VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

2014 **Bc. Jan Ducháč**

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Katedra informatiky

API pro zobrazení výstupů diagnostických metod formou grafů

API for displaying of diagnostic methods outputs in graphs

2014 **Bc. Jan Ducháč**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Ducháč**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: API pro zobrazení výstupů diagnostických metod formou grafů
API for Displaying of Diagnostic Methods Outputs in Graphs

Zásady pro vypracování:

API pro zobrazení výstupů diagnostických metod formou grafů. Dále webové rozhraní pro zpracování vstupních parametrů, knihoven a exportu do grafických formátů.
1. Analyzovat stávající API pro zobrazování cca 25-ti grafů.
2. Přepracovat výchozí strukturu (renderers, plots, markers, datasets, charts, axis, aj.) dle nových potřeb a zajistit kompatibilitu s novou verzí JFreeChart.
3. Implementace speciálních renderů a plotů pro specifické požadavky grafů.
4. Integrační testy.
5. Webové rozhraní pro generování a export grafů do grafických formátů.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Daniel Holešínský**

Konzultant diplomové práce: Ing. Martin Němec, Ph.D.

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 7.5.2014

.....

podpis

Abstrakt česky

Práce představuje jednotné aplikační rozhraní (API) pro generování výstupů diagnostických metod formou grafů. Je určeno pro potřebu firmy DAP Services. Zahrnuje analýzu a přepracování stávajícího rozhraní a vlastní knihovny grafů, vycházející ze standardní JFreeChart knihovny. Při stávajícím stavu jsou grafy obtížně definovatelné, mají mnoho redundancí a je komplikovaná rozšiřitelnost. Výsledkem je rozhraní, které má snadné zadávání dat a konfiguraci grafů.

Klíčová slova : Application Programming Interface (API), JFreeChart, Chart, Dataset, Render, Plot, JSON format, Java SE

Abstrakt english

The aim of this work is create the uniform application programming interface (API) to generace outputs diagnostic methods in graphs. It is designed to meet business needs DAP Services. Includes analysis and recasting the existing interface and libraries graphs, based on the standard JFreeChart library. At the current state of the charts are difficult to define, they have a lot of redundancy and extensibility is complicated. The result is an interface that is easy data entry and configure charts.

Keywords : Application Programming Interface (API), JFreeChart, Chart, Dataset, Render, Plot, JSON format, Java SE

Seznam použitých zkratek a symbolů

API Application Programming Interface – rozhraní pro komunikaci nebo propojení (často) vnitřních zdrojových kódů s jinými prostředími či programy

Java – programovací jazyk

JFreeChart – standardní java knihovna pro tvorbu grafů

Dataset – soubor vstupních dat (hodnot) pro vygenerování grafů

Render – třída, která provádí samotné vykreslení jednotlivých prvků, u knihovny JFreeChart využívá standardní třídu Graphics2D

Plot – vnitřní plocha vykresleného grafu (neobsahuje titulek ani legendu)

JSON format – jednotný formát pro výměnu dat, jednoduše čitelný

Obsah

1. Úvod

Firma DAP Services, která se zabývá mj. diagnostickými metodami barvových asociací, potřebuje aktualizovat a přepracovat stávající způsob generování výsledných grafů do formátů hlavně jpeg a png, později rozšíření na další např. svq. K tomuto účelu je potřeba vytvořit jednoznačné, snadno definovatelné a rozšiřitelné prostředí.

V první kapitole se seznámíme se standardní knihovnou JFreeChart, její charakteristikou, datovou strukturou a blíže si rozebereme nejpoužívanější pojmy, které budou uváděny v dalších kapitolách. Cílem není podrobně rozepsat a definovat všechny její části, ale vytvořit dostatečný podklad pro rychlé pochopení a orientaci v pojmech a knihovně JFreeChart obecně.

Stejně tak druhá kapitola má za úkol nám přiblížit co to API obecně je, určitý návrh a jaký má vývojový proces.

Další část je zaměřena na praktičtější stránku věci, a to je Analýza stávajícího stavu, definování požadavků, jaké existují známé omezení a možné rizika.

Nato se dostáváme ke kapitole Praktické řešení, kde je popsán vývoj návrhu nového rozhraní, řešení a celková struktura.

Poslední část je hlavně prezentační, kde je uveden výběr výsledných grafů, a také je zde zmíněno jejich testování a další možnosti směřování.

2. Kapitola : JFreeChart

Obsahuje výstižný popis o čem knihovna JFreeChart je, co obsahuje, jejího aplikačního rozhraní a struktury.

Jedna z nejrozšířenějších knihoven pro tvorbu grafů, plně napsána v jazyce java. Vykreslení se provádí standardním Java2D API. Projekt byl založen požátkem roku 2000 a do dnešní doby pokračuje a je stále veden zakladatelem Davidem Gilbertem. K dnešnímu dni má více než 2,1 milionů stažení. Aktuální verze JFreeChart knihovny je 1.0.17 (2013-11-24). Je závislá na další knihovně JCommon, která je distribuována společně, aktuálně ve verzi 1.0.22 (2014-02-28) [1].

2.1 Vlastnosti

Má širokou škálu typů grafů, standardní a ucelený design, který lze překonfigurovat a rozšířit, a také podporuje mnoho typů výstupů (JPEG, PNG) včetně Swing komponent a formáty vektorové grafiky včetně PDF, EPS, SVG.

Jelikož je napsána v javě, může být použita jak u desktopové, tak i u webové aplikace. Na straně klienta jako applet a na straně serveru jako servlet. JFreeChart je knihovna tříd pro použití vývojáři, nikoliv pro koncového uživatele.

Licence

Licence má GNU Lesser General Public Licence (LGPL) [1], což znamená, že licence neomezuje programátora a lze ji tudíž přeprogramovat pro specifické požadavky. Hodí se jak pro tvorbu proprietárního softwaru (komerčního), tak i pro open source programátory.

Dokumentace

Podrobná a plnohodnotná elektronická dokumentace je placená, od 65 USD (cca 1300,-Kč) za osobní licenci až po 1500 USD (cca 30 000,- Kč) za licenci firemní. Bezplatná dokumentace je pouze ve formě javadocu, neboli vygenerový popis aplikačního prostředí [1]. Tento systém

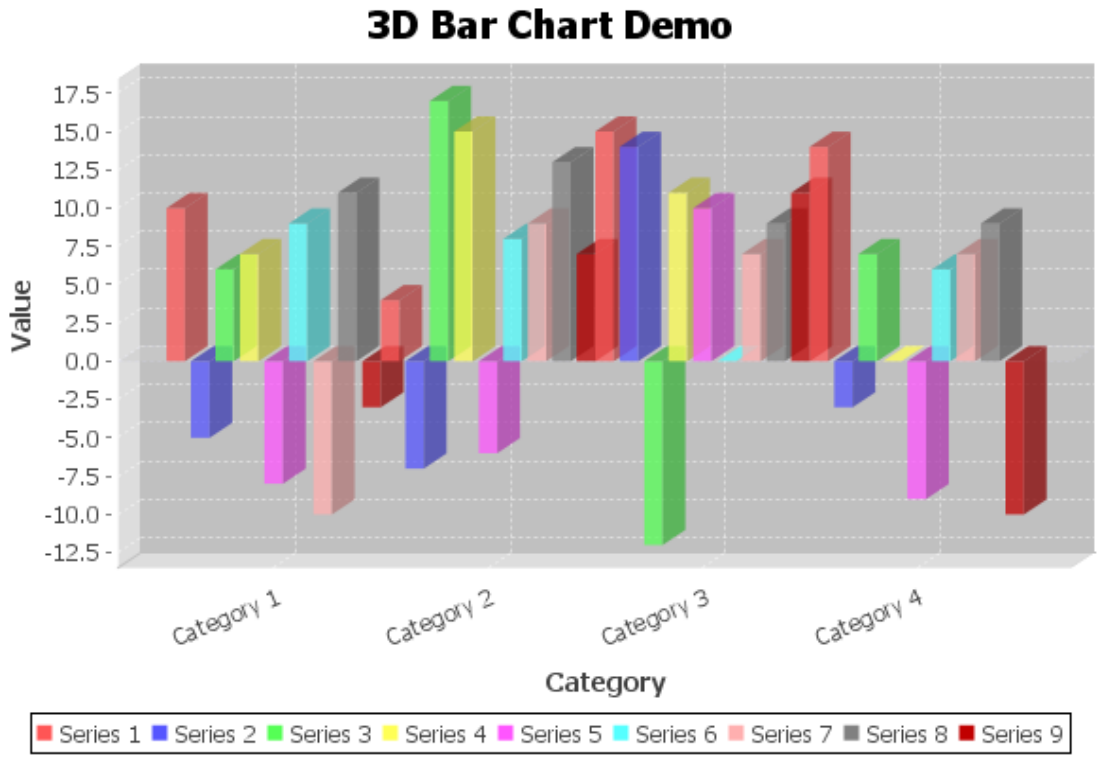
placené dokumentace je spíše v rozporu se stávajícím trendem volně či za levně dostupných plně popsaných dokumentací, návodů, příkladů i popsaným kódem pro získání co největšího počtu uživatelů. Na druhou stranu je z tohoto druhu příjmu financován další vývoj.

Omezení

Vzhledem k tomu, že je knihovna postavena na platformě java a je zároveň open-source, tak má obrovské možnosti využití, modifikace, rozšiřitelnosti.

Co neumí je zobrazovat dynamické procesy v reálném čase, to lze řešit asi jen automatickým překreslováním, které může být výkonově náročné.

Dále všechny grafy jsou vykreslovány do dvourozměrných polí, některé 3D prvky jsou pouze a jen vizualizací, kde pro vykreslení používají grafickou standardní knihovnu Graphics2D (jako např. BarRenderer3D). Použití lze vidět na následujícím obrázku.

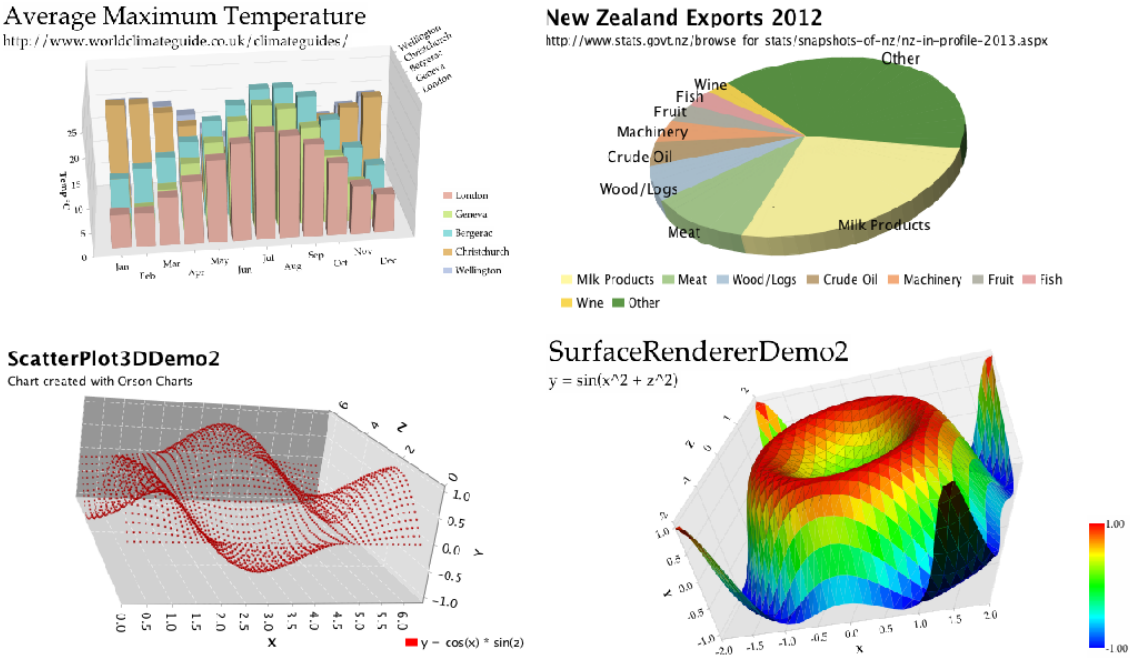


Obrázek 2.1: Příklad 3DBarChart

Pro zobrazení trojzorměrných polí a modelování např. fyzikálních systémů je vhodné použít speciální aplikace jako ParaView, Tulip, a pro statistické účely např. gnuPlot, R [2].

OrsonChart3D

Zajímavou novinkou ve verzi 1.0.17 je nový modul OrsonChart3D [1], který dokáže generovat několik typů 3D grafů (vizuálně). Částečně se zlepšila a zjednodušila definovatelnost vstupních hodnot a i grafická reprezentace. Osobně jsem tuto knihovnu zatím neměl možnost blíže prozkoumat a nejsem si jist, zda vychází či vycházela ze sady Java3D i když na ni a ani na OpenGL není závislá.



Obrázek 2.2: Příklady OrsonCharts3D [3]

Požadavky

Aktuální verze vyžaduje platformu Java 2 (JDK verze 1.6.0 nebo vyšší). Pro webové rozhraní (applet, servlet) Java EE.

2.2 Obsah knihovny

Podporuje několik desítek typů grafů a mezi samozřejmosti patří sloupcový (BarChart),

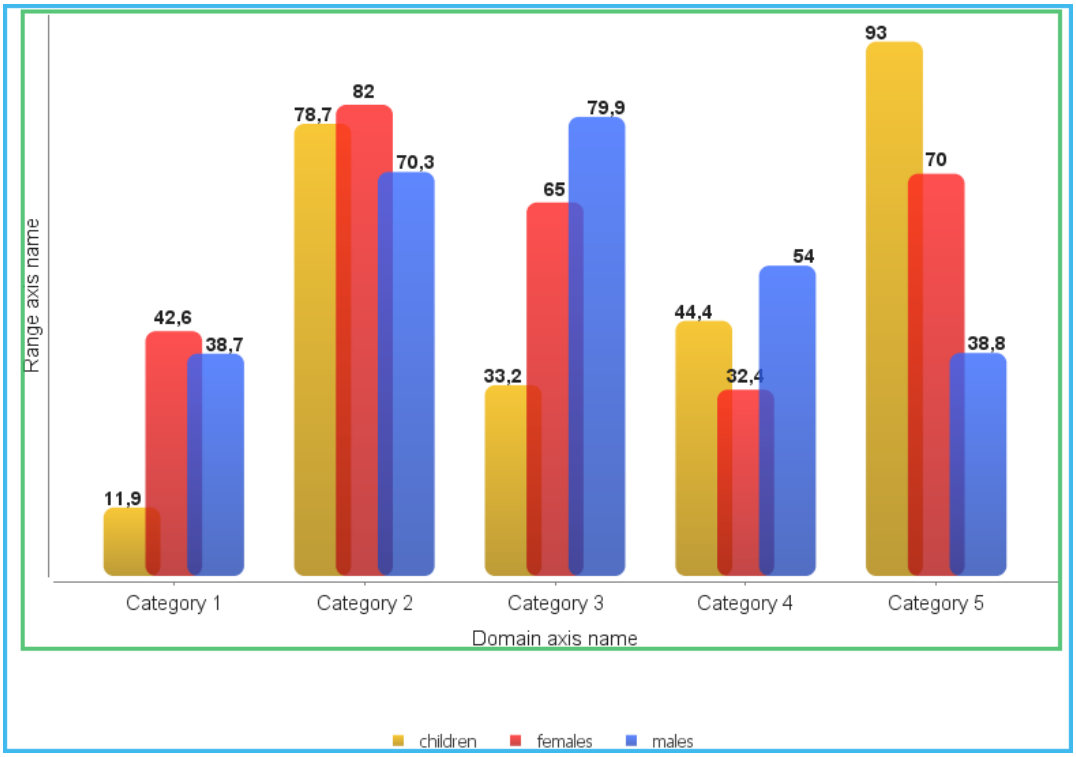
kruhový (PieChart), prstencový (DonutChart), bodový (XYChart), spojnicový (LineChart), polární graf (PolarChart), krabicový (BoxChart), a histogram a další jejich modifikace. Na stránkách JFreeChart - Samples [1] lze najít ukázky příkladů různých typů a hlavně stažitelné demo.

Pro rozsáhlost obsahu, a také částečného odklonu od předmětu této práce, zde nebudu jednotlivé grafy popisovat.

2.3 API

Skladba knihovny je postavena na principu oddělení vstupního souboru dat a jejich prezentace. Jsou zde uvedeny jen nejdůležitější části pro pochopení základní struktury a principu.

Rozložení jednotlivých částí si popíšeme za pomocí následujícího obrázku (*Obrázek 2.3*). Jednotlivé části jsou popsány podrobněji dále.



Obrázek 2.3: Rozložení grafu

Hranice grafu jsou znázorněny modrým rámečkem. Tuto část zajišťuje převážně hlavní třída

JFreeChart. O plochu ohraničenou zeleným rámečkem se stará tzv. *Plot* neboli plocha grafu. Její součástí jsou popisky kategorií a nesmíme zapomenout také na osy. O vykreslení datových struktur (v našem případě jednotlivé sloupce s hodnotami) se stará *Renderer*.

Prezentace

Základem je třída JFreeChart, která se stará o samotné vykreslení grafu jako celku za pomocí Graphics2D (metody draw). Kromě toho se stará o tzv. okolí grafu, tedy titulku, legendy, ohraničení, pozadí apod. Hlavní složkou je ale abstraktní třída Plot, která reprezentuje samotný graf.

Plot

Prakticky nejdůležitější část vůbec. Tato abstraktní třída zajišťuje správu vnitřních popisků datasetu, renderů, os a předává nastavení a hodnoty pro pokročilé grafické vykreslení. Tím, že je abstraktní tak zajišťuje jednoznačnou funkcionalitu pro všechny své potomky a jim přenechává její implementaci právě z důvodů různých typových struktur (Category, Pie apod.). Taktéž je zde definována skupina tříd, která dědí z této abstraktní třídy.

Plot
- *CategoryPlot*
- *CompassPlot*
- *ContourPlot*
- *FastScatterPlot*
- *MeterPlot*
- *MultiplePiePlot*
- *PiePlot*
- *PolarPlot*
- *SpiderWebPlot*
- *ThermometerPlot*
- *WaferMapPLot*
- *XYPlot*

Každá třída má své základní nastavení pro rychlé zobrazení dat bez další konfigurace.

Renderer

Koncové zpracování vstupních dat do podoby požadovaného grafu – např. sloupce. Provádí transformace bodů do zobrazované plochy a vykresluje jednotlivé prvky dle nastavených parametrů předaných z Plotu. Základní třídou je zde AbstractRenderer, který se dělí ještě na dva typy, a to AbstractCategoryItemRenderer a AbstractXYItemRenderer.

Z „category" typu vychází mimo mnoho jiných také AreaRenderer, BarRenderer nebo LineAndShapeRenderer, ze kterých dědí další specifické renderery BarRenderer3D, IntervalBarRenderer nebo LineRenderer3D.

Ve druhé skupině můžeme naleznout základní StandardXYItemRenderer, VectorRenderer, XYAreaRenderer, XYBarRenderer či XYLineAndShapeRenderer.

```
AbstractRenderer
   | AbstractCategoryItemRenderer
      | AreaRenderer
      | BarRenderer
      | LineAndShapeRenderer
      | …
   | AbstractXYItemRenderer
      | StandardXYItemRenderer
      | VectorRenderer
      | XYAreaRenderer
      | XYBarRenderer
      | XYLineAndShapeRenderer
      | …
```

Takže pokud bychom chtěli upravit např. zaoblení sloupců u sloupcovitého grafu, musíme vytvořit vlastní Renderer, který dědí např. z BarRendereru a přepsat jeho vykreslovací metody. A nezapomenout implementovat použití nového renderu pro daný plot.

Axis

Nedílnou součástí u grafů jsou také osy či osa. Výchozí je opět abstraktní třída Axis, která má dva hlavní potomky, a to CategoryAxis a ValueAxis.

CategoryAxis představuje osu pro kategorie a při vertikální orientaci grafu si ji můžeme

klasicky představit jako osu x. Druhá ValueAxis reprezentuje osu hodnot (míry nebo času).

```
Axis
   | CategoryAxis
      | CategoryAxis3D
      | ExtendedCategoryAxis
      | SubCategoryAxis
   | ValueAxis
      | DateAxis
      | LogAxis
      | NumberAxis
            | CyclicNumberAxis
            | …
      | PeriodAxis
```

Má svůj textový popis, který v mnoha případech slouží i jako identifikátor. Takže v případě vícero serií můžeme tyto data kategorizovat do skupin jako právě na předchozím obrázku *2.3*, kde máme tři série (Děti, Ženy a Muže) porovnány pospolu v šesti jedinečných kategoriích. V závislosti na typu grafu se liší i vlastní nastavení os mimo základní nastavení (např. velikost stupnice či odsazení).

Dataset

Datovou množinu představuje objekt Dataset, který může obsahovat od jedné hodnoty až po list sérií dat. V závislosti na typu grafu je definována skupina rozhraní a tříd.

```
Dataset
-  CategoryDataset
      | org.jfree.data.category.DefaultCategoryDataset
-  KeyedValueDataset
      | org.jfree.data.general.DefaultKeyedValueDataset
-  PieDataset
      | org.jfree.data.general.DefaultPieDataset
-  SeriesDataset
```

```
        | org.jfree.data.general.AbstractSeriesDataset
- ValueDataset
        | org.jfree.data.general.DefaultValueDataset
```

CategoryDataset má použití hlavně u sloupcových a plošných grafů. Ale nejrozšířenější použití má SeriesDataset, a to převážně pro čárové a bodové XY grafy. Pro koláčové grafy a tachometer grafy je speciální PieDataset.

DatasetUtilities

Užitečnou utilitou pro práci s datasety je třída DatasetUtilities, která nám usnadňuje vytváření, manipulaci, konverzi a vyhledávání v datasetu.

```
org.jfree.data.general.DatasetUtilities
```

Napojení na databázi

Další možností je přímé připojení na databázi. Pro tyto účely lze využít třídy JDBCCategoryDataset, JDBCPieDataset, JDBCXYDataset z balíčku

```
org.jfree.data.jdbc
```

Pro připojení slouží standardní třída java.sql.Connection.

XML reader

Pro čtení XML dokumentů slouží třída DatasetReader, která podporuje dva typy datasetů, a to CategoryDataset a PieDataset. Podporované xml tagy:

```
CategoryDataset or PieDataset
    | Series
        | Item
            | Key
            | Value
```

ChartFactory

Ideální pro jednoduché vytváření různých typů grafů. Co nelze nikdy předdefinovat je pouze dataset, který je vždy parametrem.

Příklad implementace

Existují dvě možnosti jak inicializovat graf. První je za pomocí základního konstruktoru JFreeChart anebo využití třídy ChartFactory s mnoha statickými metodami pro tvorbu různých typů grafů. Důležité objekty jsou Plot (vnitřní reprezentace grafu) a Dataset (soubor vstupních hodnot). Rozdíl v těchto dvou použitích je jen v inicializaci třídy Plot. Všechny parametry a oblasti lze nakonfigurovat i zpětně než budeme chtít graf vykreslit. Jde jen o styl, přehlednost a čistotu. Nakonfigurovaný JFreeChart objekt můžeme pomocí třídy ChartUtilities uložit do souboru anebo pouze vykreslit pomocí třídy ChartFrame.

```
DefaultValueDataset dataset = new DefaultValueDataset();
// dataset.set… {value}
MeterPlot plot = new MeterPlot(dataset);
// plot.set… {configure}
JFreeChart chart = new JFreeChart(plot);

ChartUtilities.saveChartAsJPEG(file, chart, width, height);
or
ChartFrame frame = new ChartFrame("Title", chart);
frame.pack();
frame.setVisible(true);
```

2.4 Souhrn

Knihovna JFreeChart je dnes jedna z nejsilnějších a nejrozšířenějších knihoven na světě na platformě java. Její možnosti využití, modifikace a rozšiřitelnosti v rámci programátorského prostředí jsou obrovské. Každá firma, organizace či zadavatel může mít rozdílné požadavky, tak je jasné, že nelze vyjít všem vstříc. Navrhnout jednotné řešení a zároveň zachovat určitou

kontinuitu a jádro knihovny, která si zachovává stále svou původní formu a čistotu bývá často obtížné.

Desktopová aplikace (uživatelské prostředí), která souběžně reaguje na nové verze JFreeChart je velmi užitečná pro všechny uživatele bez nutnosti umět programovat. Přímá manipulace s grafem je velmi užitečná (např. odečet hodnot z kterékoliv části grafu) a samozřejmě exportu do široké škály podporovaných výstupních formátů.

JFreeChart ale není žádnou dokonalou knihovnou, má své mouchy, které vychází z široké škály rozdílných typů grafů a jejich nepřeberné množství parametrů. Nepodporuje všechny odvětví, kde je potřeba zobrazovat data. Není důvod.

Vzhledem k neustálému růstu a zdokonalování 3D vizualizací ve světě informatiky je příjemným přírůstkem OrsonChart3D.

Co bych opravdu vytkl, jsou nevhodně zvoleny výchozí barvy, které se nezměnily od první stabilní verze. V informatice je vývoj za posledních deset let neuvěřitelný a takové vizuální osvěžení je dnes již nutností pro jakýkoliv starší produkt. Je stále ve vývoji, a tak se snad někdy setkáme i s trochu živějšími barvami.

3. Kapitola : API

Zaměří se hlavně na návrh a vývojový proces API obecně, jaké mohou být důsledky špatně navrženého API [4].

Pod pojmem aplikační rozhrazní (Application programming interface, dále jen API) si můžeme představit něco jako prostředníka, který propojuje vstupně výstupní části (I/O) mezi dvěma subjekty či více. Subjektem máme na mysly např. podsystém, jiný program, jiné API nebo také uživatel či programátor sám. Hlavním důvodem vzniku rozhraní je separace. Může jít o oddělení stran, povinností, úrovní abstrakce, překladů, vývoje apod. Těmto odděleným částem umožňuje rozhraní komunikovat ustáleným způsobem. Příklady lze nalézt kdekoliv.

API platforem:

- operační systémy
- jazykové runtimes (Java, .NET, OpenGL)
- jazyky samotné

API programů:

- Firefox, IntelliJ IDEA, Eclipse, hry

API webových služeb:

- Google Maps, Google Search

3.1 Charakteristika

Dobře navržené API

Je převážně napsáno pro uživatele, a proto je důležité, aby ho dokázalo zaujmout a udržet. Je snadné se ho naučit, je intuitivní, regulární a dobře čitelné i bez dokumentace. Nemělo by nechat uživatele dělat nekvalitní nebo dokonce chybné věci. Mělo by tlačit uživatele tak, aby ho používal správně, jak bylo zamýšleno. Využití takového API poté přináší snadné porozumění klientského kódu, snižuje složitost a zvyšuje udržovatelnost.

Žádné API by se nemělo snažit umět vše, ale mělo by být schopno uspokojit požadavky. Vždy je třeba dbát na čistotu a jednoduchost návrhu. Pokud bude úspěšné, bude růst potřeba ho

rozšiřovat, proto je důležité ho trefit napoprvé. Špatné věci lze těžko vzít zpět, když už je používáno byť jediným uživatelem. Musí mít snadnou rozšiřitelnost, což neznamená implementaci samotnout, ale pouze a jednoduše umožnění nové implementace. To už ale zabíháme spíše do zásad dobrého programování.

Špatně navržené API

Můžeme poznat ve dvou případech. V prvním, podle ohlasu uživatelů. Je to nejrychlejší zpětná vazba a může přijít užitečný požadavek na specifikaci, který jsme si dříve nemuseli ovědomit. Ideální otestovat před distribucí oficiální verze (např. jako beta verze). Poté je už na změny pozdě. Zpětná vazba a následný požadavek nám mohou taktéž napovědět, zda jsme API navrhli dostatečně rozšiřitelné a snadno upravitelné. Jinak se výsledek odráží jak na pověsti firmy, tak i na nákladech ve formě údržby a podpory špatného rozhraní.

Druhý je odstup času. Zda existuje, zda je vývoj ještě podporován, jaká je režie a jaký je ohlas.

Modifikátory

Můžeme se zamyslet nad dalším úhlem pohledu. Ten vychází z knížky „*Praktický API Design*“ od Jaroslava Tulacha, který čerpá ze svých dlouholetých zkušeností z návrhu NetBeans API. Zaměřuje se na návrh trochu z vyššího hlediska. Zmiňuje se o tom, co o metodě řeknou její modifikátory (public, final, abstrakt, atd.) [5]. Máme zde vyzdviženy tři kombinace:

`public final`
- metoda je k tomu, aby ji někdo volal

`protected abstract`
- aby ji někdo naimplementoval

`protected final`
- metoda má být volána podtřídami

Ostatním kombinacím modifikátorů bychom se měli spíše vyvarovat, neboť umožnujeme uživateli používat API jinak, než je zamýšleno. Na závěr bych ještě vzpomenul tvrzení, že krása kódu je sice užitečná věc, ale v podstatě neužitečná [5]. Z pohledu programového je to sice pravda, ale musíme brát zřetel na toho, kdo s API pracuje – uživatel, který může být i programátor. Proto bych doporučoval zlatou střední cestu. Otázka implementace je ale věc

druhá, zde upřednostňujeme samozřejmě uživatele od programátora, který rozhranní vyvíjí.

Životní cyklus

API je neustále ve vývoji. Může vznitknout *spontánně*, tak že někdo vyvine nějaký program, část se zalíbí jinému, začne ji používat, bude ji chtít upravit a rozšířit. Časem se toto stabilizuje a vznikne API [4].

Také často vzniká při *návrhu procesu*, kde je potřeba stabilního propojení mezi subsystémy. A v závislosti na prostředcích firmy (zdroje, čas a peníze) se může vyvinout a stabilizovat.

Jednotlivá stádia můžeme označit jako:

`private, friend`

- typický počáteční stav spontánního vývoje

`under development, stable, offical`

- řízený návrh

`deprecated`

- zastaralé, nepodporované

3.2 Obecné principy

Očekávání uživatele se těžko odhaduje a navíc každý uživatel může mít své požadavky, které jsou konfliktní s požadavky jiného. Proto je vhodné nesnažit se uživatele překvapit a implementovat širší škálu možností, ale raději ponechat pár základních, jasně definovaných. Samozřejmě záleží na účelu a pro koho je rozhraní navrženo.

Ze základu dobrých programovacích principů vychází i vytváření oddělených modulů. Metody by měly být jasně pochopitelné a na první pohled je dobré vědět, co daná metoda dělá. Vyvarovat se složitým, navzájem propleteným kódům a raději rozdělit takovýto kód do několika čistějších metod. Ty pak zahrnout do modulů. Když nelze metoda pojmenovat, něco je špatně. Používejte stejná slova pro stejné věci.

Pamatujte, že v budoucnu lze do API přidávat, nikoliv odebírat. Implementační detaily jsou matoucí a omezují možnost změny. Pozor na příliš detailní specifikaci metod. Nechte viditelné jen opravdu to, co chcete a co je nutné. Usnadňujete tím lepší pochopení, testování a ladění.

Dobře napsaná dokumentace znamená, že máme kvalitně zpracované návody, a také FAQ

(Frequently Asked Questions, často kladené dotazy). Při dobře zpracovaném rozhraní je i práce na dokumentaci snažší a uživatel rychleji chápe. Naopak se vyvarujte rozsáhlým popisům jednotlivých prvků rozhraní, jednoznačné prvky je dokonce na škodu popisovat. U nich vychází význam ze svých názvů. Dokumentujte v malém měřítku také vzájemné vazby, vlastnictví či vedlejší efekty. Nenechte uživatele hádat, co daná funkce dělá a už vůbec nedovolte, aby byl nucen se dívat do zdrojového kódu.

3.3 Proces návrhu

Proces návrhu API se podobá v mnoha směrech procesu návrhu každého programu. I tady platí určité rady a zásady [4].

Požadavky

Uživatel často neví, co chce a snaží se místo toho navrhovat řešení, tedy *jak* místo *co*. A cílem je samozřejmě se dopracovat ke skutečným požadavkům. Ze začátku stačí vytvořit kratší specifikaci, kterou lze snadno v případě potřeby měnit. Tuto specifikaci proberte s co největším počtem lidí a naslouchejte jejich připomínkám. Když budete mít základní stavební kameny, můžete dále dopracovat detaily. Analytik dokáže ze zkušenosti navést uživatele a dopracovat se k otázkám, které předtím neřešil nebo si myslel, že nejsou podstatné. V takových případech (a nejen v nich) je užitečné využít softwarového inženýrství a znázornit situace pomocí diagramů.

Zobecnění

Platí zásada nesnažit se naprogramovat či připravit vše, nechejte jen ty metody, které se opravdu využívají. Snažte se o střízlivou univerzálnost, ale zároveň pozor na přílišné zobecňování, které může vést k nekonzistenci.

Testovací plán

Hned při definování základních požadavků pracujte na testovacím plánu. Dle specifikace definujte akceptační kritéria, která bývají schvalována zadavatelem. Detaily můžete dořešit pozdějit, ale nezapomeňte na ně při finalizaci. Testy budou užitečné i při dalším rozšíření. Části

testů nebo postupy můžete použít i pro jiný vývoj.

Konzistence

Určete jednoho člověka zodpovědného za výsledný návrh. Vzhledem k množství vstupů je lépe schopen udržet konzistenci. Málokdy je rozhraní tak veliké, aby to jeden člověk nevzádl. Buďte realisté a počítejte s chybami.

3.4 Souhrn

Vždy je důležité si říci, co od API očekáváme, jeho účel a pro koho je určeno.

4. Kapitola : Analýza

Popisuje stávající stav, požadavky, omezení a možné rizika.

4.1 Stávající stav

Stav firemních knihoven pro tvorbu grafů je nevyhovující. Existuje přibližně 25 typů grafů včetně nejrůznějších jejích modifikací. Přibližně polovina z nich je zastaralá a nepočítá se dále s jejich použitím, ale nechávají se v záloze pro budoucí úpravu a automatizaci. Druhá polovina je aktivně používána a dále rozvíjena do současných výstupů. Současný způsob generování grafů je z části podobný jako navrhované nové řešení, neboť vychází ze stejného vstupního požadavku. A to je vstupní textový soubor s hodnotami a základními parametry ve formátu JSON (blíže uveden dále).

Problém spočívá ve struktuře a hierarchii současné knihovny. Ačkoliv byla původně navržena s jasně definovanými parametry a způsobem volání, postupným rozšiřováním a změnami požadavků na grafickou podobu grafů se stala velmi nepřehlednou a systém nastavení parametrů grafů byl přepisován na několika místech. Důsledkem toho byla nefunkčnost některých parametrů či jejich zkreslení. Při implementaci nových částí byla složitá orientace v kódu a často docházelo k nepochopení některých metod, a také k redundancím.

4.2 Požadavky

Jak už bylo jednou zmíněno vstupem a nositelem dat pro grafické knihovny je textový soubor ve formátu JSON. Zde si dovolím odbočku a popíši tento formát podrobněji.

JSON

JSON (JavaScript Object Notation) - objektový zápis. Je to datový formát pro výměnnu a uchování dat, nezávislý na počítačové platformě. Lehce čitelný, univerzální a podporován mnoha programovacími jazyky. Vstupem je libovolná organizovaná datová struktura (číslo, řetězec, boolean, objekt anebo z nich složené pole). Výstupem je vždy řetězec. Složitost

hierarchie vstupní části není teoreticky nijak omezena. Oproti XML má menší datovou náročnost [6].

```
{
      "nameString" : "valueString",
      "nameNumber" : 123,
      "nameBoolean" : true,
      "nameObject" : {
            "nameSubString" : "valueString",
            "nameSubObject" : {
                  "nameNumber1" : 1,
                  "nameNumber2" : 100
            },
            "nameSubArrayNumber" : [ 0, 1, 2, 3 ],
            "nameSubArrayObject" : [ {
                        "nameString1" : "string1",
                        "nameBoolean1" : true
                  }, {
                        " nameString2" : "string2",
                        " nameBoolean2" : false
                  }
            ]
      }
}
```

Má jednoduché pravidla strukturování a zanořování, a jak již bylo zmíněno je podporován mnoha programovacími jazyky, tudíž např. přenos v jazyce java do objektové struktury je velmi snadný a umožňuje to hned několik knihoven. Jedna z prvních je `java.org.json` a zajímá nás hlavně třída `JSONObject`, ta bohužel spíše zaostala a předčil ji `com.google.gson.Gson`, který je snadnější na použití a taktéž má lépe vyřešenou (chybovou) zpětnou vazbu.

Existují ale i příbuzné formáty, které jsem měl možnost vyzkoušet dva roky zpět. Mimo xml je jeden velmi podobný, a to je *YAML* format (Ain't Markup Language). Ten může obsahovat i komentáře. Problém ale často vzniká s odsazováním a prázným prvním řádem, taktéž jeho konverze do javy mi připadala složitá (což už se ale mohlo změnit). Z vlastní zkušenosti se hodí

nejlépe pro definici konstant. A v poslední řadě je nutno ještě zmínit *SDL* (Simple Declarative Language).

Vratmě se ale zpět k požadavkům. Pomocí uvedeného formátu chceme vkládat hlavně data (dataset) spolu s nejpoužívanějšími a často měněnými parametry (dále označovány jako aktivní). Ostatní parametry, které jsou stálé, mají být definovány stranou, nejlépe pomocí konstant. Ty by měly být prozatím uvnitř knihovny. O jejich vytažení mimo knihovnu např. jako soubor ve formátu *YAML*, se prozatím neuvažuje. Z pohledu přenositelnosti je to spíše krok zpět. V případě přenosu stálé konstanty mezi aktivní by docházelo ke zbytečné řežii.

Aktuálně se musí každý vstup zvlášť konvertovat do potřebné podoby požadovaného grafu. Tím opět dochází k redundanci, zbytečné režii a složitosti.

Standardní a jednotná inicializace grafů, stejně tak jejich parametrizace, je dalším požadavkem. V současné hierarchii každý graf vychází z vlastní třídy `DefaultChart`, která obsahuje abstraktní metodu `makeChart()`. Ta má za úkol inicializovat graf, a to také v pořádku provádí, jenže configurace grafu je často měněna a volána z jejích podtříd a dochází k přepisování a špatné nastavitelnosti plotu, navíc je tímto ztížena orientace. Snadná a jednoznačná definovatelnost plotu je hlavním kritériem pro možnost dalšího rozšíření či aktualizace. Pod toto musíme také započíst volání vlastních Rendererů a nastavení upravených os.

Některé datové objekty vznikly pro specifické potřeby rendererů, jiné jsou redundantní s již existujícími objekty z knihovny JFreeChart a nemají užitek. Jejich další použitelnost je nejspíše nulová.

Zachování pokud možno co nepřesnější grafické prezentace (stejnost grafů). Nová verze knihovny JFreeChart 1.0.17 může některé prvky vykreslovat rozdílně oproti dosud používané verzi 1.0.5 a je potřeba toto ověřit. Některé parametry mohou být zastaralé či pozměněny, a tím také změnit výsledný vzhled.

Posledním požadavkem, a také nejdůležitějším je možnost exportu do jiných formátů. Ze základu PNG či JPEG, a dále se uvažuje o SVG či jiných vektorových formátů.

Po shrnutí máme tyto požadavky:

- vstupem je textový soubor ve formátu JSON, který obsahuje jak data, tak aktivní parametry
- defaultní hodnoty
- jednotná konverze dat
- jednoznačná inicializace grafů a jednotné definování (parametrizace plotů)
- redukce a odstranění nepoužívaných či nepotřebných prvků
- stejnost grafů
- export do grafických formátů

4.3 Omezení

Současným i budoucím omezením pro nový návrh by se mohly zdát následující části.

Vstupní formát

Je zatím jediným způsobem jak předat data a vygenerovat graf, bez toho, abychom cokoliv programovali. Vzhledem k tomu, že firma má vlastní programátory, a také soubor testů, je tento způsob dostačující.

Defaultní hodnoty

Způsob, umístění a správa výchozích hodnot parametrů je potřeba důkladně promyslet. Každý graf může mít své a oproti ostatním rozdílné defaultní hodnoty. Naopak některé parametry by měly být společné pro všechny typy grafů a vycházet z jednotného stylu firmy.

API JFreeChart

Nesmíme také opomenout, že nejvíce pracujeme s knihovnou JFreeChart a jsme odkázáni na její API. My využíváme a dědíme z některých jejích částí, které potom rozšiřujeme a upravujeme. Ve výsledku jsme stejně odkázání na předem definovaný způsob inicializování, strukturu datasetů a nastavování parametrů JFreeChart API.

4.4 Možné problémy

Vstupní textový soubor není strukturálně omezen a dokáže znázornit všechny typy hodnot, uzpůsobit je do hierarchie a umět zpracovat zanoření. Jeho konverze do objektů javy je snadná a může být jednotná.

Výchozí hodnoty parametrů je nutno pokud možno zachovat stávající, ale jejich umístění a správa je potřeba přepracovat do jednotné struktury, která bude srozumitelná a na jednom místě. Zde může docházet ke konfliktům při kombinaci specifických parametrů pro každý typ nebo skupinu grafů a společnými parametry. Tato struktura výchozích hodnot bude i v budoucnu neustále procházet úpravami.

Způsob inicializace grafů by měl být zachován jednotný, v současném stavu ale nemusí být dodržován (není vynucen jednoznačný způsob) a je jen na programátorovi jestli bude dodržovat stejné použití metody `makeChart()`, jak její implementaci, tak volání této inicializační metody.

Parametrizace plotů je část, která byla jedním z hlavních důvodů návrhu nového rozhraní, proto je důležité se na tuto oblast zaměřit a nepodcenit ji. Vzhledem k tomu, že konfigurace plotu se může provádět víceméně kdykoliv před samotným exportem, je důležité zachovat soudružnost při dalším rozšiřování grafů. Toto je ale záležitost programátora a zachování konzistence zpracování grafů.

Vzhledem ke stabilnímu vývoji JFreeChart API by nemělo docházet k výraznému rozdílu ve výsledné podobě grafu. Může sice docházet k menším změnám parametrizace, ale spíše k lepšímu než k horšímu.

Knihovna JFreeChart a její utilita ChartUtilities umožňuje celou škálu exportu do různých výstupních grafických formátů a zachování kvality, a proto není důvod k nějakým možným obavám. Navíc lze samotný způsob exportu přeprogramovat pro specifičtější požadavky.

4.5 Souhrn

V současné době nevidím žádné zásadní problémy, které by nějak komplikovaly návrh a provedení nového rozhraní při zachování stejnosti grafů a splnění všech požadavků.

5. Kapitola : Praktické řešení

V této části se budeme zabývat konkrétním návrhem nového rozhraní, řešení a celkové struktury.

5.1 Návrh

Musíme se zamyslet, jaké rozhraní vlastně chceme vytvořit a k jakému účelu bude sloužit. Samozřejmě vycházíme nejprve z požadavků a snažíme se maximálně využít zdravé komponenty stávající knihovny. Musíme si také uvědomit omezení dané knihovnou JFreeChart – tedy že jsme vázání na standardní postup vytváření grafů a jejich exportu. A toto elegantně propojit se vstupním rozhraním, což je v našem případě textový soubor v JSON formátě.

Prvotní průběh návrhu

Nejprve jsem předpokládal, že budu vytvářet určitou upravenou kopii stávajícího JFreeChart API, kde bude výběr používaných funkcionalit a nastavení upravených pro potřeby firemní knihovny. Od této myšlenky jsem záhy upustil, neboť neřešila byť jediný požadavek a vedla pouze směrem k problémům. Nedala by se udržovat, existovalo by mnoho redundancí, s dalšími úpravami by docházelo k nekonzistenci a ve výsledku by to bylo stejné jako doposud.

Dále jsem vycházel ze stávajícího procesu volání jednotlivých tříd grafů a jejich předávání vstupních dat. Snažil jsem se vytvořit určitou hierarchii, která by zajišťovala potřebné služby. Z tohoto důvodu jsem postupoval směrem od shora dolů, tedy od zajištění základních služeb jako jsou hlavně *načtení vstupních dat* a *inicializace* třídy grafu, která by měla funkci pro export. U tohoto návrhu by hlavní abstraktní třída sice potřebné služby zajišťovala, ale všechny bychom museli implementovat v jejích podtřídách. To by ale vedlo k přesnému opaku toho, čeho chceme dosáhnout, a to k jednotnému způsobu předávání dat a inicializace tříd grafů.

5.2 Řešení

Zaměřil jsem se na vstupní část celého procesu, která byla pro konečný návrh klíčová. Zde totiž docházelo ke spouštění celého procesu a uvědomil jsem si zde právě to oddělení dvou částí, které jsou *předání vstupních dat* a *inicializace tříd grafů*.

- načtení vstupních dat
- předání vstupních dat
- inicializace tříd grafů
- export

Každá část procesu má své pravidla a povinosti, které si dále rozebereme.

Načtení vstupních dat

O načtení se stará již jednou zmiňovaná třída `Gson` z balíčku `com.google.gson`. Té předáváme jako parametr cestu k json souboru a třídu reprezentující celkovou objektovou strukturu vstupního souboru `cz.daps.general.dto.DefaultDTO`. Gson překladač populuje data standardní cestou přes getter / setter, které třída `DefaultDTO` samozřejmě obsahuje.

V této tříde máme „pouze“ dva atributy, které reprezentují prvky vstupního json souboru.

```
private String presentation;
private ChartConfigurationDTO chartConfiguration;
```

```
"presentation" : "CHART",
"chartConfiguration" : {
   ...
}
```

Zde je důležitá třída `ChartConfigurationDTO` která nám reprezentuje hlavní strukturu vstupních dat a dále ji předáváme tzv. *factory* třídám. Všechny DTO třídy jsou z balíčku `cz.daps.general.dto` jako (Data Transfer Object) a přesně typově kopírují jednotlivé

objekty v JSON formátu.

Důležité pro naše účely je, aby všechny atributy těchto DTO tříd byly objekty a ne primitivními typy. A to z důvodu zajištění vlastních defaultních hodnot. Podrobněji popsáno níže v části Konvertor.

Předání vstupních dat

Pro předání vstupních dat správné třídě grafu slouží základní třída `ChartFactory` z balíčku `cz.daps.general.factory` se statickou metodou `makeChart()`. Ta volá na základě vstupního parametru konstruktor konkrétní *factory* třídy a předává jí vstupní data reprezentované již zmíněnou třídou `ChartConfigurationDTO`. Všechny tyto třídy implementují základní rozhraní `IChart`, kde je jen jedna metoda `createChart()`, která volá až inicializaci konkrétní třídy grafu a předává jí konvertované vstupní parametry do lépe použitelné podoby.

```
public class TachometerChartFactory implements IChart {
    private DTOConverter DTOConverter;

    public TachometerChartFactory(ChartConfigurationDTO configurationDTO) {
        DTOConverter = new DTOConverter(configurationDTO);
    }

    @Override
    public byte[] createChart() throws ChartException {
        return createTachometerChart().saveAs();
    }

    private TachometerChart createTachometerChart() throws ChartException {
        TachometerChart tachometerChart = new TachometerChart(
                DTOConverter.getBaseConfiguration(),
                DTOConverter.getPlotConfiguration(),
                DTOConverter.getDatasetChart());
        return tachometerChart;
    }
}
```

Třída `TachometerChart` zde reprezentuje již třídu grafů z vlastní knihovny grafů.

Factory třídy mají ještě jednu úlohu, a to rozdělit data určená pro základní nastavení grafu, data pro konfiguraci plotu a dataset.

Konvertor

Je určen pouze pro konverzi DTO tříd, které mají kvůli vstupnímu textovému souboru určitou specifikou formu, někdy i možnost zapsání hodnot více způsoby. Je tedy nutné je rozpoznat a převést do použitelných objektů, které využívají třídy grafů. Tyto objektové třídy mají ze základu nastaveny defaultní parametry, na které je třeba dbát zřetel a přepisovat je jen v případě, že jsme skutečně nějakou hodnotu ze vstupního souboru získali. Proto je důležité, aby DTO třídy měly atributy jako objekty a ne primitivní typy, jinak bychom např. nemohli rozeznat, zda si přejeme mít nastaven parametr startValue = 0 nebo ne.

Stačí nám pouze jedna třída `cz.daps.general.converters.DTOConverter`, která má tři základní *public* metody a *privátní* metody, které konvertují jednotlivé DTO subobjekty na použitelné objekty z balíčku `cz.daps.general.api`. Jako příklad můžeme uvést třídu `TextChart`, která obsahuje informace ohledně textu (viditelnost, text, font, barva písma, barva pozadí, odsazení).

```
getBaseConfiguration()
getPlotConfiguration()
getDataset()
```

Zde můžeme poprvé vidět základní oddělení parametrů a dat (podrobněji dále). A pro úplnost si ještě uvedeme příklad, kde ve vstupním json souboru můžeme zapsat některé hodnoty více způsoby a jejich konverzi. Kvůli slohovému omezení je výsledná podoba metody upravena.

```
"baseConfigurationChart" : {
        "padding" : [0, 20, 0, 10] anebo "padding" : [10]
}
```

```
private RectangleInsets convertPadding(Double[] padding)
throws ChartException {
    if (padding.length == 4) {
        return new RectangleInsets(
              padding[0], padding[1], padding[2], padding[3]);
    } else if (padding.length == 1) {
        return new RectangleInsets(
              padding[0], padding[0], padding[0], padding[0]);
    } else {
        LoggerMessage lm = new LoggerMessage(
        "Unsupported padding format. Correct is Double[top, left, bottom, right]
        or Double[commonValue]")
              .addValue("actual: ", padding);
        throw new ChartException(lm.toString());
    }
}
```

5.3 Třídy grafů

Třídy grafů z balíčku `cz.daps.library.charts` jsou hlavní částí vůbec a řídí samotné vytváření grafů z knihovny JFreeChart a naplnění plotů a datasetů s použitím JFreeChart API. Také zde dochází k připojení námi upravených rendererů, os apod. Každá třída přijímá pomocí konstruktoru vstupní data rozdělená do skupin *Base*, *Plot* a *Dataset*. Tímto se zajistí vynucenost pro vstupní data.

V tomto kontruktoru také dochází k inicializaci a konfiguraci vlastního JFreeChart grafu. Opět pro slohové omezení je výsledná podoba ukázky upravena.

```
public class TachometerChart extends BaseChart {
    private JFreeChart chart;

    public TachometerChart(
        BaseConfigurationChart baseConfiguration,
        PlotConfigurationChart plotConfiguration,
        DatasetChart dataset) {
            chart = new JFreeChart(createPlot(plotConfiguration, dataset));
            chart = this.setBaseConfiguration(chart, baseConfiguration);
            if (!baseConfiguration.getLegendVisible()) { chart.removeLegend(); }
    }
```

```
    private CustomMeterPlot createPlot(
        PlotConfigurationChart plotConfiguration,
        DatasetChart dataset) {
        ...
    }
    ...
}
```

BaseChart

Všimněme si, že každá třída grafů dědí ze třídy `BaseChart`, ta má metodu `setBaseConfiguration()`. Zde se dostáváme k základní myšlence rozdělit celou konfiguraci na společnou skupinu parametrů pro všechny grafy, jako jsou legenda, titulek, odsazení, barva pozadí, ohraničení, vyhlazování a na konfiguraci plotů. Zde je ale jedno úskalí dědičnosti, které nám umožňuje tuto děděnou metodu přepisovat i v třídách grafů, čemuž se chceme právě vyvarovat pro udržení jednotné konfigurace.

Základní nastavení obsahuje taktéž klíčové informace pro export, a to jsou rozměry grafu a výstupní formát (PNG, JPEG). Tyto údaje jsou použity až při samotném exportu a pro inicializaci či nastavení grafu nejsou potřeba.

Důvod rozdělení je jednoduchý, a to je přehlednost, která při úpravách a nastavení výchozích hodnot je velmi důležitá.

V této třídě je také implementována další společná metoda `saveAs()`, která exportuje nakonfigurovanou instanci grafu do výstupního grafického formátu a používá právě již zmíněné parametry (výška, šířka, formát). Pro univerzálnost vrací byte[].

5.4 Konstanty

Výchozí hodnoty

Označme si je i jako tzv. *stálé* konstanty, které aktivně neměníme. Hierarchie a použití výchozích definovaných hodnot není na první pohled tak jednoduchá jak by se dalo předpokládat. Musíme si uvědomit, že nám zde existují jak společné hodnoty, tak hodnoty specifické pro každý typ a variantu grafu. Pokud z tohoto vycházíme, můžeme celkem jednoduše vytvořit pro každou třídu grafů vlastní soubor konstant, který zajistí specifické

výchozí nastavení. Pro to, abychom měli k dispozici i společné konstanty, budou tyto specifické soubory konstant dědit právě ze společné třídy konstant. Pokud bychom chtěli upravit některou výchozí hodnotu pro určitou třídu grafu, není nic jednoduššího než ji jednoduše přepsat ve specifické třídě konstant. Konstanty jsou v balíčku `cz.daps.library.constants`.

CommonChartConstants
- | *Bar2DOverlayChartConstants*
- | *Bar2DVerticalTextChartConstants*
- | *BaseChartConstants*
- | *PieChartConstants*
- | *TachometerChartConstants*
- | *XYPaletteChartConstants*

Použití těchto konstant je relativně dobře zařazeno přímo do použitelných objektů z balíčku `cz.daps.general.api`.

```
import static cz.daps.library.constants.CommonChartConstants.Border.*;

public class BorderChart {
    private boolean visible = VISIBLE;
    private float size = SIZE;
    private Stroke stroke = STROKE;
    private Paint color = COLOR;

    // construktors
    // getters / setters
}
```

A konvertor nám zajistí, že aktivními konstantami přepíše výchozí a u těch, které nejsou použité, je nechá. Opět pro slohové omezení je upraven vzhled metody.

```
private BorderChart convertBorder(BorderDTO borderDTO) {
   if (borderDTO != null) {
         BorderChart border = new BorderChart();
         // zde již máme zajištěny výchozí hodnoty
```

```
            if (borderDTO.getVisible() != null) {
                  border.setVisible(borderDTO.getVisible());
            }
            if (borderDTO.getSize() != null) {
                  border.setSize(borderDTO.getSize());
            }
            if (borderDTO.getStroke() != null) {
                  border.setStroke(borderDTO.getStroke());
            }
            if (borderDTO.getColor() != null) {
                  border.setColor(convertColor(borderDTO.getColor()));
            }
            return border;
      } else {
            return null;
      }
}
```

Existuje zde ale problém, který není vyřešen. Je sice pravda že většina výchozích hodnot pro objekty z balíčku `cz.daps.general.api` je použita ze společné `CommonChartConstants`, ale ne vždy tomu tak bude. Pokud bychom měli určitou společnou konstantu pro více typů grafů (např. barva pozadí plotu) a každý typ grafu si ji definoval speciálně sám na vlastní hodnotu a tuto konstantu bychom použili jako aktivní konstantu, vznikne problém, kterou výchozí hodnotu vlastně pro daný objekt nastavit.

```
CommonChartConstants (Plot.backgroundColor = WHITE)
   | PieChartConstants (Plot.backgroundColor = RED)
   | TachometerChartConstants (Plot.backgroundColor = BLUE)
```

Ve vstupním json souboru bychom tento atribut mohli i nemuseli použít. A právě v případě, když ho nepoužijeme, tak nemůžeme zajistit, že výchozí barva pozadí plotu bude pro PieChart červená a pro tachometer modrá.

```
public class PlotConfigurationChart {
   private Paint backgroundColor =
                              CommonChartConstants.PLOT_BACKGROUND_COLOR;
}
```

Nevýhoda možných řešení je, že budeme vždy potřebovat nějaký identifikátor, který nám řekne, o který typ grafu se jedná.

A u další myšlenky přesunutí definice výchozích hodnot do jiné vrstvy nám vzniká problém s rozlišením, kdy jsme atribut ve vstupním json souboru nastavili a kdy ne. Potřeba dále rozpracovat a dořešit.

Aktivní konstanty

Aktivními hodnotami rozumíme ty, které jsou zároveň používány ze vstupního JSON souboru. Také mají nastaveny své výchozí (stálé) hodnoty, a pokud nejsou použity ve vstupním souboru, zůstávají jim výchozí nastavení.

Změna stálé konstanty na aktivní není sice primitivní a vyžaduje určitou režii, ale také není nijak složitá. Důležité je, aby byl zajištěn příslušný atribut v DTO třídách a konvertor uměl na tento atribut reagovat a správně ho předat do objektových tříd api. Vzhledem k tomu, že v současné době `DTOConvertor` dokáže překonvertovat všechny použitelné typy je v tomto ohledu jen malá řežie.

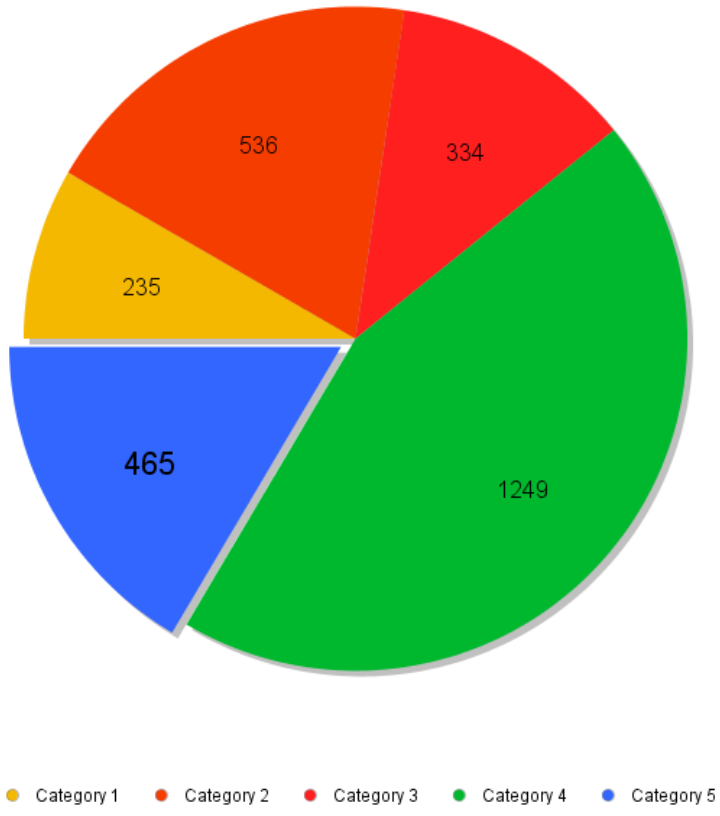
6. Kapitola : Prezentace a testy

6.1 Ukázky

Vstupní soubory jsou uvedeny na cd v elektronické podobě ve složce `resources/json`.

PieChart

Vstupní soubor: `chart_pie.json`
Třída grafu: `cz.daps.library.charts.PieChart`

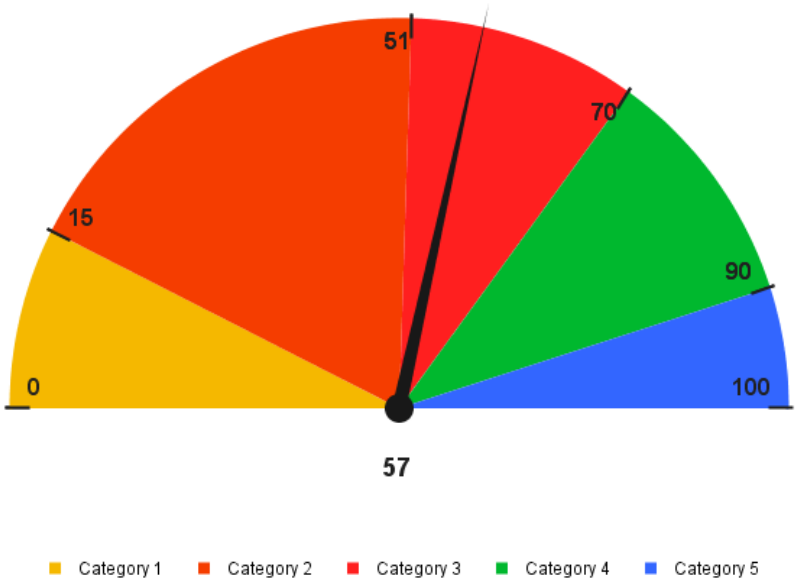


Obrázek 6.1: PieChart

TachometerChart

Vstupní soubor: `chart_tachometer.json`

Třída grafu: `cz.daps.library.charts.TachometerChart`

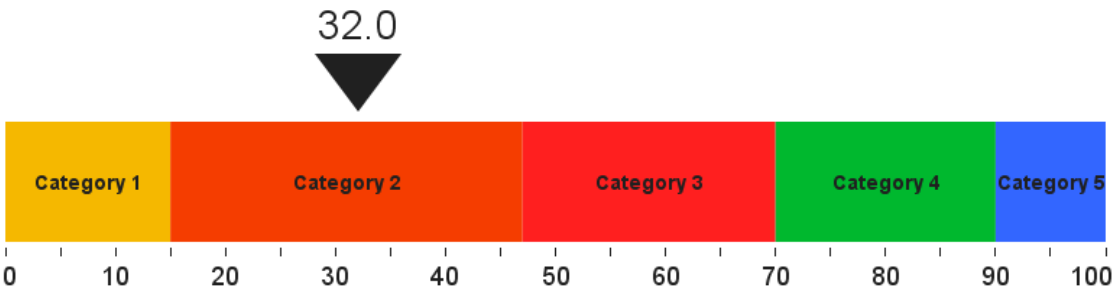


Obrázek 6.2: TachometerChart

XYPaletteChart

Vstupní soubor: `chart_xypalette.json`

Třída grafu: `cz.daps.library.charts.XYPaletteChart`

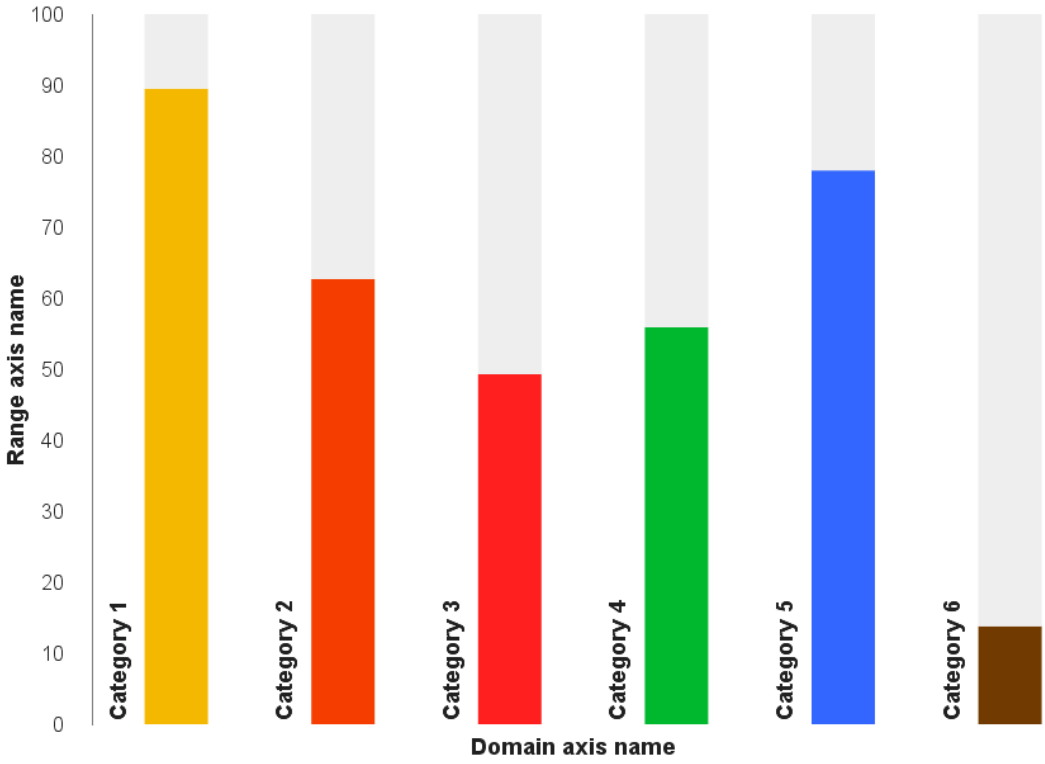


Obrázek 6.3: XYPaletteChart

Bar2DVerticalTextChart

Vstupní soubor: `chart_bar2Dvertical.json`

Třída grafu: `cz.daps.library.charts.Bar2DVerticalTextChart`

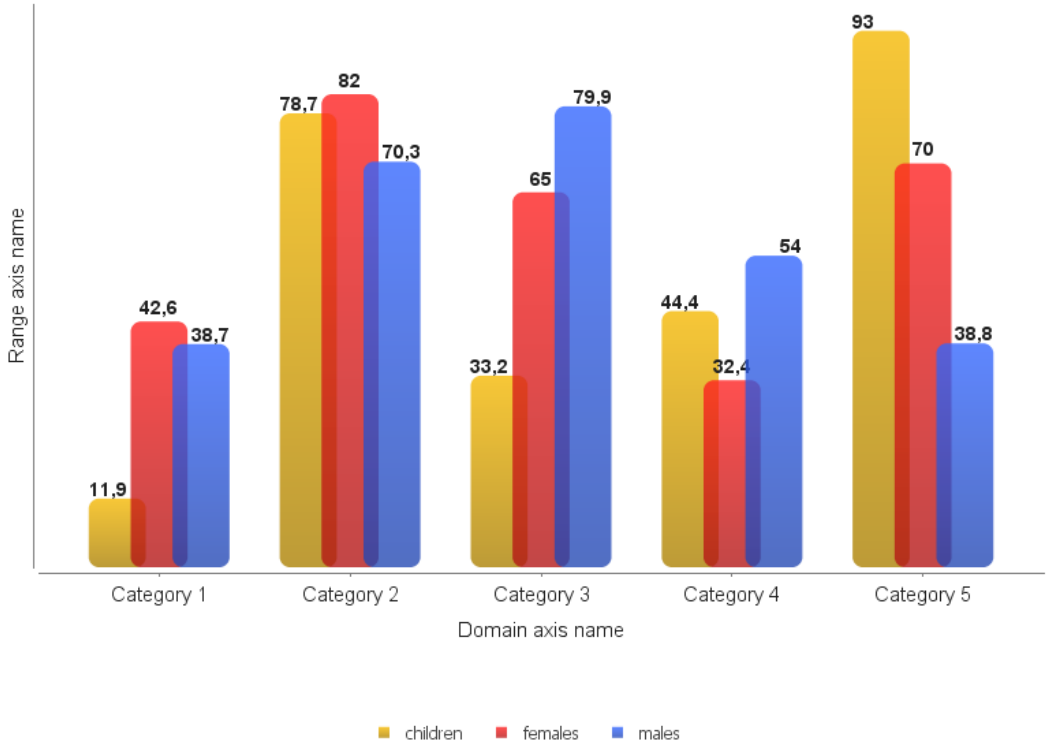


Obrázek 6.4: Bar2D VerticalTextChart

Bar2DOverlayChart

Vstupní soubor: `chart_bar2DOverlay.json`

Třída grafu: `cz.daps.library.charts.Bar2DOverlayChart`



Obrázek 6.5: Bar2D OverlayChart

6.2 Testy

Součástní projektu jsou samozřejmě i testy. Vzhledem k elegantnímu a jednoduchému návrhu rozhraní a počtu public metod a konstruktorů jsou třídy snadno otestovatelné.

Pro zajištění funkčnosti postačí jako základ testy tříd grafů a k nim integrační testy, které ověří celý proces. Testy budou dále rozvíjeny a rozšiřovány. V současné době, ale příliš podrobné testy nejsou zapotřebí, neboť rozhraní je stále ve vývoji.

Pro správu testů využíváme balíček `org.junit`, ten je již velmi podobný dalšímu testovacímu balíčku `org.testng`, který je využíván ve firmě.

Testy tříd grafů

Testy jsou postaveny na jednoduchém principu porovnání CRC součtů generovaných grafů s externími obrázky. Ve zkratce se CRC kontrolní součet používá pro ověření, zda kontrolovaný objekt se nezměnil či nedošlo k poškození. V našem případě tohoto využíváme pro ověření bytového pole vzorového obrázku, který byl vytvořen např. při vývoji a obrázku, který byl vygenerován právě při provádění testu. Pokud jsou CRC součty totožné, obrázek musí být nuťně stejný. Dokáže odhalit rozdíl i v jediném pixelu. V javě využíváme třídu `CRC32` z balíčku `java.util.zip` a její metodu `update()`, která nám převede vstupní bytové pole do bytového pole CRC součtu.

Jednoduchý postup testování:

1. přímé vytvoření a naplnění datasetu
2. nastavení plotu
3. inicializace grafu
4. vygenerování grafu
5. načtení zdrojového souboru pro porovnání
6. převod do pole bytů
7. porovnání

Integrační testy

Integrační test má za úkol otestovat celý proces od začátku do konce se vším všudy, tedy často i s databází pokud to proces vyžaduje. Zde se liší integrační testy oproti testům tříd grafů v první části, a to ve vytvoření a naplnění datasetu. Ten plníme právě za pomocí našeho nového rozhraní a vstupem dat je externího json soubor.

```
@Test
public void testBar2DOverlayChartGraph() throws Exception {
   byte[] actual = Run.generateGraph(FILE_PATH + JSON_NAME_Bar2DOverlay);
   byte[] expected = getExpected(IMG_NAME_Bar2DOverlay);
   assertGraph(actual, expected);
}
```

```
private byte[] getExpected(String imgName) throws IOException {
    InputStream expectedIS =
     Bar2DOverlayChart.class.getResourceAsStream(RESOURCE + imgName + PNG);
    return TestUtils.getBytesFromInputStream(expectedIS);
}

public void assertGraph(byte[] actual,byte[] expected) throws IOException {
    CRC32 crcAct = new CRC32();
    CRC32 crcExp = new CRC32();
    crcAct.update(actual);
    crcExp.update(expected);
    Assert.assertEquals(crcAct.getValue(), crcExp.getValue());
}
```

6.3 Možnosti rozšíření

Možnosti rozšíření nového rozhraní jsou veliké. Jak už ve způsobu vkládání a formě vstupních dat, nastavování parametrů, tak i výstupů do vektorových formátů.

Nadále se pokračuje s přidáváním nových grafů a rendererů a jejich úpravami. Při zachování navrhnutého konceptu může být i po pár letech kvalitním rozhraním pro potřeby firmy.

7. Kapitola : Závěr

Cílem bylo přepracovat stávající rozhraní pro generování grafů pro potřeby firmy DAP Services do formy, která je jednoznačná, čitelná a málo závislá na režii. Dále přepracovat stávající firemní knihovnu upravených grafů, pročistit ji a sjednotit strukturu a konfiguraci za použití nejnovější verze JFreeChart API. Dále vytvořit sadu integračních testů a jednoduché webové rozhraní pro načítání vstupních dat ve formě json.

Vycházel jsem hlavně z požadavků na vstup a z možností inicializace grafů JFreeChart API, které byly klíčem ke struktuře výsledného rozhraní.

Podařilo se vytvořit jednoduché a částečně i jednoznačné rozhraní pro generování grafů. Je lehce čitelné a neskrývá žádnou složitou strukturu, která by mohla v budoucnu být přítěží. Má velké možnosti rozšíření. Na vývoji i rozšiřování o nové grafy se nadále pracuje.

Problémy vznikaly hned na začátku při definování požadavků a uvědomění si určitých procesů, které vycházely z API JFreeChart. Vznikaly zbytečné vazby a odchylování se od myšlenky sjednocení určitých částí.

Z mého pohledu se jedná o kvalitní návrh.

8. Seznam odborné literatury

[1] Oficiální stránky JFreeChart [online]. 2005-2013 [cit. 2014-05-03]. Dostupné z: http://www.jfree.org/jfreechart.

[2] Knihovna JFreeChart vyrobí z nudných čísel hezké grafy, František Bártík [online]. 2013-08-01 [cit. 2014-05-05]. Dostupné z: http://www.linuxexpres.cz/software/knihovna-jfreechart-vyrobi-z-nudnych-cisel-hezke-grafy.

[3] Object refinery, images [online]. 2014 [cit. 2014-05-03]. Dostupné z: http://www.object-refinery.com/orsoncharts/.

[4] Doporučené postupy v programování, Lubomír Bulej, David Majda [online]. 2013-05-09 [cit. 2014-05-04]. Dostupné z: http://d3s.mff.cuni.cz/teaching/programming_practices/.

[5] Java crumbs [online]. 2009-04-11 [cit. 2014-05-04]. Dostupné z: http://blog.krecan.net/2009/04/11/prakticky-api-design/.

[6] JSON wikipedie [online]. 2013-10-09 [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/JavaScript_Object_Notation.

9. Přílohy

9.1 Obsah CD

- Src
- Javadoc
- Resources