhraním pro přístup k parametrům sloupců.

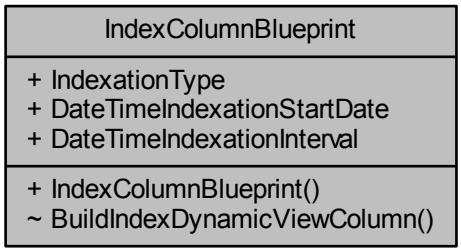
3.1.3.1 Rozhraní pro přístup k hodnotám

Aby mohly být základní tabulce poskytnuty data z datové struktury aplikace, musí tato struktura implementovat rozhraní **IDynamicTableDataSource** (viz obr. 3.5).

Rozhraní poskytuje informace o počtu sloupců a řádků – vlastnosti **ColumnCount** a **RowCount**.



Obrázek 3.6: Diagram třídy DataColumnBlueprint



Obrázek 3.7: Diagram třídy IndexColumnBlueprint

Přístup k hodnotám sloupců a jejím agregacím je pak poskytován skrze metody **GetValue**, **SetValue**. V datových strukturách aplikace jsou data fyzicky uložena, zatímco do datových pohledů se dopropagují až na vyžádání. Datové pohledy tedy data z datových struktur přímo neuchovávají.

Pro přístup k agregátním hodnotám slouží pak metody **GetColumnAverage** a **GetColumnSum** také obdobně k datovým pohledům. Prvotně nebyl důvod vyžadovat v tomto rozhraní součty a průměry hodnot, základní tabulka si je mohla spočítat sama. Nicméně vzhledem k tomu, že datové struktury aplikace mají tyto hodnoty napočítány dopředu a používají k tomu specifický výpočet (jak například nakládat se neobvyklými hodnotami číselných datových typů jako *NaN* apod.), jsou z nich tyto hodnoty přejímány, aby výpočet nebyl prováděn na více místech. Dalším důvodem také je, že v datových strukturách jsou tyto hodnoty cachovány, takže je jejich výpočet prováděn jen, když je to nezbytné.

3.1.3.2 Rozhraní pro přístup k parametrům sloupců

Základní tabulce dat je potřeba kromě samotných dat poskytnout i parametry datových sloupců (název, datový typ apod.) a volitelně i indexačního, pokud jej tabulka má mít. Ty jsou poskytnuty pomocí kolekce předpisů pro datové sloupce a volitelného předpisu indexačního sloupce.

Předpis datového sloupce – třída **DataColumnBlueprint** (viz obr. 3.6) se naplní v konstruktoru parametry sloupce a poté pomocí metody **BuildDataDynamicViewColumn(sourceView)** umí při dodání zdrojového pohledu

(v tomto případě základní tabulky dat) sestavit datový sloupec.

Předpis indexačního sloupce – třída **IndexColumnBlueprint** (viz obr. 3.7) je také při konstrukci naplněn parametry sloupce, ale kromě toho se mu také určí kritéria pro vygenerování samotných hodnot indexace. Vygenerování hodnot indexace pak probíhá na požádání při volání metody **GetIndexValue** v základní tabulce dat. Pro sestavení indexačního sloupce slouží, obdobně jako u předpisu datového sloupce, metoda **BuildIndexDynamicViewColumn(sourceView)**.

Indexace je předepsána – vlastností **IndexationType**, která určí, o jaký typ indexace se jedná. V případě indexace kalendářními daty jsou zde pak vlastnosti **DateTimeIndexationStartDate** a **DateTimeIndexationInterval**, určující počáteční datum a časový interval mezi dvěma sousedními hodnotami. Tento typ indexace se používá v aplikaci zatím nejvíce, pokud se však začnou používat i indexace jiného typu, bude tato část, která je „šitá na míru“ právě indexací daty, zřejmě vhodným kandidátem k zobecnění.

3.1.4 Základní tabulka dat

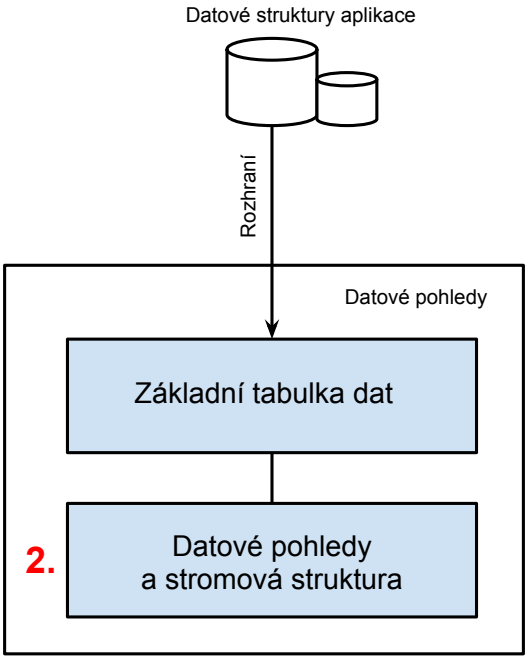
Základní tabulka dat je reprezentována třídou **DynamicTable**. Ta, jak již bylo řečeno, implementuje rozhraní datových pohledů *IDynamicView*.

V konstruktoru přebírá objekt (datovou strukturu) implementující rozhraní pro získání dat – **IDynamicTableDataSource**, předpisy datových sloupců – kolekci **DataColumnBlueprint** a volitelně předpis indexačního sloupce – **IndexColumnBlueprint**. Na základě těchto parametrů se pak inicializuje a poté poskytuje přístup k datům zdrojové datové struktury prostřednictvím rozhraní datových pohledů.

Implementace základní tabulky dat je poměrně přímočará, takže zde není diskutována. Zajímavější je pak to, jakou roli hraje základní tabulka dat v problematice notifikace o změně dat událostmi. Tomu je věnována následující podkapitola o datových pohledech.

3.2 Datové pohledy a jejich uspořádání do stromové struktury

V této části (viz obr. 3.8) je popsáno, jak jsou prováděny operace nad daty pomocí datových pohledů, jak jsou tyto datové pohledy provázány a jak spolupracují. Jsou zde diskutovány požadavky na konkrétní operace s daty, a jak je datové pohledy řeší. Závěr této části je pak věnován notifikaci změn v datových pohledech pomocí událostí.



Obrázek 3.8: Diagram znázorňující rozčlenění modulu – část 2, datové pohledy a stromová struktura

3.2.1 Mechanismy datových pohledů a jejich spolupráce

3.2.1.1 Datové pohledy a operace nad daty

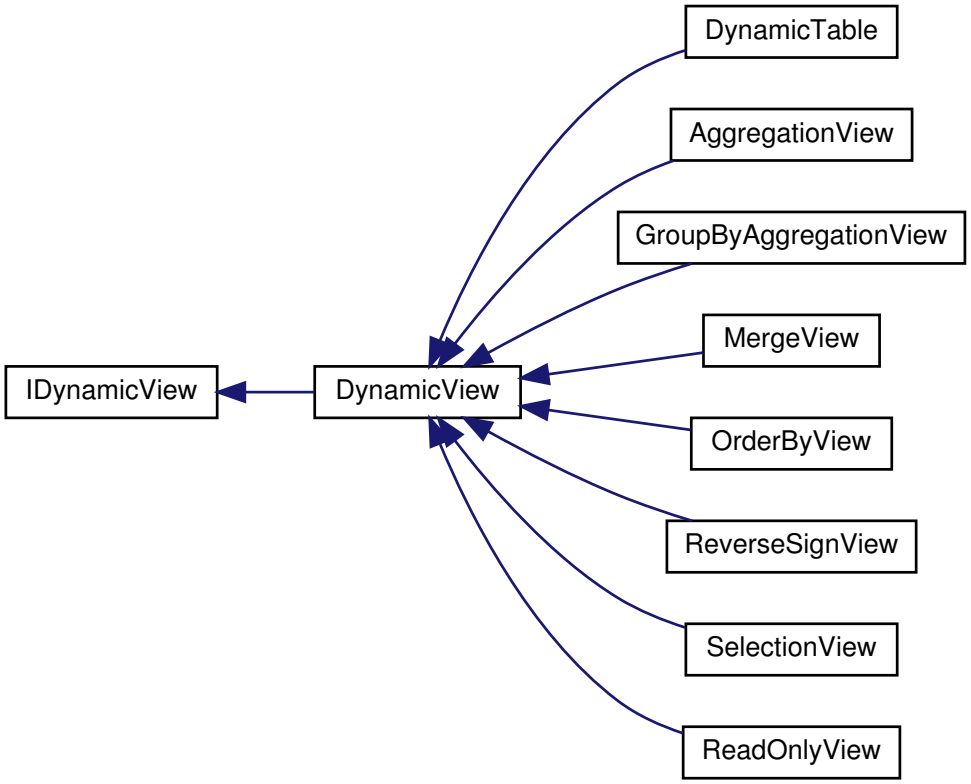
Datové pohledy jsou navrženy tak, aby každý z nich uměl provést nad daty jednu, pokud možno co nejelementárnější operaci (viz odvozené třídy datových pohledů na obr. 3.9). Operace budou podrobně diskutovány v sekci 3.2.2.

Všechny datové pohledy implementují stejné rozhraní. Vzhledem k tomu, že část implementace tohoto rozhraní je pro většinu pohledů stejná, byla jako mezičlánek v jejich hierarchii založena abstraktní třída **DynamicView**, která jednotně implementuje některé prvky tohoto rozhraní. Všechny pohledy tedy dědí až od této abstraktní třídy a ne přímo od rozhraní *IDynamicView* (viz obr. 3.9).

3.2.1.2 Uspořádání pohledů do stromové struktury

Každý pohled, kromě základní tabulky dat, přijímá v konstruktoru referenci na zdrojový – rodičovský pohled.

Výjimku tvoří pohled pro sloučení tabulek – *MergeView*, který těchto zdrojových pohledů může přijmout více. Nebýt této výjimky, kde tento pohled může mít více rodičů, tvořily by pohledy strukturu přímo stromovou. Ačkoli se tedy o přímo stromovou strukturu nejedná, dále bude tak pro zjednodušení



Obrázek 3.9: Diagram dědičnosti datových pohledů

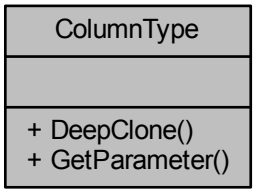
označovaná. Po zavedení některých důležitých mechanismů datových pohledů následuje příklad této stromové struktury (viz sekce 3.2.2.4 a obr. 3.15).

Nad daty tohoto zdrojového pohledu potom pohled provádí svou datovou operaci. Referenci na zdrojový pohled obdrží pod jednotným rozhraním pohledů – *IDynamicView*, a tím pádem je schopen být navázán na jakýkoli jiný pohled, aniž by potřeboval rozpoznat, jestli pracuje například nad základní tabulkou dat nebo na již modifikovaných tabulkách, které poskytují další pohledy.

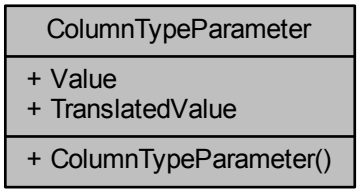
V rozhraní pohledů je pro účely orientace ve stromové struktuře vlastnost **Parents**, která uchovává reference na rodičovské pohledy.

Je-li tedy na tuto strukturu nahlíženo jako na stromovou, základní tabulky dat figurují jako kořeny těchto stromů. Postupně se tedy od kořenů stromů skrz jednotlivé uzly provádí dílčí operace nad zpracovanými daty a lze je téměř libovolně řetězit.

Jednotlivé datové pohledy a jejich operace jsou popsány dále v sekci 3.2.2.



Obrázek 3.10: Diagram třídy ColumnType



Obrázek 3.11: Diagram třídy ColumnTypeParameter

3.2.1.3 Identifikace a výběr sloupců

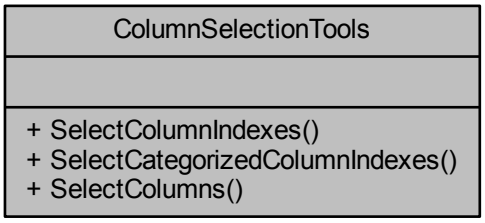
Protože je pro některé pohledy nutné specifikovat, nad jakými sloupci mají danou operaci provést, má každý sloupec sadu parametrů, které slouží k jeho identifikaci – **ColumnType** (viz obr. 3.10). Tuto sadu obsahuje každý datový sloupec ve stejnojmenné vlastnosti **ColumnType**.

Identifikační parametr je reprezentován třídou **ColumnTypeParameter** (viz obr. 3.11). Parametr je určen jeho názvem a hodnotou. Název je uložen jako klíč ve slovníku ve třídě *ColumnType* (tento slovník není součástí veřejného rozhraní třídy), kde jsou parametry uloženy. Hodnota parametru je ve vlastnosti **Value**. Pro získání parametru dle jeho názvu slouží metoda **GetParameter(parameterName)**. Mezi typické identifikační parametry sloupce patří název třídy zdrojové datové struktury (*ClassName*), identifikátor instance datové struktury (*ClassInstanceId*) apod.

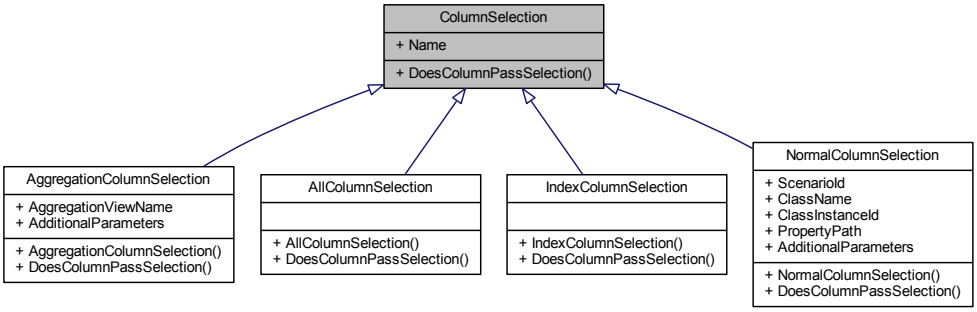
Vlastnost **TranslatedValue** je diskutována až později, v souvislosti s překladem názvů sloupců (viz sekce 3.2.3).

Aby bylo možné sloupec na základě této sady parametrů vybrat, byly zavedeny tzv. selekce sloupců – třída **ColumnSelection**. Selekce obsahují kritéria na identifikační parametry sloupců, která jsou tvořena názvem parametru a jeho očekávanou hodnotou. Jestli sloupec kritérii selekce prošel a je tedy vybrán, lze zjistit skrze metodu **DoesColumnPassSelection(dataColumn)**. Tato metoda ověří přesnou shodu skutečné hodnoty parametru s jeho očekávanou hodnotou danou kritériem.

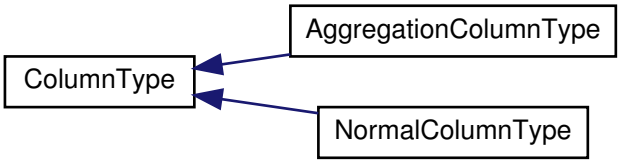
Další pomocné metody pro výběr sloupců pomocí jejich selekcí jsou umístěny ve statické třídě **ColumnSelectionTools** (viz obr. 3.12). Lze zde např. pomocí metody **SelectColumnIndexes(sourceView, columnSelections)**



Obrázek 3.12: Diagram třídy ColumnSelectionTools



Obrázek 3.13: Hierarchický diagram tříd ColumnSelection



Obrázek 3.14: Diagram dědičnosti tříd ColumnType

získat indexy sloupců, které byly vybrány zadanými selekcemi apod.

Třídy identifikačních parametrů – *ColumnType* a selekcí – *ColumnSelection* jsou abstraktní a zajišťují pouze sdílené rozhraní, se kterým se následně polymorfně pracuje. Čili libovolná selekce lze uplatnit na libovolnou sadu identifikačních parametrů. Konkrétní třídy, které jsou znázorněny v následujících diagramech, pak tyto abstraktní třídy implementují (viz obr. 3.13 a 3.14).

Nejběžnějšími představiteli jsou **NormalColumnType** pro sadu parametrů a **NormalColumnSelection** pro selekci sloupců. Všechny sloupce ze základní tabulky dat mají právě tuto standardní sadu parametrů, která obsahuje pevně dané parametry jako název třídy zdrojové datové struktury, id instance téže struktury, apod. Standardní selekce (*NormalColumnSelection*) potom má připravena kritéria (v podobě jednotlivých vlastností) k této sadě sadu parametrů.

Ostatní implementace sad parametrů a selekcí jsou diskutovány v rozboru

jednotlivých operací s daty v následující sekci 3.2.2.

Datové pohledy, které potřebují určit, nad jakými sloupci mají pracovat, přijímají v konstruktoru posloupnost těchto selekcí sloupců. Aby bylo zpětně dohledatelné, jaké selekce byly použity, má rozhraní datových pohledů vlastnost **UsedColumnSelections**, kde jsou aplikované selekce uloženy.

3.2.2 Operace s daty

3.2.2.1 Výběr podmnožiny sloupců – pohled SelectionView

První požadovaná operace s daty je výběr podmnožiny sloupců. Základní tabulka dat může mít totiž desítky sloupců a zobrazovat je všechny by bylo nepřehledné. Proto je potřeba mít možnost vybrat jaké sloupce zobrazovat a definovat jejich pořadí. To realizuje datový pohled pro výběr sloupců – **SelectionView**.

Ten přijímá v konstruktoru zdrojový pohled a posloupnost selekcí sloupců. Ze sloupců zdrojového pohledu potom vybere sloupce na základě poskytnutých selekcí. Tento výběr si uloží jako zobrazení indexů svých sloupců na sloupce zdrojové tabulky. Pak při implementaci metod pro přístup k datům *GetDataValue* a *SetDataValue* toto zobrazení použije a přístup přesměruje na daný sloupec zdrojové tabulky.

Sám tedy neobsahuje žádná data, ale pouze zprostředkovává podmnožinu dat zdrojového pohledu. Vytváří si však svoje vlastní instance datových sloupců, které přebírají parametry odpovídajících sloupců ze zdrojového pohledu.

3.2.2.2 Tvorba agregačních sloupců – pohled AggregationView

Další požadovanou operací je tvorba agregačních sloupců, které provádí nad několika sloupci danou aritmetickou operaci. Hodnoty agregačního sloupce jsou tedy výsledky dané aritmetické operace nad hodnotami vstupních sloupců. Tuto operaci realizuje agregační datový pohled – **AggregationView**.

Tento pohled přijímá v konstruktoru zdrojový pohled a typ aritmetické operace. Agregace je provedena nad všemi sloupci zdrojového pohledu. Typ aritmetické operace je dán výčtovým typem **EAggregationOperation**. Implementované aritmetické operace jsou: *součet*, *rozdíl*, *součin*, *podíl*, *minimum*, *maximum* a *průměr*. Případně není problém doimplementovat další potřebné operace.

Na vstupní sloupce je kladen požadavek, aby byly číselného datového typu. Konkrétně je zatím implementována podpora pouze pro datový typ *double*. Dále musí všechny sloupce být stejného datového typu a o stejných jednotkách. V opačném případě by agregace nedávala smysl a je vyhozena výjimka.

Následně pohled vytvoří jeden agregační sloupec, který „obsahuje“ agregované hodnoty. Agregované hodnoty jsou vypočteny použitím příslušné aritmetické operace až na dotázání metody pro získání hodnoty *GetDataValue*.

Dále je potřeba provést vlastní implementaci výpočtu součtů a průměrů jednotlivých sloupců – metody *GetDataColumnSum* a *GetDataColumAverage*. Nelze zde totiž obecně vzít hodnoty součtu jednotlivých sloupců a udělat nad nimi danou aritmetickou operace a výsledek považovat za součet jednotlivých hodnot agregačního sloupce. Například pro aritmetickou operaci minimum totiž neplatí, že když vezmu součty jednotlivých sloupců a vypočtu nad nimi minimum, že by výsledek byl roven součtu minim jednotlivých řádků jednotlivých sloupců.

Následující rovnost tedy obecně **neplatí**:

Nechť n je počet řádků, m je počet sloupců, $s_{m,n}$ je hodnota n-tého řádku m-tého sloupce:

$\min((s_{1,1} + s_{1,2} + \ldots + s_{1,n}), (s_{2,1} + s_{2,2} + \ldots + s_{2,n}), \ldots, (s_{m,1} + s_{m,2} + \ldots + s_{m,n})) = \min(s_{1,1}, s_{2,1}, \ldots, s_{m,1}) + \min(s_{1,2}, s_{2,2}, \ldots, s_{m,2}) + \ldots + \min(s_{1,n}, s_{2,n}, \ldots, s_{m,n}))$

To, že tato rovnost obecně neplatí lze ukázat na následujícím příkladu:

Nechť má první sloupec hodnoty (1, 0) a druhý sloupec hodnoty (0, 1). Po dosazení do rovnosti vyjde na levé straně: $\min(1, 1) = 1$, a na pravé straně: $0 + 0 = 0$. Rovnost tedy neplatí.

Agregační sloupec je z principu pouze pro čtení – jeho vlastnost **IsReadOnly** tedy vrací *true*. Datový typ a jednotky přebírá od zdrojových sloupců (jejichž jednotnost je zajištěna zmíněnou podmínkou).

Tento sloupec si vytváří vlastní identifikační sadu parametrů, konkrétně instanci třídy **AggregationColumnType**. Ta již neobsahuje standardní parametry jako zmíněný *NormalColumnType*, ale pouze název agregačního pohledu. Agregační pohled poskytuje vždy jen jeden sloupec a tím pádem může být tento sloupec jednoznačně identifikován názvem tohoto pohledu (pokud by tedy nebylo vytvořeno více datových pohledů se stejným názvem). Jako selekce pro tento typ sloupců pak slouží **AggregationColumnSelection**, která má připraveno kritérium právě pro název agregačního pohledu.

Pokud by bylo potřeba vytvořit agregační sloupec jako výsledek složeného aritmetického výrazu nad jinými sloupci, lze toho docílit vhodně zvolenými a poskládanými agregačními pohledy a pohledy pro výběr sloupců.

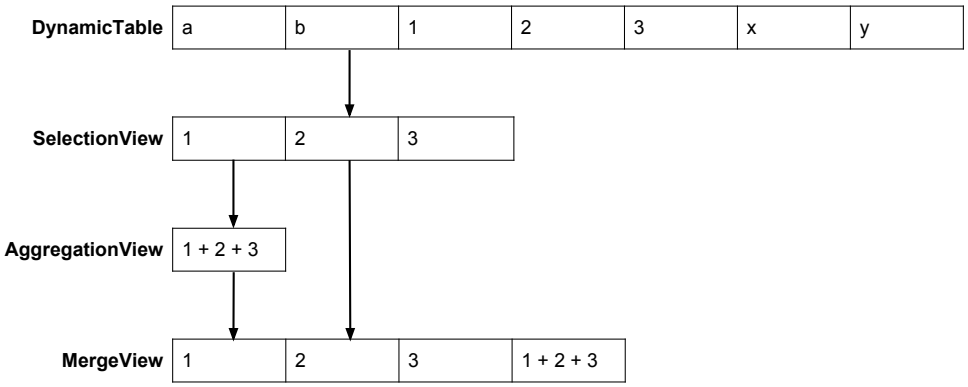
3.2.2.3 Slučování tabulek – pohled MergeView

Další operací, kterou je potřeba zajistit, je slučování tabulek – několik vstupních tabulek sloučit do jedné.

V souvislosti s tvorbou agregačních sloupců pomocí pohledu *AggregationView*, kdy vznikne z tabulky jen jeden sloupec, je pak slučování tabulek nezbytné, neboť agregační sloupce jsou zpravidla zobrazovány spolu s jinými sloupci a nikoli jako tabulka o jednom sloupci.

Tuto operaci zajišťuje datový pohled pro sloučení tabulek – **MergeView**.

Tento pohled přijímá v konstruktoru posloupnost zdrojových pohledů, které mají být sloučeny.



Obrázek 3.15: Příklad struktury datových pohledů SelectionView, AggregationView a MergeView

Na tyto pohledy je kladena podmínka, že musí mít všechny stejný počet řádků a stejný způsob indexace. V opačném případě je vyhozena výjimka.

Dále pohled vytvoří odpovídající datové sloupce zkopírováním jednotlivých sloupců (resp. jejich parametrů) vstupních datových pohledů (podobně jako u pohledu pro výběr sloupců).

Metody pro přístup k hodnotám *GetDataValue* a *SetDataValue*, jsou pak implementovány tak, aby ze zadaného indexu sloupce *MergeView* našly odpovídající sloupec v jednom ze zdrojových pohledů a poskytnou přístup k jeho hodnotám.

3.2.2.4 Příklad struktury pohledů

Pro lepší představu, předvedu typický příklad způsobu využití zmíněných tří typů pohledů ve stromové struktuře (viz obr. 3.15).

Cílem je připravit tabulku kde budou pouze sloupce 1, 2, 3 a jejich součet.

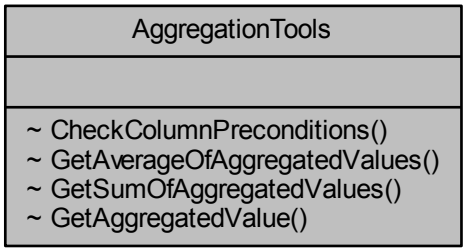
Ve zdrojové tabulce je více sloupců, které ale nejsou žádoucí. Pomocí pohledu pro výběr sloupců tedy vyberu jenom sloupce 1, 2, 3.

Dále pomocí agregačního pohledu připravím sloupec se součtem těchto sloupců.

Na závěr pomocí pohledu pro slučování tabulek sloučím tyto dva pohledy v požadovaném pořadí a tím získávám požadovanou tabulku.

3.2.2.5 Seskupování sloupců – pohled GroupByAggregationView

Seskupování sloupců si lze představit podobně jako funkcionalitu klauzule *GroupBy* databázového jazyka *SQL* [3]. Jde o to, vybrané sloupce rozdělit do skupin podle daného identifikačního parametru a tyto skupiny reprezentovat jedním sloupcem, který nad danou skupinou sloupců provádí agregaci. Seskupování zajišťuje skupinový agregační pohled – **GroupByAggregati-**



Obrázek 3.16: Diagram třídy AggregationTools

onView. Příklad tohoto pohledu ve stromové struktuře následuje dále, po zavedení pohledu pro řazení sloupců, v sekci 3.2.2.7 a na obr. 3.17.

GroupByAggregationView přijímá v konstruktoru zdrojový pohled, typ agregační operace (daný výčtovým typem *EAggregationOperation*, jako je tomu u normálního agregačního pohledu) a název identifikačního parametru, dle kterého proběhne seskupení.

Následně se připraví zobrazení skupin sloupců – každá výsledná skupina, reprezentovaná jedním výsledným agregačním sloupcem, se zobrazuje na kolekci svých zdrojových sloupců.

Nad jednotlivými skupinami sloupců, které budou agregovány, proběhne stejná kontrola vstupních podmínek jako u normálního agregačního pohledu. Dále proběhne kontrola, že každý sloupec má definovaný identifikační parametr, dle kterého probíhá seskupení. V opačném případě je vyhozena výjimka.

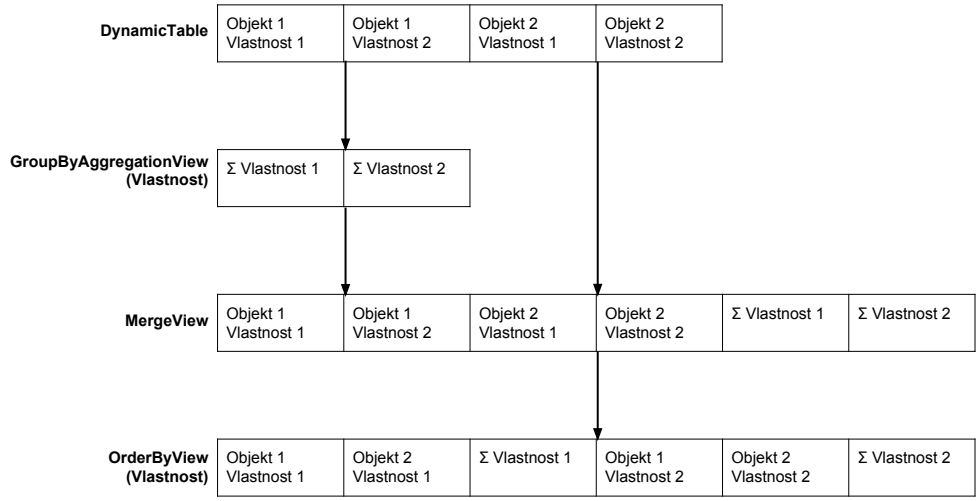
Přístup k hodnotám *GetDataValue*, *SetDataValue*, *GetDataColumnSum*, *GetDataColumnAverage*, je řešen obdobně jako u normálního agregačního pohledu, je však respektováno rozdělení do skupin pomocí zmíněného zobrazení skupin sloupců.

Vzhledem k tomu, že kontroly vstupních podmínek i výpočet hodnot agregačních sloupců jsou v obou agregačních pohledech obdobné, jsou tyto agregační nástroje sdruženy ve statické třídě **AggregationTools** (viz obr. 3.16), kterou používají oba pohledy.

Vzniklé datové sloupce potom mají agregační identifikační parametry – **AggregationColumnType** podobně jako v normálním agregačním pohledu, kde je ale navíc parametr, podle kterého proběhlo seskupení, aby bylo možné jednotlivé sloupce od sebe odlišit.

3.2.2.6 Řazení sloupců – pohled OrderByView

Operace řazení sloupců potom umožňuje seřadit sloupce tabulky dle hodnoty určitého identifikačního parametru. Respektive, je zde možnost řadit sloupce postupně podle n kritérií – n identifikačních parametrů. Operaci řazení zajišťuje pohled pro řazení – **OrderByView**.



Obrázek 3.17: Příklad struktury datových pohledů GroupByAggregationView a OrderByView

Pohled pro řazení obdrží v konstruktoru zdrojový pohled a posloupnost řadících kritérií – identifikačních parametrů. Dále si pohled připraví seřazené pořadí sloupců reprezentované jako zobrazení výsledných (seřazených) sloupců na zdrojové (neseřazené) – podobně jako v pohledu pro výběr sloupců. Řazení probíhá postupně dle všech zadaných kritérií.

Řazení je stabilní, tj. vzájemné pořadí prvků se stejnou hodnotou parametru, dle kterého probíhá řazení, zůstává neměnné.

Při implementaci metod pro přístup k datům *GetDataValue* a *SetDataValue*, přesměruje pohled zadané indexy sloupců na indexy sloupců zdrojového pohledu s využitím zmíněného zobrazení (podobně jako u pohledu pro výběr sloupců).

3.2.2.7 Příklad struktury pohledů s využitím seskupení a řazení

Pro lepší představu uvedu příklad struktury pohledů, kde se využívá seskupení a řazení (viz obr. 3.17).

Cílem je získat tabulku seřazenou dle vlastností, kde u každé vlastnosti bude i její součet. Předpokládejme, že počet ani název vlastností není dopředu znám, tudíž je nelze postupně vybrat.

Nejdříve pomocí skupinového agregačního pohledu vytvořím součty jednotlivých vlastností. Tabulku součtů potom sloučím s původní tabulkou.

Nyní mám všechny sloupce, které potřebuji, pouze v nesprávném pořadí. Pomocí řadícího pohledu je tedy seřadím a získám požadovanou tabulku.

3.2.2.8 Pomocné operace s daty

Kromě těchto základních operací nad daty bylo potřeba implementovat ještě některé pomocné operace.

První z nich je převod celé tabulky, aby byla pouze pro čtení. K tomu slouží datový pohled **ReadOnlyView**.

Další požadovanou funkcí je otočení znamének vybraných číselných sloupců. Toho se pak využívá především při zobrazení některých sloupců v grafu. Například při zobrazování nákladů a výnosů v grafu by bylo vhodné sloupec nákladů znázornit se záporným znaménkem kvůli přehlednosti. Tuto funkci zajišťuje datový pohled pro otočení znaménka – **ReverseSignView**.

Ačkoli většina selekcí sloupců bývá realizována třídami standardních selekcí (*NormalColumnSelection*) a agregačních selekcí (*AggregationColumnSelection*), jsou zde ještě naimplementovány selekce, které vybírají sloupec dle jeho indexu – **IndexColumnSelection** a selekci, která vybere všechny sloupce – **AllColumnSelection**. Například selekce dle indexu sloupce je využívána při testování, když není žádoucí řešit, jaké identifikační parametry má testovací sada dat apod.

3.2.3 Generování názvu sloupce s využitím placeholderů

Další problematikou je generování názvů sloupců. Konkrétně je potřeba vyřešit dva problémy.

Prvním problémem je, že pokud jsou například zobrazeny v tabulce vlastnosti k jedné entitě, název sloupců by bylo vhodné popsat jenom názvem vlastnosti – např. „Cena“. Pokud však bude existovat tabulka obsahující více entit a jejich vlastnosti, bylo by již potřeba, aby názvy sloupců byly tvořeny jak názvem vlastnosti, tak názvem entity, aby byly názvy jednoznačné – např. „Objekt 1 – Cena“.

Druhým problémem je pojmenování agregačních sloupců. Normální agregační pohled vytvoří vždy jeden sloupec, tam stačí určit název tohoto sloupce v konstruktoru pohledu. Problém však nastává u skupinového agregačního pohledu, kde vznikne těchto agregačních sloupců několik a dopředu není známo kolik, ale přesto je třeba názvy těmto sloupcům určit.

Jako řešení těchto dvou problémů byl použit předpis názvů sloupců, ve kterém můžou figurovat tzv. placeholdery. Ty obsahují název identifikačního parametru, který je před zobrazením nahrazen uživatelsky čitelnou podobou hodnoty tohoto parametru.

Placeholdery jsou speciální podřetězce, které jsou součástí řetězce s předpisem názvu. Ty jsou následně přeloženy – nahrazeny překladem. Příkladem předpisu názvu sloupce s použitím placeholderu je: „Sloupec třídy {className}“, překlad hodnoty parametru „className“ pak bude například: „Moje třída“. Po přeložení by tedy sloupec měl název „Sloupec třídy Moje třída“.

První problém tedy lze vyřešit tím, že by předpis názvů sloupců pro tabulku vlastnosti jedné entity mohl mít předpis „{propertyName}“. Zatímco předpis názvu pro tabulku obsahující více entit by mohl vypadat takto – „{instanceName} – {propertyName}“. Kratší název sloupce by tedy například byl „Cena“ a delší název pak „Objekt 1 – Cena“.

Definování názvů dopředu neznámého počtu agregačních sloupců má tím pádem také řešení. Agregačním sloupcům lze předat předpis jejich názvu, ve kterém by figuroval placeholder parametru, podle kterého probíhá seskupení – např. „Součet vlastnosti {propertyName}“. Jednotlivé agregační sloupce by pak měly po překladu názvy jako: „Součet vlastnosti A“, „Součet vlastnosti B“ apod.

Implementace tohoto řešení je následující:

- Každý identifikační parametr – **ColumnTypeParameter** má kromě své hodnoty i přeloženou, uživatelsky čitelnou hodnotu – vlastnost **TranslatedValue** (tou je nahrazen placeholder parametru při překladu).
- Datové sloupce – **DataDynamicViewColumn** mají stanovený předpis názvu – vlastnost **NameBlueprint**. A název, který se bude výsledně zobrazovat uživateli, poskytuje vlastnost **Name**, která až při zavolání předpis přeloží.

3.2.4 Notifikace o změnách v datových pohledech

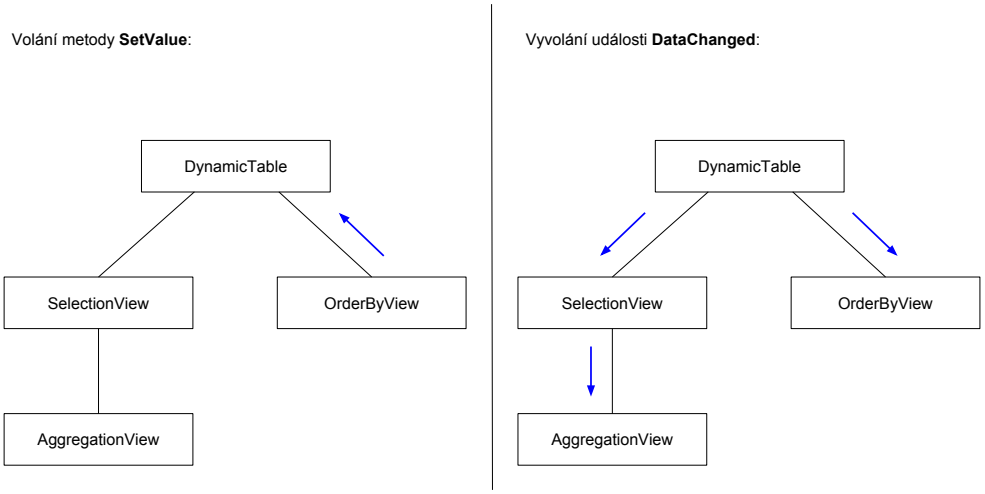
V datových pohledech mohou nastat změny buď v tabulkových datech, nebo změny parametrů pohledů.

Pro notifikaci změn parametrů daného pohledu slouží událost **ParametersChanged**. Nakonec však na stávajících pohledech nemá tato notifikace využití. Je to vzhledem k tomu, že se pohledy vždy zkonstruují, parametry se nastaví a dál už fungují zpravidla beze změn v těchto parametrech, čili není co notifikovat.

3.2.4.1 Notifikace pohledů ve stromové struktuře

Změny v tabulkových datech jsou však důležité. V datech mohou z principu nastat změny (u datových pohledů skrze metodu *SetDataValue*) a je potřeba, aby se o nich zobrazovací komponenty aplikace dozvěděly. Zobrazovací komponenty čerpají data z datových pohledů (což je diskutováno dále v části o schématech zobrazovacích komponent). Je tedy potřeba zajistit, aby datové pohledy umožňovaly na tyto změny upozornit. Tím získají zobrazovací komponenty podnět si zobrazená data znovu načíst.

Jak již bylo řečeno, datové pohledy data neuchovávají, jejich úložiště je ve zdrojových datových strukturách nad kořenovými základními tabulkami dat. Když se budeme rekurzivně pohybovat z jakéhokoli pohledu k jeho rodičům, vždy dojdeme ke kořenové tabulce.



Obrázek 3.18: Šíření události DataChanged ve stromu datových pohledů

Řešením notifikace tedy je, aby se vždy o změně dozvěděla kořenová tabulka, která data poskytuje a vyslala tuto zprávu dál navěšeným pohledů. Ty by jí pak rekurzivně přeposílaly svým navěšeným pohledům a tímto způsobem by se každý pohled, který ze zdrojové tabulky data čerpal, o změně dozvěděl.

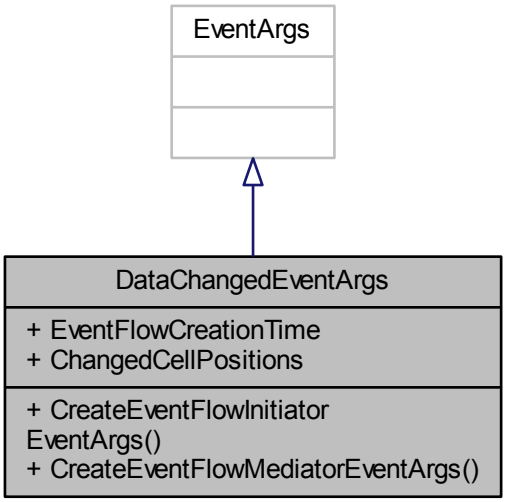
Volání metody *SetDataValue*, kterou lze hodnoty v pohledech měnit, se pouze propaguje směrem ke kořenové tabulce, a až tam volání této metody něco s daty udělá – změní je ve zdroji dat. Tím pádem máme zajištěno, že se každá změna dat dostane ke kořenové tabulce.

Aby vůbec pohledy mohly notifikovat změny dat, obsahuje jejich rozhraní událost **DataChanged**.

Dále je zajištěno, aby každý pohled naslouchal na tuto událost svému rodiči (rodičům v případě *MergeView*) a jakmile událost u rodiče nastane, vyvolal jí u sebe. Tím je zajištěno šíření událostí od kořenové tabulky všem pohledům dané struktury. To, aby pohledy naslouchaly na událost svým rodičům, zajišťuje zavolání metody **AttachDataChangedToParents** abstraktní třídy *DynamicView* při inicializaci každého pohledu. Tím je zajištěno, že se o změně dat každý datový pohled ve stromu pod danou kořenovou tabulkou dat dozví (viz obr. 3.18).

3.2.4.2 Specifikace oblasti změny

Aby se při každé menší změně dat nemusely zobrazovací komponenty překreslovat celé, je potřeba zajistit, aby se spolu s událostí o tom, že se něco změnilo, komponenty dozvěděly i o tom, jaké konkrétní buňky byly změněny. Pak mohou překreslit jenom změněné buňky. V případě, že se změní buňky, které aktuálně nejsou zobrazeny, nemusí komponenty reagovat vůbec.



Obrázek 3.19: Diagram třídy DataChangedEventArgs

Spolu s událostí *DataChanged* se tedy posílají i parametry události, kde jsou změněné buňky vyznačeny. Tyto parametry jsou reprezentovány třídou **DataChangedEventArgs** (viz obr. 3.19).

Tyto parametry poskytují souřadnice změněných buněk prostřednictvím vlastnosti **ChangedCellPositions**. Souřadnice se vztahují vždy k aktuálnímu pohledu, odkud byla událost vyvolána. Je tedy potřeba, aby jak se událost propaguje od kořenové tabulky, si ji každý pohled zpracoval a souřadnice změněných buněk příslušně upravil. Případně pokud dané buňky jsou ve sloupcích, které pohledem nejsou přebírány, tak událost může ignorovat a dál ji neposílat.

Každý datový pohled proto musí implementovat abstraktní metodu **OnSourceDataChanged(sender, args)** abstraktního předka pohledů – *DynamicView*. V této metodě se událost od rodiče zpracuje a s opravenými souřadnicemi změněných buněk se případně událost opět vyvolá z aktuálního pohledu. Tato metoda je zpracovatelem události o změně dat (*DataChanged*) z rodičovských pohledů (což zajišťuje již zmíněná metoda *AttachDataChangedToParents*).

Spíše pro testovací účely je zde možnost rozlišit události na základě času, kdy byl celý tok události stromem vytvořen – vlastnost **EventFlowCreationTime**. Při tvorbě parametrů události je tedy potřeba rozlišovat, jestli se jedná o událost, která přímo iniciuje tok události – to jsou události vyslané z kořenové tabulky, či pouze událost přeposílá – události z ostatních datových pohledů. Proto lze parametry události vytvořit pomocí dvou statických metod – **CreateEventFlowInitiatorEventArgs(changedCellPositions)** pro kořenovou tabulku a **CreateEventFlowMediatorEventArgs(changedCellPositions, eventFlowCreationTime)** pro ostatní pohledy.

3.2.4.3 Pozastavení notifikace

Při změně celého bloku dat tabulky naráz, pomocí volání změn jednotlivých buněk (metodou *SetDataValue*), by byla událost o změně vyvolána mnohokrát a aplikaci by to mohlo zpomalit. Proto je naimplementována funkcionalita pro pozastavení notifikace událostmi a její následné znovuspuštění.

Starají se o to metody datových pohledů **SuspendDataChangedEvents()** a **ResumeDataChangedEvents()**. Volání těchto metod se vždy propaguje směrem ke kořenové tabulce. Při zavolání pozastavení na ní se pak v kořenové tabulce notifikace událostmi pozastaví. Jakmile dojde ke změně dat při jejich pozastavení, tak namísto vyslání události, se informace o události (její parametry) začnou skladovat na úložiště. Jakmile je pak zavoláno znovuspuštění událostí, tak pokud úložiště není prázdné, vyšle se událost, která obsahuje informace o všech změnách, které nastaly při pozastavení událostí, najednou.

3.2.4.4 Příliš častá nebo duplicitní notifikace

Může se také stát, že notifikace zobrazovacích komponent nastává příliš často, ale kdybychom tak často zobrazená data obnovovali, aplikace by se zbytečně zpomalovala.

To může nastat, pokud by se uživatel rozhodl data zapisovat buňku po buňce v krátkých intervalech. Ještě častější situací, která zatím úmyslně nebyla zmiňována, je, když se ve stromové struktuře vyskytuje pohled pro sloučení tabulek.

Jak je vidět z obrázku 3.20, k *MergeView* může událost o změně dorazit dvakrát (kvůli porušení stromové struktury). Sice po každé bude pohled notifikován o jiném sloupci, ale přesto donutí zobrazovací komponentu obnovit se dvakrát. U složitějších stromových struktur by to pak mohlo být i vícekrát.

Z tohoto důvodu je implementován adapter změn dat – **DataChanged-CooldownAdapter** (viz obr. 3.21). Ten se naváže na datový pohled, odkud čerpá *DataChanged* události. Ve svém veřejném rozhraní pak vlastní událost **DataChanged** poskytuje. Adapter má stanovený interval, ve kterém má události vysílat.

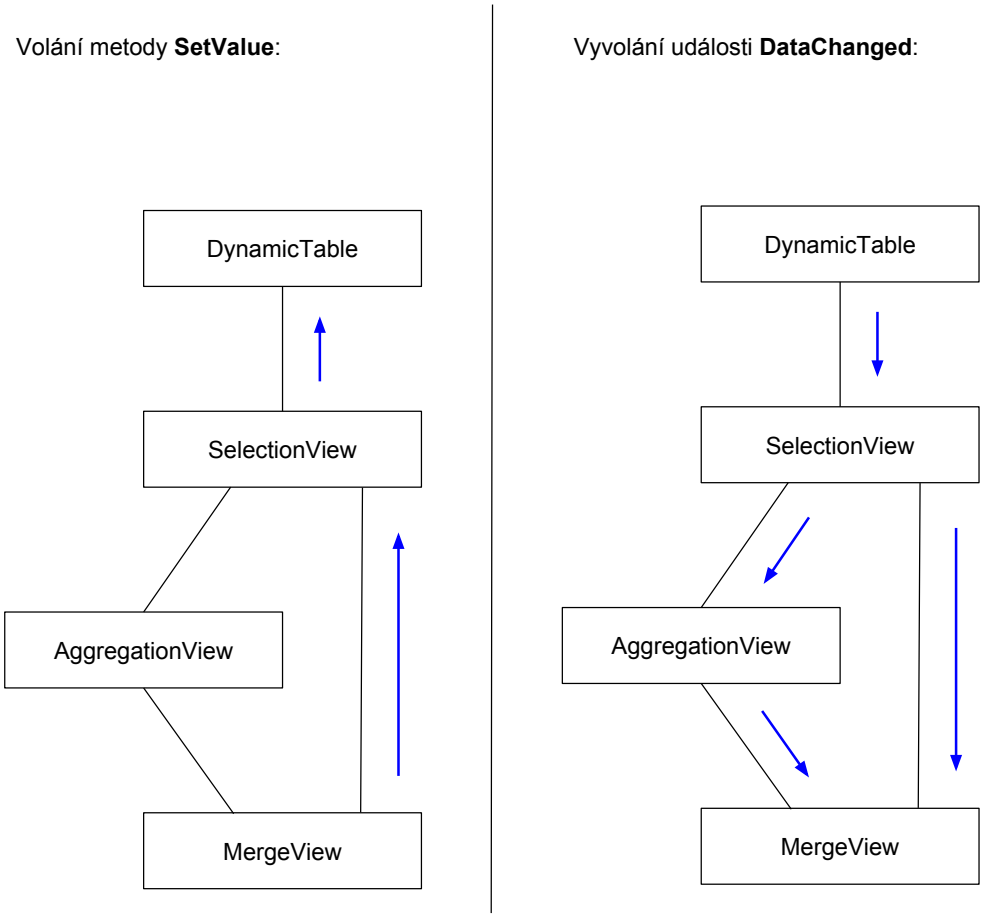
Zobrazovací komponenta se pak nenavazuje přímo na událost *DataChanged* datového pohledu, ale udělá si svůj *DataChangedCooldownAdapter* a naváže se na jeho *DataChanged* událost.

Fungování adapteru je následující:

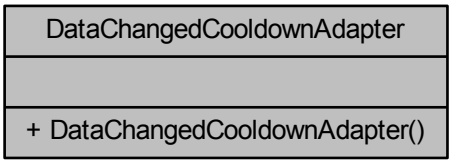
1. Na začátku má adapter vypnuté stopky.

2. Jakmile obdrží událost o změně, uloží si ji a zapne stopky se stanoveným intervalem.

3. Když stopky běží a adapter obdrží další události, uloží si je také.



Obrázek 3.20: Šíření události DataChanged v porušené stromové struktuře



Obrázek 3.21: Diagram třídy DataChangedCooldownAdapter

4. Jakmile stopky doběhnou, uložené události se seskupí do jedné kumulované události (podobně jako u pozastavení notifikace), tu adapter vyšle a stopky vypne.

5. Poté následuje znovu krok 1.

3.3 Schéma pro zobrazovací komponenty a jejich rozhraní

Tato část modulu (viz obr. 3.22) se věnuje struktuře, která zprostředkováná předání dat z datových pohledů pro zobrazovací komponenty graf a datagrid – tzv. schéma pro zobrazovací komponenty. Součástí této části je dále definice rozhraní, skrze která graf a datagrid přijímá svá data a nastavení.

3.3.1 Popis rozhraní pro zobrazovací komponenty

V rozhraní je potřeba komponentám poskytnout data a dále některá jejich nastavení.

Data jsou poskytovány skrze rozhraní datových pohledů – *IDynamicView*, která předávají komponentám data ve formě tabulek hodnot.

Nastavení mají potom obě zobrazovací komponenty individuální. Nastavení grafu zaštiťuje třída **ChartExtraParameters** a nastavení datagridu třída **SDGVExtraParameters**.

3.3.1.1 Rozhraní pro graf

Graf se tedy zkonstruuje na základě již probraného datového pohledu – **IDynamicView** a jeho nastavení – **ChartExtraParameters**.

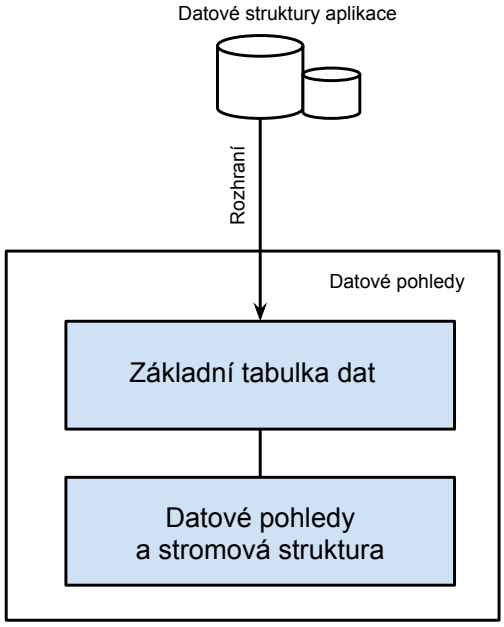
Vzhledem k tomu, že všechny řady (sloupce) hodnot grafu mají společnou osu Y, je na datový pohled kladena podmínka, aby všechny datové sloupce byly o stejné jednotce. V opačném případě je vyhozena výjimka. Tato kontrola však proběhne až při konstrukci grafu, která již sahá mimo rámec implementovaného modulu.

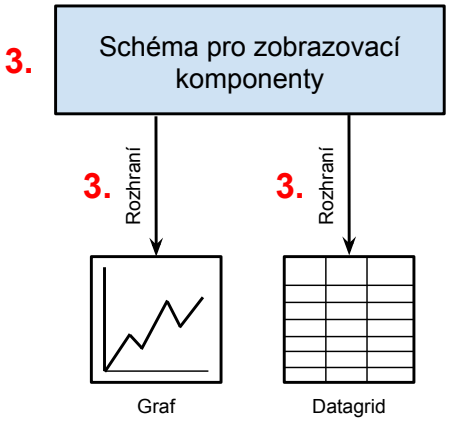
Další podmínkou je, aby tabulka dat měla indexační sloupec, jeho hodnoty pak definují hodnoty na ose X. Tato podmínka je opět kontrolována až při konstrukci grafu.

Následuje rozbor jednotlivých prvků nastavení grafu – *ChartExtraParameters* (viz obr. 3.23).

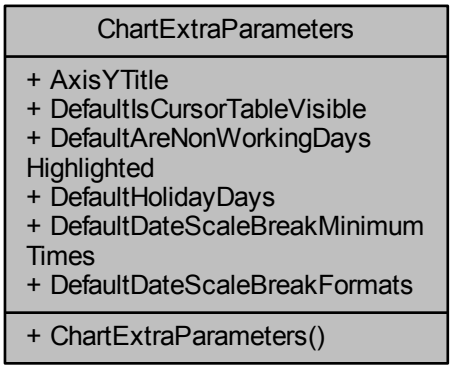
Nastavení grafu obsahuje titulek osy Y – vlastnost **AxisYTitle**, který nelze z tabulky (reprezentované datovým pohledem) získat. Popisek osy X lze naopak převzít z názvu indexačního sloupce.

Další nastavení (vlastnosti s předponou *Default* v diagramu třídy) jsou již více specifická pro aplikaci. Kromě toho nejsou součástí později probíraného

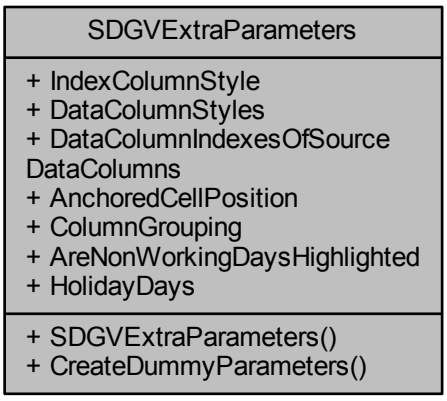




Obrázek 3.22: Diagram znázorňující rozčlenění modulu – část 3, schéma pro zobrazovací komponenty



Obrázek 3.23: Diagram třídy ChartExtraParameters



Obrázek 3.24: Diagram třídy SDGVExtraParameters

předpisu schématu v souboru, ale přebírají se přímo z nastavení aplikace, proto nejsou dále diskutovány.

3.3.1.2 Rozhraní pro datagrid

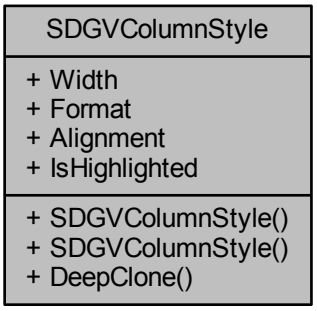
Datagrid se podobně jako graf, zkonstruuje na základě datového pohledu – **IDynamicView** a jeho nastavení – **SDGVExtraParameters**.

Na datový pohled pro datagrid nejsou kladena omezení jako tomu bylo u grafu.

Nastavení datagridu jsou o něco složitější než nastavení pro graf a jeho hlavní rysy jsou v následující části popsány (viz obr. 3.24).

Datagrid může být ukotven na konkrétní buňce, podobně jako např. ukotvení v aplikaci Microsoft Excel [4]. Souřadnice ukotvené buňky jsou reprezentovány vlastností **AnchoredCellPosition**.

Styly sloupců tabulky jsou uloženy ve vlastnosti **IndexColumnStyle**



Obrázek 3.25: Diagram třídy SDGVColumnStyle

– pro indexační sloupec a ve vlastnosti **DataColumnStyles** – pro datové sloupce.

Jednotlivé styly jsou reprezentovány třídou **SDGVColumnStyle** (viz obr. 3.25). Těm pak lze definovat parametry:

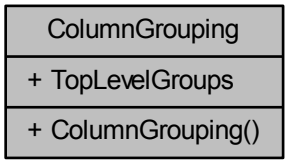
- šířka sloupce – vlastnost **Width**,
- formátovací řetězec pro zobrazení hodnot – vlastnost **Format**,
- zarovnání hodnot – vlastnost **Alignment**,
- hodnota, indikující zda je sloupec zvýrazněn – vlastnost **IsHightlighted**.

Datové pohledy zpravidla obsahují každý sloupec jen jednou. Při jejich tvorbě se totiž používají selekce sloupců, a i kdyby pohled obsahoval stejný sloupec vícekrát, selekce by jej stejně nebyla schopna rozlišit. Nicméně v datagridu je v některých situacích žádoucí mít v tabulce sloupec vícekrát. Proto nastavení datagridu obsahuje v podstatě zjednodušený datový pohled *SelectionView* – respektive je zde zobrazení, které definuje výsledně zobrazené sloupce v datagridu.

Zobrazení těmto výsledným sloupcům přiřazuje odpovídající zdrojové sloupce z datového pohledu. Konkrétně je toto zobrazení realizováno jako pole, kde indexy pole odpovídají indexům výsledných sloupců a hodnoty odpovídají indexům zdrojových sloupců.

V případě, že by v datovém pohledu byly sloupce a, b, c a v datagridu by bylo cílem zobrazit stejné sloupce, tak by zobrazení vypadalo takto: ($0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 2$).

Pokud by však cílem bylo zobrazit sloupec a ještě jednou za ostatní sloupce, odpovídající zobrazení by vypadalo takto: ($0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 2$, $3 \rightarrow 0$). Poslední dvojice v zobrazení ($3 \rightarrow 0$) by pak značila, že bude čtvrtý sloupec odpovídat prvnímu sloupci datového pohledu (indexace je od nuly).



Obrázek 3.26: Diagram třídy ColumnGrouping

Toto zobrazení je reprezentováno vlastností **DataColumnIndexesOfSourceDataColumns**.

Vlastnosti **AreNonWorkingDaysHighlighted** a **HolidayDays** jsou nastavení, přebíraná přímo z aplikace, nikoli z předpisu v souboru, proto nebudou diskutovány.

Poslední a asi nejsložitější částí je seskupení sloupců, reprezentované vlastností **ColumnGrouping**.

V datagridu mohou být sloupce organizovány do skupin. Jednotlivé skupiny vždy obsahují několik sloupců, z toho jeden je rodičem této skupiny. Skupiny mohou být libovolně zanořeny a tvoří tedy stromovou strukturu.

Třída **ColumnGrouping** (viz obr. 3.26), jejíž instance je právě uložena ve stejnojmenné vlastnosti, definuje toto seskupení.

Odkazy na nejvýše umístěné skupiny (kořeny stromů těchto skupin) najdeme ve vlastnosti **TopLevelGroups**. Jsou to instance třídy **Group**, která reprezentuje jednu skupinu.

Každá z těchto skupin má parametr, určující z jaké strany ji lze expandovat – vlastnost **ExpandDirection**. A také hodnotu, která indikuje, jestli je již skupina expandovaná – vlastnost **IsExpanded**.

Z hlediska stromové struktury skupin je zásadní vlastnost **GroupMembers**, ve které je posloupnost členů skupiny. Členem skupiny může být buď další skupina, nebo přímo sloupec (figurující jako list stromu).

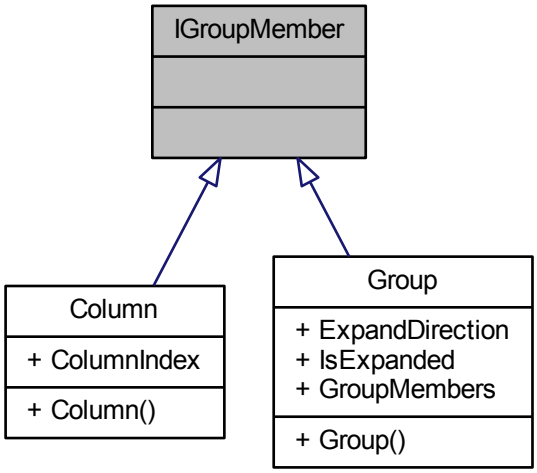
Aby bylo možno strukturu takto definovat, tak mají skupiny (třída **Group**) i sloupce (třída **Column**) společné rozhraní **IGroupMember** (viz obr. 3.27).

Jednotlivé sloupce pak obsahují index – vlastnost **ColumnIndex**, která udává, o jaký datový sloupec se jedná (dle jeho pořadí ve zdrojovém pohledu).

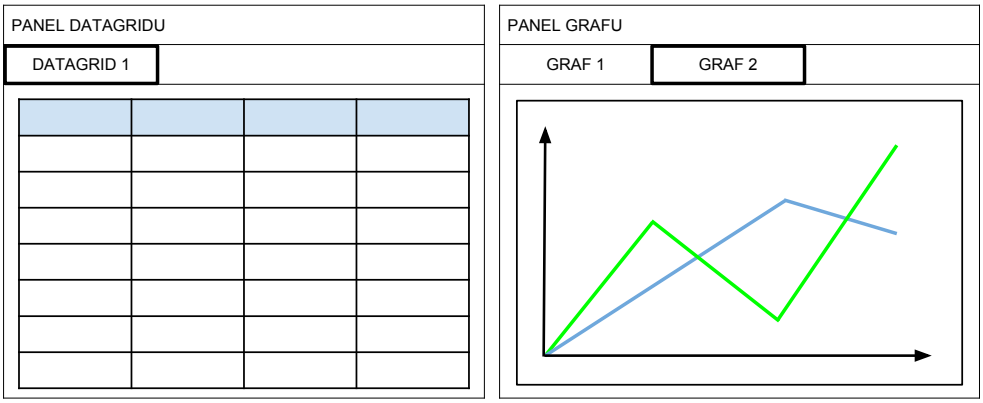
Vzhledem k tomu, že je příprava nastavení datagridu složitější, tak je zde pro testovací účely možnost k danému datovému pohledu vytvořit defaultní nastavení pomocí statické metody **CreateDummyParameters(viewDataColumnCount)**.

3.3.2 Popis struktury schématu pro zobrazovací komponenty

Jak bylo řečeno ve specifikaci, grafy i datagridy mají v aplikaci vlastní panel. Každý panel se dále dělí na jednotlivé záložky, kde jsou teprve konkrétní grafy a datagridy umístěny (viz obr. 3.28).



Obrázek 3.27: Hierarchický diagram tříd IGroupMember



Obrázek 3.28: Panely a záložky grafu a datagridu

Veškerá data pro přípravu obsahu pro tyto panely poskytuje schéma pro zobrazovací komponenty – třída **SchemeData** (viz obr. 3.29).

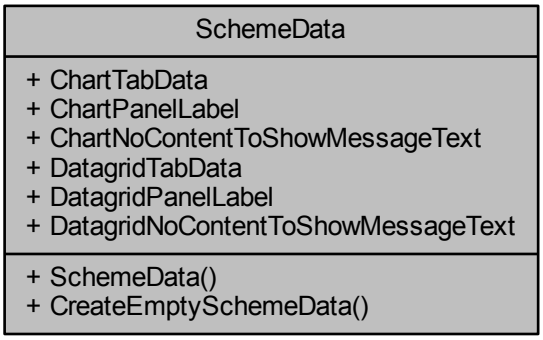
Každý panel má zde svůj titulek – vlastnosti **ChartPanelLabel** a **DatagridPanelLabel**.

V případě, že není pro daný panel připraven žádný obsah, je zde připraven text informační zprávy – vlastnosti **ChartNoContentToShowMessageText** a **DatagridNoContentToShowMessageText**.

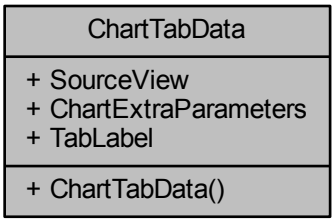
Data jednotlivých záložek jsou uloženy ve vlastnostech **ChartTabData** a **DatagridTabData**.

Vlastnost **ChartTabData** poskytuje kolekci instancí stejnojmenné třídy (viz obr. 3.30).

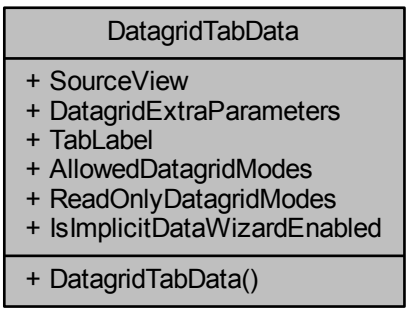
Tato třída obsahuje titulek záložky – vlastnost **TabLabel**.



Obrázek 3.29: Diagram třídy SchemeData



Obrázek 3.30: Diagram třídy ChartTabData



Obrázek 3.31: Diagram třídy DatagridTabData

Dále pak zmíněné data potřebné pro zkonstruování grafu:

- jeho datový pohled – vlastnost **SourceView**,
- jeho další nastavení – vlastnost **ChartExtraParameters**.

Obdobně jako data záložek grafu, poskytuje vlastnost **DatagridTabData** kolekci instancí třídy **DatagridTabData** (viz obr. 3.31).

Třída záložky datagridu také poskytuje její popisek – vlastnost **TabLabel** a data pro zkonstruování datagridu – vlastnosti **SourceView** a **DatagridExtraParameters**.

Dále jsou zde vlastnosti pro definici módu datagridu, které jsou spíše interní záležitost datagridu – vlastnosti **AllowedDatagridModes**, **ReadOnlyDatagridModes** a **IsImplicitDataWizardEnabled**.

3.3.3 Restriktivní rozhraní

Třída **SchemeData** je navržena tak, že se jednou vytvoří a dále zůstává **neměnná** a slouží pouze jako zdroj dat.

Tento fakt je zohledněn v její vnitřní implementaci i v jejím veřejném rozhraní. Obě části jsou pokud možno co nejvíce restriktivní.

Všechny její vlastnosti mají pouze getter metody. Vlastnosti reprezentované jako kolekce jsou speciálními typy kolekcí pouze pro čtení – generická třída **ReadOnlyCollection**. Ty umožňují k prvkům pouze přistupovat, nikoli je přidávat, nahrazovat, či kolekci jinak modifikovat [5].

Dále je omezení promítnuto i uvnitř implementace třídy. Členské proměnné, které jsou obaleny getter metodami vlastností, mají modifikátor **readonly**. To znamená, že do proměnných s tímto modifikátorem lze přiřadit hodnota pouze v konstruktoru a dále již do těchto proměnných nelze novou hodnotu přiřazovat [6].

Tento přístup – omezovat rozhraní a implementaci, pokud možno co nejvíce, jsem se snažil dodržovat při implementaci celého modulu. Vede to totiž k tomu, že kód je pak snáze pochopitelný a méně náchylný k chybám [1].

3.4 Předpis schématu v souboru

Tato část modulu (viz obr. 3.32) se zabývá předpisem schémat pro zobrazovací komponenty v souboru.

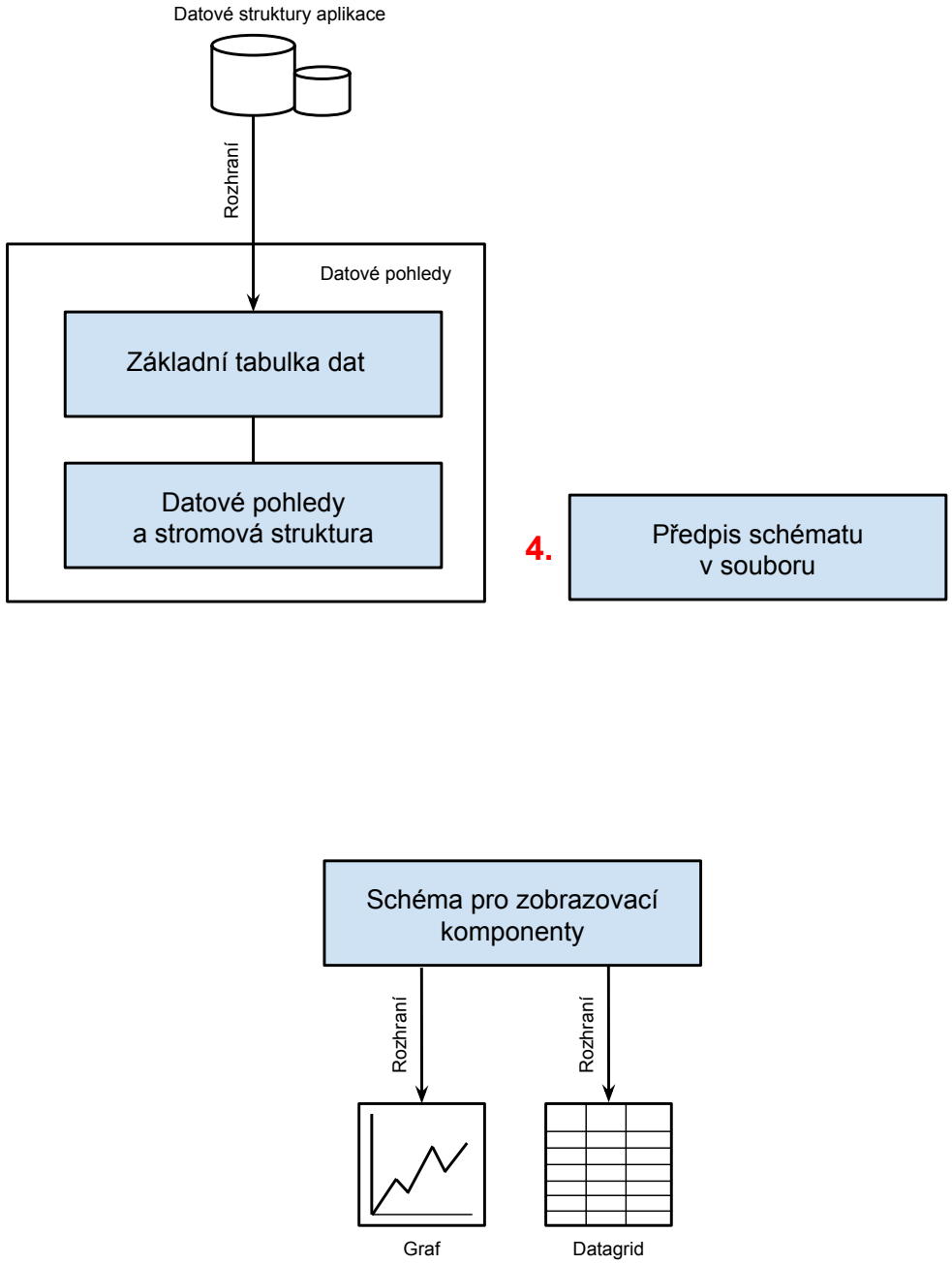
Je zde diskuze, co je vlastně potřeba v souboru předepsat, dále analýza možných formátů souborů a následné zdůvodnění výběru jednoho z nich. Také se zde nachází rozbor struktury předpisu.

3.4.1 Co je třeba předepsat

Předpis schématu musí obsahovat veškeré parametry potřebné pro sestavení schématu. Detailní rozbor těchto parametrů lze nalézt v předchozí kapitole o schématu pro zobrazovací komponenty.

Tyto parametry lze rozdělit do těchto skupin:

- definice datových pohledů pro jednotlivé grafy a datagridy,
- definice dalších parametrů (nastavení) pro jednotlivé grafy a datagridy,



Obrázek 3.32: Diagram znázorňující rozčlenění modulu – část 4, předpis schématu v souboru

- parametry panelů a záložek, v nichž jsou jednotlivé grafy a datagridy umístěny.

3.4.2 Volba formátu souboru

Nastavení panelů a záložek mají povahu jednotlivých parametrů nebo jejich posloupností. Definice nastavení zobrazovacích komponent také, až na výjimku u seskupení sloupců, které lze popsat stromovou strukturou. Definice datových pohledů by bývalo také mělo povahu stromovou, nebýt datového pohledu pro sloučení tabulek (*MergeView*), který, jak již bylo řečeno, může mít více předků.

Z tohoto hlediska se jeví definice datových pohledů jako nejsložitější část předpisu, a proto budu při diskuzi volby formátu především hledět na vhodnost právě vůči této části předpisu.

O struktuře datových pohledů lze říci, že se jedná o orientovaný, acyklický graf. Uzly tohoto grafu jsou pak zpravidla uvažovány v topologickém uspořádání [7].

První uvažovanou možností je soubor formátovat ve značkovacím jazyce XML. XML dokument je organizován jako strom, kde jednotlivé uzly jsou XML elementy, které mohou obsahovat text, XML atributy a další XML elementy [8]. Jedná se o standardní, velmi rozšířený formát, čili by byl bližší ostatním vývojářům aplikace.

Dále je pro formát XML podpora zabudovaná přímo ve standardních knihovnách jazyka C# – LINQ to XML [9].

Je zde však problém, že struktura pohledů není strom a tudíž ji nelze na strukturu XML uzlů mapovat přímo.

Další možností je formátovat soubor v jazyce DOT. Jedná se o jazyk, který popisuje grafy textově, pomocí uzlů a hran mezi nimi. V tomto jazyce jsou předepisovány grafy pro software Graphviz [10].

Pomocí tohoto jazyka by tedy bylo možné graf struktury datových pohledů popsat i přesto, že nemá stromovou strukturu.

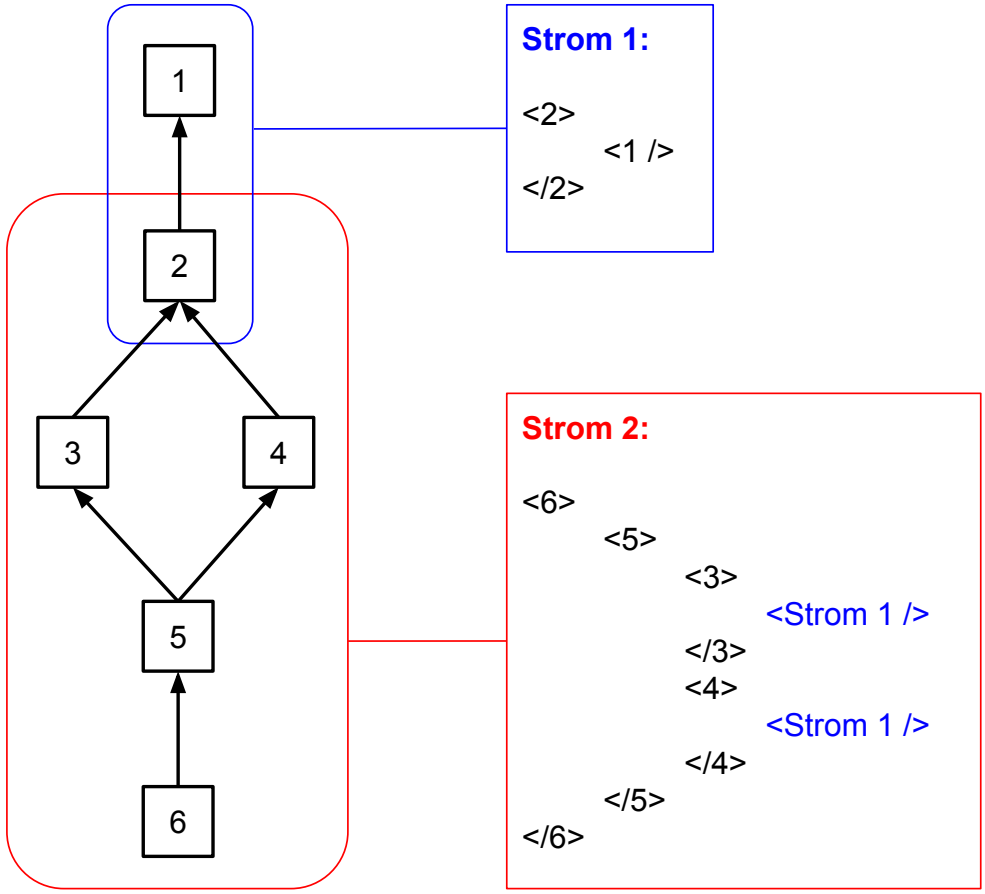
Další variantou a zároveň kompromisem je formátovat soubor v jazyce GraphML. Tento jazyk je založen na XML. Používají se v něm speciální XML elementy pro definici uzlů a hran grafu [11].

Oba tyto formáty jsou nicméně složitější než přímé využití stromové struktury uzlů formátu XML.

Při bližší analýze struktury datových pohledů se ukázalo, že lze jejich acyklickou grafovou strukturu rozložit na stromy, kde listy těchto stromů mohou být buď uzly bez následníků (základní tabulky), nebo odkazy na kořeny jiných stromů.

Za tohoto předpokladu tedy postačuje pro popis struktury datových pohledů využít stromové struktury jazyka XML.

Na následujícím obrázku 3.33 je znázorněn příklad rozložení nestromovitého grafu datových pohledů na stromy a odpovídající XML předpis. Pro



Obrázek 3.33: Rozklad struktury datových pohledů na stromy a odpovídající předpis pomocí XML

lepší představu, je v následující kapitole popsán algoritmus parsování těchto stromů.

Pro popis stromové struktury seskupení sloupců je XML formát také vhodný. Ostatní části předpisu se pak skládají jen z parametrů a jejich posloupností a pro jejich definici je XML také vhodným formátem.

Po zvážení těchto argumentů jsem zvolil pro přepis v souboru formát XML, zvláště pak pro jeho jednoduchost oproti ostatním formátům.

3.4.3 Jak je předpis strukturován, ukázka

Nejprve bude předveden ukázkový XML soubor pro lepší ilustraci – konkrétně jeho vybrané části. Poté následuje podrobnější rozbor struktury jednotlivých sekcí předpisu.

Na následující ukázce (viz obr. 3.34) je nastíněn význam a struktura nejdůležitějších částí předpisu.

```
<scheme>

  <tables>…</tables>

  <trees>

    <tree treeId="stromProGraf1">
      <selection viewId="pohledProGraf1" columnSelectionIds="sloupecA,sloupecE,sloupecD">
        <table tableId="zakladniTabulka" />
      </selection>
    </tree>

    <tree treeId="stromProGraf2">
      <selection viewId="pohledProGraf2" columnSelectionIds="sloupecB,sloupecC">
        <table tableId="zakladniTabulka" />
      </selection>
    </tree>

  </trees>

  <chartViews>
    <view viewId="pohledProGraf1" chartStyleId="stylGrafu" chartTabLabelResource="Záložka grafu 1" />
    <view viewId="pohledProGraf2" chartStyleId="stylGrafu" chartTabLabelResource="Záložka grafu 2" />
  </chartViews>

  <datagridViews>
    <view viewId="zakladniTabulka" datagridStyleId="stylDatagridu"
          datagridTabLabelResource="Záložka datagridu" />
  </datagridViews>

  <columnSelections>…</columnSelections>
  <chartStyles>…</chartStyles>
  <datagridStyles>…</datagridStyles>
  <datagridColumnStyles>…</datagridColumnStyles>

</scheme>
```

Obrázek 3.34: Ukázka z předpisu schématu v XML

Elementy nejvyšší úrovně rozdělují předpis na jednotlivé části. Elementy **chartStyles**, **datagridStyles** a **datagridColumnStyles** předepisují parametry zobrazovacích komponent. Jejich obsah je v této ukázce pro zjednodušení skryt. Ostatní části předpisu pak souvisí hlavně s datovými pohledy. Element **tables** předepisuje zdrojové tabulky dat a element **columnSelections** předepisuje selekce sloupců. V této ukázce je jejich obsah také skryt.

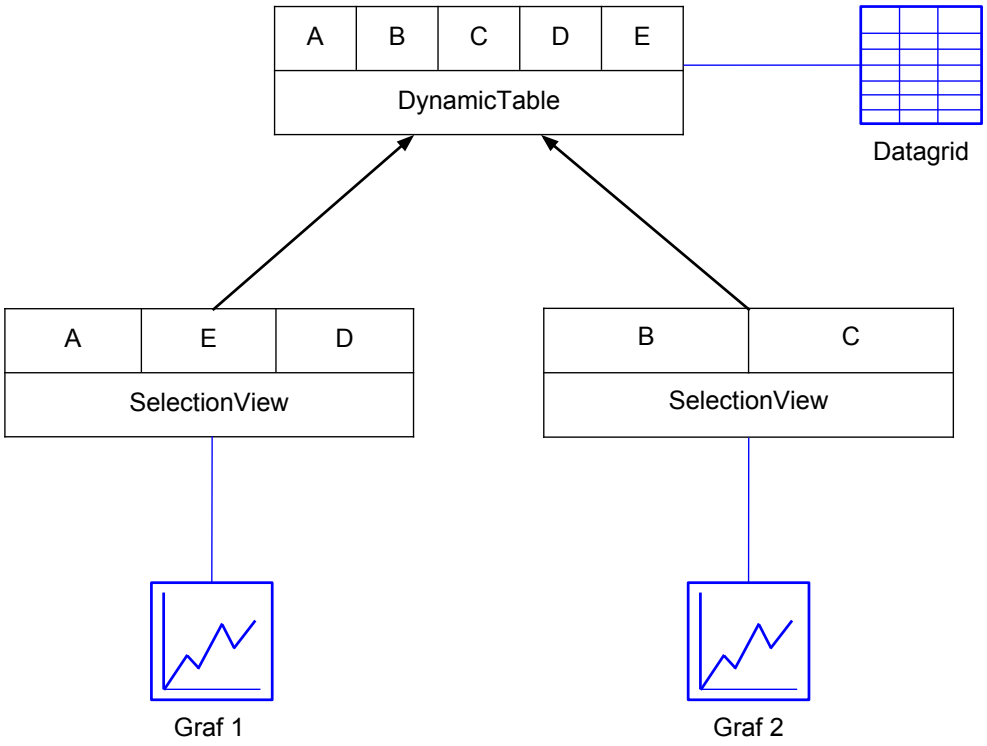
Zadání struktury datových pohledů a jejich navázání na jednotlivé zobrazovací komponenty k tomuto předpisu je znázorněno v následujícím diagramu (viz obr. 3.35).

Kořenem struktury je základní datová tabulka a na ní jsou navázány dva pohledy pro výběr sloupců.

Základní tabulka je definována v elementu **tables**. V elementu **trees** je pak část popisující strukturu datových pohledů. V této části je vidět definice pohledů pro výběr sloupců, použité selekce sloupců a jejich navázání na základní tabulku.

Předpis jednotlivých grafů, datagridů a jejich záložek je v elementech **chartViews** a **datagridViews**. Každá záložka zobrazovací komponenty má odkaz na datový pohled a na parametry pro tuto komponentu.

Jak pohledy, tak některé parametry jsou označeny identifikátory, aby se



Obrázek 3.35: Zadání k ukázkovému předpisu schématu

na ně bylo možné v jiných částech předpisu odkazovat.

Úplný předpis k této ukázce se nachází na přiloženém DVD.

3.4.4 Rozbor jednotlivých částí předpisu

3.4.4.1 Selekce sloupců

V elementu **columnSelections** se nachází definice selekcí sloupců. Každá selekce má svůj identifikátor **columnSelectionId**, který umožní se na selekce odkazovat při definici datových pohledů. Selekce mají dále uveden svůj typ (viz odvozené třídy od *ColumnSelection*) a obsahují posloupnost identifikačních parametrů – jejich názvů a hodnot.

3.4.4.2 Základní tabulky dat

Aby se mohlo schéma zkonstruovat, je potřeba k předpisu mít sadu základních tabulek dat, nad kterými lze stavět datové pohledy.

V této části předpisu – element **tables**, se seznam těchto základních tabulek vypíše a tabulkám se přidělí identifikátor **viewId**, který je následně využíván, aby bylo možné se na tabulky odkazovat při definici datových pohledů.

3.4.4.3 Stromy datových pohledů

V elementu **trees** se nachází předpis datových pohledů – jejich stromů.

Jednotlivé stromy jsou strukturovány tak, že jejich listy jsou základní tabulky nebo jiné stromy (resp. kořeny těchto stromů), které se v každém patře výše obalují datovými pohledy.

Stromy mají svůj identifikátor **treeId**, díky kterému mohou být použity u definice jiných stromů.

Jednotlivé datové pohledy jsou reprezentovány vlastním elementem. V definici pohledů lze používat zavedené selekce sloupců pomocí jejich identifikátorů **columnSelectionId** (viz posloupnost těchto identifikátorů v atributu *columnSelectionIds* v ukázce předpisu 3.34). Každý datový pohled je označen identifikátorem **viewId**, díky kterému se na něj lze při definici grafů odkazovat.

Kromě těchto parametrů, pak mohou mít datové pohledy další parametry, které jsou dány konkrétním typem pohledu (viz odvozené třídy od *DynamicView*).

3.4.4.4 Parametry pro graf

Parametry grafů jsou definovány v elementu **chartStyles**. Jednotlivé parametry jsou označeny identifikátorem **chartStyleId**, dle kterého se na ně při definici grafů odkazuje.

V současné době lze grafům parametrizovat pouze popisek osy Y.

3.4.4.5 Styly sloupců datagridu

Styly jednotlivých sloupců pro datagridy jsou v elementu **datagridColumnStyles**. Každý styl má svůj identifikátor **datagridColumnStyleId**, podle kterého se na něj lze z definice parametrů datagridu odkazovat.

Sloupcům lze předepsat šířku, formátovací řetězec, zarovnání a zvýraznění.

3.4.4.6 Parametry datagridu

V elementu **datagridStyles** jsou definovány parametry datagridu. Každé parametry mají svůj identifikátor **datagridStyleId**, na základě kterého se lze na parametry odkázat z definice datagridů.

Parametry datagridu mají definované souřadnice ukotvené buňky, seznam výsledných sloupců s jejich styly a seskupení sloupců (viz třída *SDGVExtraParameters*).

Seskupení sloupců má stromovou strukturu a je tedy přímo reprezentováno stromovou strukturou elementů.

3.4.4.7 Jednotlivé grafy

V elementu **chartViews** jsou definovány jednotlivé grafy, respektive jejich záložky. Každý graf má odkaz na svůj zdrojový pohled – **viewId**, odkaz na svoje parametry – **chartStyleId** a dále popisek záložky.

3.4.4.8 Jednotlivé datagridy

V elementu **datagridViews** jsou obdobně jako u grafu definovány jednotlivé datagridy – jejich záložky. Každý datagrid se také stejným způsobem odkazuje na svůj zdrojový pohled – **viewId**, a na své parametry – **datagridStyleId**.

3.5 Parsování předpisu schématu

Tato část modulu (viz obr. 3.36) se zabývá parsováním předpisu schématu ze souboru.

Jsou zde shrnuty vstupy a výstupy této části modulu, dále jsou zde popsány a zdůvodněny vybrané nástroje pro parsování. Nachází se zde také návrh tohoto submodulu, pořadí parsování a rozbor závislostí jednotlivých částí předpisu. Na závěr je zde popsán algoritmus načítání stromů datových pohledů.

3.5.1 Vstupy a výstupy

Vstupy pro parsování jsou:

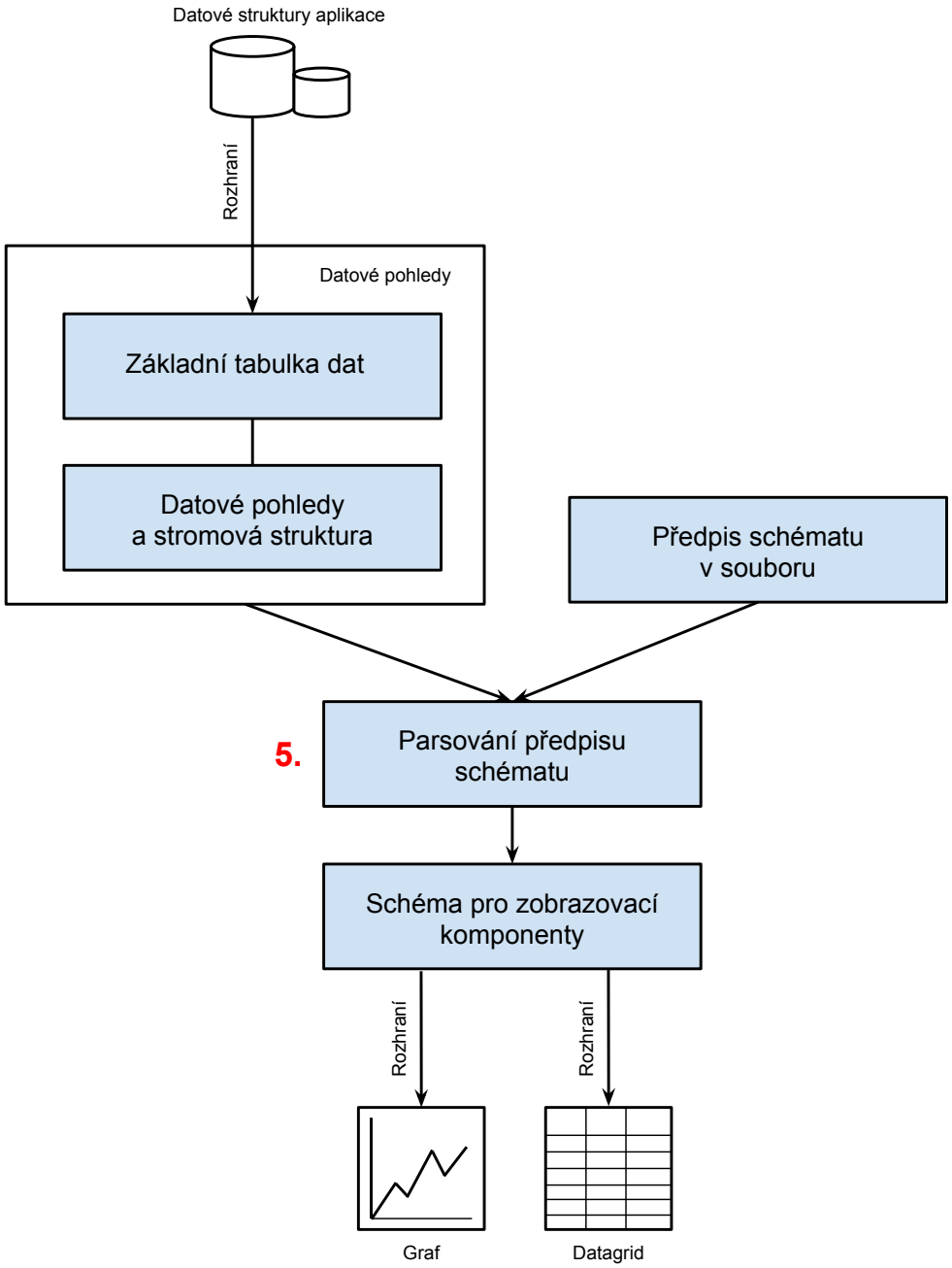
- Předpis schématu v souboru.
- Množina základních tabulek, nad kterými jsou datové pohledy stavěny.
- Nástroje ze submodulu datových pohledů.

Výstupem je pak sestavené schéma pro zobrazovací komponenty.

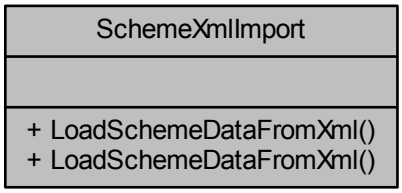
3.5.2 Nástroje pro parsování

Přímo v základních knihovnách pro jazyk C#, je zabudovaná podpora pro práci s XML dokumenty. Ta mimo jiné umožňuje jejich parsování a práci s načtenými dokumenty v paměti, čehož může být v tomto submodulu využito [12].

Pro práci s XML dokumenty v tomto submodulu byla tedy tato sada nástrojů zvolena.



Obrázek 3.36: Diagram znázorňující rozčlenění modulu – část 5, parsování předpisu schématu



Obrázek 3.37: Diagram třídy SchemeXmlImport

3.5.3 Návrh submodulu

Zde jsou představeny základní myšlenky návrhu submodulu pro parsování předpisu.

Každá část předpisu (části předpisu viz předchozí kapitola) má odpovídající třídu, která se zabývá jejím parsováním. Tyto třídy jsou v názvu označeny příponou „Parser“. Jsou to třídy statické a poskytují metodu pro parsování XML elementu odpovídajícího dané části předpisu.

Vzhledem k tomu, že prvky výsledného schématu bývají plně definovány ve více než jedné části předpisu (např. parametry datagridu *SDGVExtraParameters* lze sestavit po načtení části předpisu s jejich definicí a zároveň až se znalostí struktury datového pohledu, ke kterému mají být vztaženy), používají se pro částečně načtené reprezentace těchto prvků pomocné třídy. Tyto třídy jsou v názvu označeny příponou „Blueprint“.

3.5.3.1 Řídící třída submodulu

Statická třída **SchemeXmlImport** (viz obr. 3.37) tvoří rozhraní celému submodulu. Tato třída je jako jediná veřejná, ostatní třídy jsou viditelné jen v rámci submodulu.

Poskytuje metodu, která přijímá výše uvedené vstupy a vrací načtené schéma – **LoadSchemeDataFromXml(xmlString, sourceObjectTables)**. Metoda má ještě druhé přetížení, kde přijímá pouze jednu základní tabulku.

Uvnitř této metody se pak postupně volají jednotlivé třídy pro parsování a přeposílají se jim již načtená data.

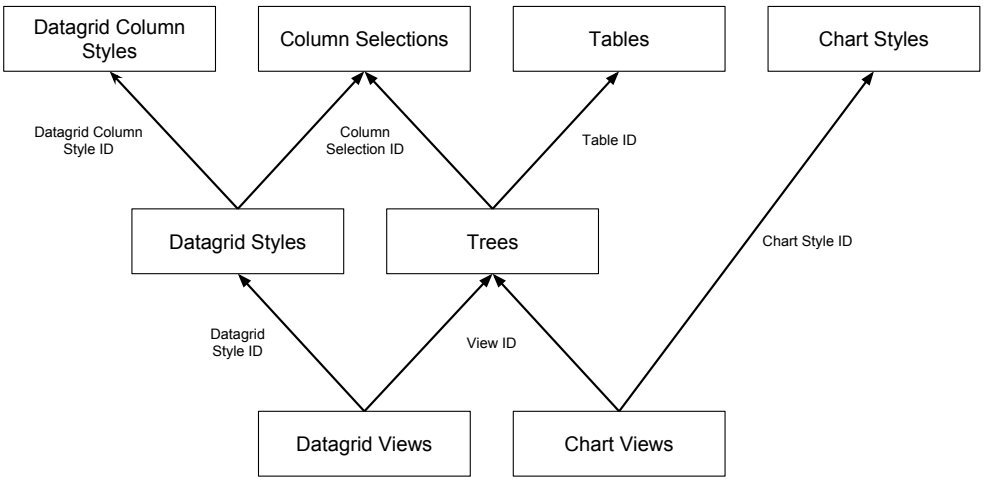
3.5.4 Postup při parsování předpisu

3.5.4.1 Rozbor závislostí mezi částmi předpisu

Na obr. 3.38 jsou znázorněny závislosti mezi jednotlivými částmi předpisu.

Z diagramu je patrné, že části předpisu s definicí stylů sloupců datagridu (*Datagrid Column Styles*), stylů grafu (*Chart Styles*), základních tabulek (*Tables*) a selekcí sloupců (*Column Selections*) jsou na ostatních částech nezávislé.

Část popisující stromy datových pohledů (*Trees*) potřebuje znát základní tabulky a selekce sloupců.



Obrázek 3.38: Znázornění závislostí mezi jednotlivými částmi předpisu schématu

Část se styly pro datagrid (*Datagrid Styles*) potřebuje znát styly sloupců datagridu a selekce sloupců.

Jednotlivé datagridy (*Datagrid Views*) pak potřebují znát styly datagridů a datové pohledy a jednotlivé grafy (*Chart Views*) zase styly grafů a datové pohledy.

Tyto závislosti do určité míry určují vhodné pořadí parsování jednotlivých částí předpisu.

3.5.4.2 Implementovaný postup

Implementované pořadí parsování jednotlivých částí předpisu je následující:

1. Nejprve se načtou selekce sloupců, základní tabulky a stromy datových pohledů – tím vznikají všechny datové pohledy.

2. Poté se načtou styly grafů a jednotlivé grafy. Obdobně se načtou styly sloupců datagridu a styly datagridů a následně jednotlivé datagridy.

3. Na závěr se připraví a vrací instance načteného schématu – *SchemeData*.

3.5.5 Algoritmus načtení stromů datových pohledů

Stromy datových pohledů mají jako své vnitřní uzly datové pohledy, které se mají vytvořit a jako své listy buď základní tabulky, nebo odkazy na kořeny jiných stromů (viz sekce Předpis schématu v souboru).

Zde je pomocí pseudokódu popsán algoritmus, podle kterého jsou stromy parsovány.

1. Množina **známých prvků** ← základní tabulky.
2. Množina **nezpracovaných stromů** ← všechny stromy z XML.
3. Pokud je množina **nezpracovaných stromů** prázdná – **hotovo**.
4. Najdi v množině **nezpracovaných stromů** libovolný strom, jehož všechny listy jsou z množiny **známých prvků**. (Pokud takový strom není, vyhoď výjimku – zadaný předpis není validní.)
5. Nalezený strom zpracuj – z jeho vnitřních uzlů vytvoř datové pohledy.
6. Tento strom přesuň z množiny **nezpracovaných stromů** do množiny **známých prvků** a pokračuj krokem 3.

3.6 Shrnutí spolupráce jednotlivých částí modulu

V předchozích kapitolách byly popsány jednotlivé části modulu. Pro shrnutí fungování modulu jako celku, popíši jeho typické použití, které se opírá o funkcionalitu všech jeho částí:

1. Zajistím, aby vybraná komponenta, ze které chci čerpat data, implementovala rozhraní zdroje dat základní tabulky – **IDynamicTableDataSource**.
2. K vybraným instancím implementujícím toto rozhraní vytvořím základní tabulky – **DynamicTable**. Tím mám připravena data ve formátu datových pohledů.
3. Navrhnu, jak bych chtěl výsledná data v aplikaci zobrazit a podle toho vytvořím soubor s předpisem schématu o správné struktuře a formátu. Pro operace nad daty v předpisu použiji zavedené datové pohledy (**SelectionView**, **AggregationView**, **MergeView**, ...).
4. Předpis schématu a základních tabulek zpracuji pomocí třídy **SchemeXmlImport** a tím získám výsledné schéma, které popisuje zobrazení dat v grafech a datagridech aplikace.
5. Požádám aplikaci o zobrazení tohoto schématu.

Kapitola 4

Testování modulu

Testování modulu probíhalo různými metodami, které jsou diskutovány v této kapitole.

Prvními testy, kterými prošla každá implementovaná funkcionalita modulu, bylo otestování dané funkcionality **pomocí ladících nástrojů** integrovaných ve vývojovém prostředí. Toto testování mělo za cíl vyzkoušet, jestli se implementace chová, tak jak bylo zamýšleno/navrženo, případně identifikovat v ní chyby či nedostatky v návrhu.

Dalším uskutečněným způsobem testováním byl **unit testing**. Pro vybrané třídy modulu byly napsány třídy, které realizovaly unit testy (třídy *OrderByViewTest*, *ReverseSignViewTest*, *GroupByAggregationViewTest*, *SchemeXmlImportTest*). Unit testy byly napsány tak, aby testovaly třídy proti jejich veřejným rozhraním, čili se jedná o black-box testy. Jeden z cílů těchto testů bylo otestovat nově implementované třídy, důležitějším cílem však bylo, aby tyto unit testy byly uchovány a vzhledem k tomu, že testují třídy proti jejich veřejnému rozhraní a to by se zpravidla nemělo příliš měnit, mohou dále sloužit jako automatizované, regresní testy. Jsou tedy určeny k tomu, aby při budoucích změnách v implementaci bylo možné automaticky otestovat, jestli předchozí funkcionalita stále funguje a nebyla poškozena [13].

Modul je součástí aplikace, na které pracují další programátoři a je průběžně testována **jako celek ostatními programátory** a tím se zároveň zčásti testuje i tento modul. Některé chyby v modulu byly tedy odhaleny ostatními programátory.

Pro zmíněné unit testy a dále pro budoucí potřeby testování byly při implementaci vytvořeny některé **pomocné metody a třídy**, které zjednodušují přípravu komplikovanějších struktur modulu. Jedná se o třídu *PrimitiveTableDataSource*, což je jednoduchá implementace rozhraní pro zdroj dat základní tabulky dat *IDynamicTableDataSource*. Tato třída, umožňuje vytvořit zdroj dat pro základní tabulku z dvojrozměrného pole hodnot. Dále pro vytvoření základní tabulky dat pro účely testování, je zde statická třída *DummyDynamicTableCreator*. Ta umožňuje vytvořit tabulku s náhodnými daty o požado-

vaném počtu řádků a sloupců pomocí metody *CreateDummyDynamicTable*. Pro vytvoření parametrů pro datagrid je zde statická metoda *CreateDummyParameters* ve třídě *SDGVExtraParameters*. Ta umožňuje pro specifikovaný datový pohled vytvořit parametry pro datagrid o defaultních hodnotách.

Některé předpoklady – **preconditions** v kódu jsou kontrolovány pomocí tzv. **asserts**. Ty umožňují zkontrolovat podmínku, která se bere jako předpoklad a v případě, že podmínka selže, pozastaví program a zobrazí chybovou hlášku [14]. Pomocí nich je kontrolováno, že nedochází k nepředpokládaným (chybným) stavům proměnných v programu nebo, že se program nedostal do neočekávaného místa. Tyto asserts testují spíše proti implementaci, a tudíž se jedná spíše o white-box testy. Mají za cíl, upozornit zavčas na chyby, které by jinak mohly být v jejich počátcích přehlédnuty/ignorovány a dále se projevovat jako hůře předvídatelné chyby na jiných místech [13].

Podobně k asserts, jsou ve zvláštních, problematických situacích **vyhazovány výjimky**. Například u některých veřejných rozhraní tříd probíhají validace parametrů, a pokud parametr není validní, je vyhozena výjimka – např. kontrola validity vstupních sloupců při konstrukci datových pohledů. Nebo pokud probíhá pokus o operaci, která je vzhledem k danému stavu objektu neplatná – např. snaha o zápis hodnoty do sloupce tabulky, který je ve stavu pouze ke čtení. Tyto výjimky mají za cíl upozorňovat, že došlo k chybové situaci, opět pokud možno v jejím zárodku. Asserts i výjimky tedy figurují v roli automatizovaných testů implementace, které probíhají při vykonání dané části programu [13].

Kapitola 5

Dokumentace modulu

Veškeré třídy modulu a jejich veřejná rozhraní – metody a vlastnosti, jsou zdokumentovány pomocí **XML dokumentačních komentářů**. Jedná se o standardní způsob dokumentace kódu v jazyce C#. Z těchto komentářů lze následně vygenerovat pomocí různých nástrojů dokumentace [15].

Dokumentace modulu byla z těchto komentářů vygenerována pomocí nástrojů **Doxygen** [16] a **Graphviz** [17] a je umístěna na přiloženém DVD.

Samotné algoritmy a doplňující popis implementace jsou pak zpravidla **okomentovány přímo v kódu**.

Při psaní kódu byl kladen důraz na to, aby **kód měl vypovídající hodnotu sám o sobě** a komentáře k němu byly potřeba jen jako coby shrnutí. Toho bylo například dosahováno vhodným strukturováním dílčích částí kódu do pomocných metod a také důrazem na to, aby proměnné a metody měly jednoznačný účel a aby jejich pojmenování mělo vypovídající hodnotu. Takto psaný kód pomáhá dokumentovat samotnou implementaci a orientovat se v ní [1].

Dalším způsobem dokumentace jsou napsané **unit testy** (viz kapitola Testování modulu). Ty ukazují na testovaných případech, jak se s danými třídami pracuje. Podobně potom **asserts a výjimky** dokumentují některé předpoklady (preconditions), povolené rozsahy parametrů, apod. Tento způsob dokumentace má výhodu v tom, že vzhledem k tomu, že je přímou součástí kódu, tak je méně náchylný stát se neaktuálním při budoucích změnách [1].

Kapitola 6

Použité nástroje

Modul byl napsán v programovacím jazyce **C#** za použití základních knihoven **.NET Framework Class Library**. Pro předpis schémat pro zobrazovací komponenty byl použit značkovací jazyk **XML**.

Jako vývojové prostředí bylo použito **Microsoft Visual Studio** [18], které napomohlo s organizací souborů programu, asistencí při psaní kódu, ladícími nástroji, nástroji pro kompilaci programu a podporou pro Unit testing.

Pro verzování, přehled o změnách a zálohování implementace byl použit systém **SVN** [19].

K organizování práce na implementaci přispěl nástroj **Redmine** [20]. Pomocí něj byly zadávány úkoly k implementaci dílčích částí modulu a k opravám nalezených bugů.

Pro generování dokumentace a diagramů tříd ze zdrojového kódu byly použity nástroje **Doxygen** [16] a **Graphviz** [17].

Pro sazbu textu bakalářské práce byl použit systém **LaTeX**.

Kapitola 7

Budoucí vývoj modulu

V současné době **se modul plně využívá ve stávající aplikaci** a zásahy v jeho implementaci už se dělají pouze výjimečně, zpravidla jako opravy nově nalezených bugů.

Samotná část datových pohledů (implementace jednotlivých datových pohledů, definice jejich rozhraní a ostatní jejich součásti) se ukázala být dostatečně obecná, že byla oddělena do samostatné knihovny, která se odkazuje pouze na všeobecné knihovny a na aplikaci je nezávislá – je tedy aplikací pouze využívána. Část schémat pro grafy a datagridy a jejich předpisů pak již s aplikací spolupracuje, neboť její součásti jako parametry pro graf a datagrid jsou specifické právě pro tyto zobrazovací komponenty v aplikaci.

Jak v aplikaci přibývají nové datové struktury, tak k nim **další programátoři předepisují XML schémata** a určují tak, co z těchto struktur bude uživateli v grafech a tabulkách zobrazeno.

Momentálně **se pracuje na rozšíření modulu o nový datový pohled**, který umožní provádět agregace nad skupinami řádků zdrojového datového pohledu. Kromě toho bude také umožňovat těmto skupinám řádků zadávat hodnoty hromadně.

Do budoucna by se modul mohl dále rozšiřovat v několika různých aspektech.

Agregace v datových pohledech jsou momentálně prováděny pouze nad datovým typem *double*, v případě potřeby by je bylo možné rozšířit o podporu dalších číselných datových typů. Místa, kde by byl potřeba provést tento zásah, jsou pak označeny jako slepé větve, ve kterých je vyhozena systémová výjimka *NotImplementedException*.

Do budoucna se dále **plánuje vytvořit GUI nástroj**, pomocí kterého bude možné schémata pro zobrazovací komponenty vytvářet pohodlně bez nutnosti sepisovat XML předpis. XML předpisy by však zůstaly a sloužily by jako podkladová data a GUI nástroj by pouze pracoval nad nimi a interně by je generoval.

Je zde také myšlenka, že schémata pro zobrazovací komponenty, a tedy

i jejich předpisy, by v budoucnu mohly být rozšířeny, aby zaštiťovala data nejen pro stávající grafy a datagridy, ale i pro **nově vzniklé zobrazovací komponenty**.

Závěr

Cílem práce bylo navrhnout, implementovat, otestovat a zdokumentovat modul, který umožní z datových struktur aplikace připravit obsah pro její zobrazovací komponenty.

Součástí modulu pak mělo být jednotné zpracování dat pro zobrazovací komponenty pomocí tzv. datových pohledů, které nad nimi umožní provádět operace výběru sloupců dat, aritmetické operace nad sloupci a jejich seskupování a řazení. Zobrazovacím komponentám měla být zajištěna notifikace o změnách v jejich podkladových datech. Měla zde být možnost předepsat, co se má k daným datovým strukturám v zobrazovacích komponentách zobrazit ve vhodně strukturovaném a uživatelsky čitelném zápisu. A dále pak definovat rozhraní, které budou implementovat datové struktury aplikace, aby na ně mohly být aplikovány datové pohledy a také definovat rozhraní, skrze které budou data poskytována zobrazovacím komponentám.

V práci se podařilo realizovat všechny vytyčené cíle. Na datové pohledy je nahlíženo skrze jednotné rozhraní (*IDynamicView*). Konkrétní implementace datových pohledů (*SelectionView*, *AggregationView*, *GroupByAggregationView*, *OrderByView*, ...) pak umožňují provádět nad daty požadované operace. Zobrazovací komponenty jsou o změnách v datech notifikovány skrze události (*DataChanged*, *ParametersChanged*), které jsou součástí rozhraní datových pohledů. Lze definovat předpis schématu, tedy to, co se má v zobrazovacích komponentách zobrazit pomocí strukturovaného, uživatelsky čitelného zápisu v jazyce XML a modul umožňuje tento zápis zpracovat. Bylo také definováno rozhraní, které umožňuje datových strukturám, aby na ně bylo nahlíženo skrze datové pohledy (*IDynamicTableDataSource*) a také rozhraní, skrze které jsou zpracovaná data poskytována zobrazovacím komponentám aplikace (*IDynamicView*, *ChartExtraParameters*, *SDGVExtraParameters*).

Realizace této bakalářské práce se zdařila. Modul je v současné době využíván v aplikaci a další programátoři jej používají a dále rozšiřují. Do budoucna jsou pak plánovány další rozšíření modulu jako GUI nástroj pro tvorbu předpisů schémat nebo implementace nových specializovaných datových pohledů

Literatura

[1] McConnell, S.: *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004, ISBN 0735619670, 9780735619678.

[2] Microsoft Corporation: Properties (C# Programming Guide). [online], [cit. 28. 4. 2015]. Dostupné z: `https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx`

[3] Refsnes Data: SQL GROUP BY Statement. [online], [cit. 28. 4. 2015]. Dostupné z: `http://www.w3schools.com/sql/sql_groupby.asp`

[4] Microsoft Corporation: Freeze or lock rows and columns. [online], [cit. 28. 4. 2015]. Dostupné z: `https://support.office.com/en-ca/article/Freeze-or-lock-rows-and-columns-3439cfe6-010c-4d2d-a3c9-d0e8ba62d724`

[5] Microsoft Corporation: ReadOnlyCollection<T> Class. [online], [cit. 28. 4. 2015]. Dostupné z: `https://msdn.microsoft.com/en-us//library/ms132474`

[6] Microsoft Corporation: readonly (C# Reference). [online], [cit. 28. 4. 2015]. Dostupné z: `https://msdn.microsoft.com/en-us//library/acdd6hb7.aspx`

[7] Black, P. E.: directed acyclic graph. [online], [cit. 28. 4. 2015]. Dostupné z: `http://www.nist.gov/dads/HTML/directAcycGraph.html`

[8] Refsnes Data: XML Tree. [online], [cit. 28. 4. 2015]. Dostupné z: `http://www.w3schools.com/xml/xml_tree.asp`

[9] Microsoft Corporation: LINQ to XML. [online], [cit. 28. 4. 2015]. Dostupné z: `https://msdn.microsoft.com/en-us/library/bb387098.aspx`

[10] AT&T Labs Research: The DOT Language. [online], [cit. 28. 4. 2015]. Dostupné z: http://www.graphviz.org/doc/info/lang.html

[11] GraphML Working Group: The GraphML File Format. [online], [cit. 28. 4. 2015]. Dostupné z: http://graphml.graphdrawing.org/

[12] Microsoft Corporation: XML Documents and Data. [online], [cit. 28. 4. 2015]. Dostupné z: https://msdn.microsoft.com/en-us/library/2bcctyt8.aspx

[13] Krátký, T.: Software testing. [online], [cit. 28. 4. 2015]. Dostupné z: http://www.profinit.eu/fileadmin/Content/profinit.eu/Academy/NSWI129/prednasky/05_Testing.pdf

[14] Microsoft Corporation: Debug.Assert Method. [online], [cit. 28. 4. 2015]. Dostupné z: https://msdn.microsoft.com/en-us/library/system.diagnostics.debug.assert

[15] Microsoft Corporation: XML Documentation Comments (C# Programming Guide). [online], [cit. 28. 4. 2015]. Dostupné z: https://msdn.microsoft.com/en-us/library/vstudio/b2s063f7

[16] van Heesch, D.: Doxygen. Version 1.8.9.1.

[17] Gansner, E. R.; North, S. C.: An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, ročník 30, č. 11, 2000: s. 1203–1233.

[18] Microsoft Corporation: Microsoft Visual Studio Express 2013 for Windows Desktop.

[19] Stefan Küng, L. O.: TortoiseSVN. Version 1.8.0, Build 24401 - 64 Bit.

[20] Lang, J.-P.: Redmine. Version 2.5.2-2.

Příloha **A**

Seznam použitých zkratek

GUI Graphical User Interface

LINQ Language Integrated Query

SQL Structured Query Language

SVN Subversion

XML Extensible Markup Language

Příloha **B**

Obsah přiloženého DVD

- `BP_Klíma_Jakub_2015.pdf` ..... text práce ve formátu PDF
- `readme.txt` ..... stručný popis obsahu DVD
- `sample_scheme_definition.xml` ..... ukázkový předpis schématu pro zobrazovací komponenty
- `doxygen_html` ..... HTML dokumentace zdrojového kódu
  - `index.html` ..... úvodní stránka HTML dokumentace
- `src_thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `images` ..... obrázky ke zdrojové formě práce