Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Department of Electronics Technology

Computer based modeling of the Hungarian macroeconomic processes

MSc Thesis work

Author: Gergely Kovács

Consultant: Dr. Béla Szikora

Date: Dec 13. 2013.

Course code: BMEVIETM387

Table of contents

HALLGATÓI NYILATKOZAT

Alulírott Kovács Gergely, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2013. 12. 13.

…….

Kovács Gergely

Abstract

The following thesis, concluded in the Business IT MSc program of the Budapest University of Technology and Economics, introduces a macroeconomic analysis framework, in which anyone can browse actual data, view forecasts, and simulate the effects of changing macroeconomic parameters. The system is available at: www.macrosim.hu

The thesis, after a brief introduction, first describes the software development tools which were used during the implementation.

Next, the database model will be explained. The chapter will provide the necessary background on how the information is stored and accessed. Also, I will explain the reasons of this design.

After, I will show how the data was put into the database, straight from the data sources, namely the Statistical Office and the National Bank of Hungary. Furthermore, this chapter introduces the administrative tools, needed to handle the data through the GUI (Graphical User Interface).

Chapter 5 describes the object oriented environment of this system. I will explain what kinds of objects were created, how they are instantiated and how they communicate in order to draw charts.

Chapter 6 explains how charts are drawn. The explanation also includes the improvements I have made to make the GUI more user-friendly.

Chapter 7 describes the forecast functions. The first part is about creating and storing them, while the second part is about specific forecast functions. This chapter is the essence of this thesis, as all the rest was created on the sole purpose to be able to create forecasts in the way, which is described here.

Finally, I will write a few words about the GUI, how it looks like and how it works. A lot of development had to be done, in order to access functions through the interface, I will also name and explain these.

1. Introduction

Since I was a kid, I was looking the financial data every week at the last pages of a newspaper called HVG, a Hungarian newspaper similar to The Economist. Thanks to my parents, who have bought various kinds of newspapers, I had the opportunity to develop and satisfy my data browsing needs. Fast forward to recent years, this desire just grew further, driving me towards analysis, to financial analysis in particular. After writing a few articles, evaluating a few business plans, I have decided to create a system which helps me analyzing various data.

Whenever I have to make an analysis, mainly in macroeconomics, I do not have the tools to set parameters comfortably, or to access the needed data quickly and easily. Moreover, I found a growing interest for reliable articles in economics in Hungary, explaining what is happening and for what reasons.

All these factors led me to the conclusion I should develop a program, more likely a framework, which helps anyone to understand basic connections among macroeconomic data. Meanwhile I also fulfill my hunger for in-depth analysis, and my wish to express my capabilities of understanding and implementing such complex systems.

The program I developed, and which I will introduce in this thesis, has multiple objectives. The first is to make simple forecasts on any kind of data sets. These forecasts are based on a wide variety of functions starting from statistical methods, till reverse engineered connections. The second is to show mid-term effects of present changes. Third, I would like to create a framework, what could be used for any complex modeling, regardless of the data itself, not just for macro-economic modeling. Finally, I hope this program could serve the educational and online journalism goals as well, for which, I believe, there is need for in Hungary.

2. Technical introduction

In this chapter I would like to briefly introduce the languages, tools and software packages that I used during the development and implementation. I wanted to make my simulator available on web, hence I chose PHP/MySQL as a basic platform for development. All the rest were chosen to fit this environment.

All the data are stored in MySQL tables, which are processed by PHP scripts. These scripts not just handle the data themselves but the simulations are also implemented in PHP, so are the source code generations for the HTML/Jscript modules.

I did not use any programming framework, like Eclipse, during the work. The reason was, I am not familiar with these environments and I did not want to waste time learning them now. Later, as I bumped into bugs and naming errors all the time, I regret this decision.

Software, libraries and tools

I needed some software packages to make both the implementation and the documentation easier. Without any detailed description of these packages, I would like to briefly introduce them.

MySQL Workbench, Navicat

MySQL Workbench is a free software package which helped me not just creating and managing my database but it also gave me an easy way to edit my data well before I finished the administration interface. Additionally, the software offers comprehensive modeling tools, which helped me a lot in the designing and understanding of the database model. All data model screenshots in this document originate from this software.

I have also used Navicat's database manipulation software, which helped me inserting and updating data on a massive scale, what was not supported by the MySQL Workbench.

PHP/Apache/HTML

The simulator itself is written in PHP language, which also handles the database results and the GUI. The program runs on a dual Xeon server with Linux/Apache. Writing the PHP classes was the main part of my work as I will show later in this document.

Visual Paradigm

The class diagrams were generated by a software called Visual Paradigm, which I found fitting my needs. The software also supports creating UML diagrams. This software would have been able to help in implementation as well, but I did not use it, mainly because I had to write code parallel in other languages, namely in JavaScript, and this program could not handle both at the same time.

Google Chart Tools

To draw charts easily, I chose to use Google's Chart Tools [1] together with JQuery, a widely used JavaScript library, which made GUI programming substantially easier and faster, giving me more time to focus on my real task: designing. During my work, I found many bugs and missing features in Google Chart Tools, which I have posted to the official forum [2].

JQuery UI

As a well-known and widely used free package, JQuery UI helped me programming the Graphical User Interface to make my program easily usable. Also, this package allowed me to develop my own interactive charts.

SPSS

Finally, I used IBM's SPSS to find regression functions among variables. Even though I found many other statistical programs, some of them written in PHP [3], SPSS proved to be the fastest and most fitting for my needs.

3. The database

In this chapter I will introduce the database that I built during my work. The most difficult part was to create a database model which resonates with my program and its objects. As I advanced in the implementation, and new ideas have surfaced, the database had to be adjusted constantly, until it took its final shape, which is presented here.

I will not fully explain every part of the database. For example the database model of forecasts will be introduced in this chapter but elaborated later, in chapter 7. Yet, I wanted to present a comprehensive picture of the database, in order to show how all the data are connected in one model.

My database consists of static and dynamic data. Static data function as a repository, while dynamic data are used by my program.

3.1. Static data

I refer as static data to everything related to statistical (not changing) data, which I want to use or forecast during my work. Among many, these data include the budget balances, GDP, prices, tax rates, etc. I store them in separate data tables, that usually match the structure of the downloadable excel files of the Central Statistical Office and the National Bank of Hungary.

Since I had to write this thesis in two semesters, I had a long summer break, after which I have returned to my work. The update of all the data, roughly 600 columns in 20 tables and 4-5 months of records, took about 4-6 hours only. Creating data tables similar to the official ones, and filling them with the help of Navicat, which can *'copy-paste'* data from excels into MySQL, made this part of the work very easy.

However, it was not always this easy. While creating data tables for prices or GDP data were fast, creating a database table for the budget data took an entire day itself. I will describe this process later, in chapter 4.

3.2. The database model

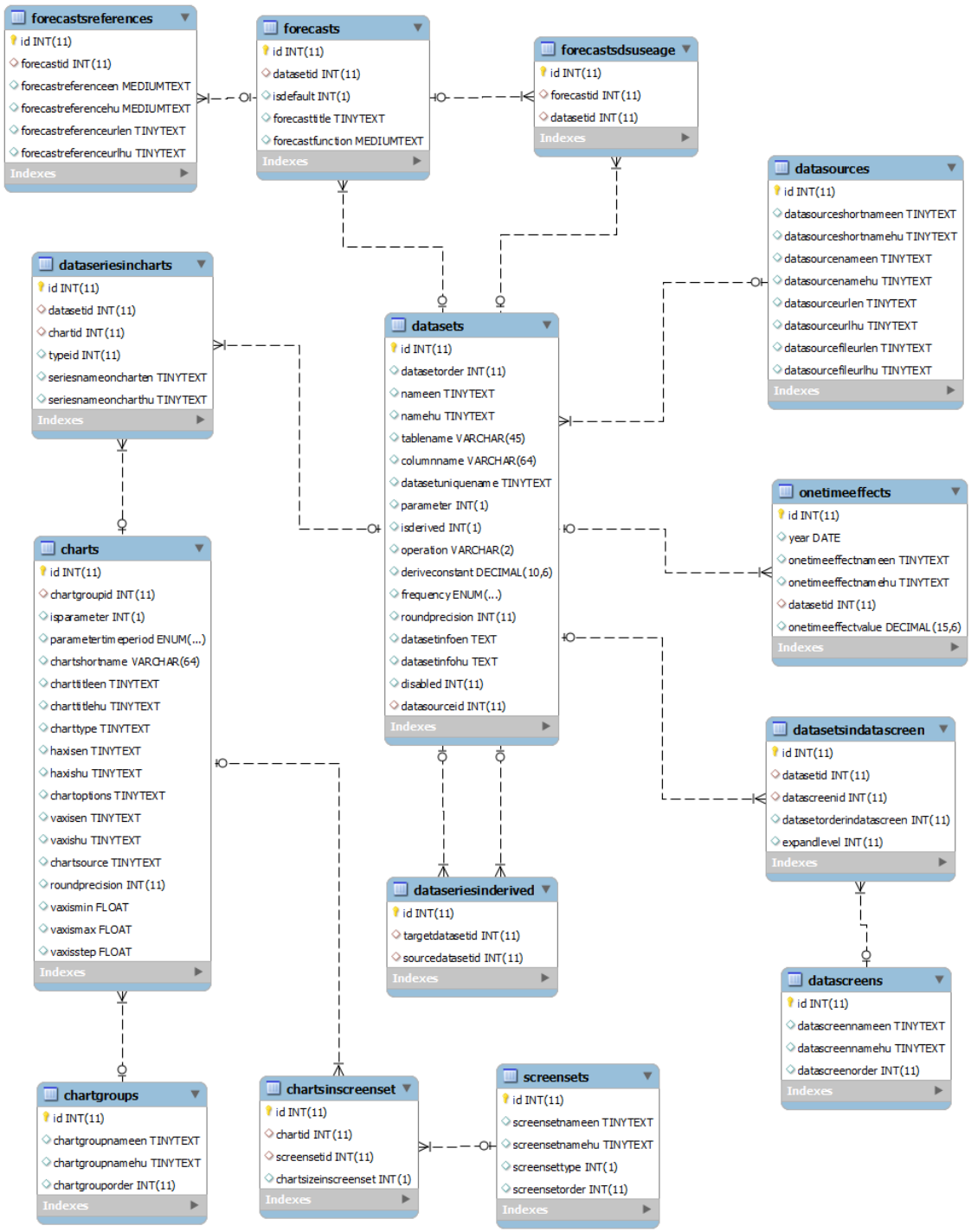

Figure 1 - The database model of the program

As you can see, the central part of the database model is the 'dataset' table. This is also true for the PHP Objects, where the DataSet object serves as the backbone of my system. Once we understand the datasets, as I use them, the rest will be crystal clear.

The tables are connected through foreign keys, generally with the setting ‘on update: cascade’, ‘on delete: restrict’. With these, I can keep my database consistent and avoid accidental deletion of precious information. As usual, the downside of security is, deleting a dataset, or a chart, might be very uncomfortable as I have to delete the corresponding information first.

3.3. Dynamic data

Dynamic data are the changing ones: chart settings, forecasts, GUI screens, etc. Basically all data, that are stored in the data tables shown on the figure above, are considered as dynamic data.

3.3.1. Data Series and Parameters

These two are all the same. I decided not to make distinction among these types of dynamic data. Every data set is considered as a data series (or a set of series), which can be also drawn on a chart, or used as a parameter. This decision proved to be a very easy way to handle both charts and forecast functions later. Making certain parts of these data modifiable, made them useable as parameters as well.

As a rule of thumb, when I write ‘data series’ I usually mean one piece of a 2 columns data series. In case of ‘data sets’ I usually mean a few data series, which are basically populated from one data series. For e.g.: ‘yearly births’ is a data series, where the first column is the year, the second is the number of children born in that year. Whereas forecasted births, together with actual births, form a dataset. In the *DataSet* object, explained in 5.2, these are present at the same time, in one object; resulting, every type of ‘births’ related data is accessible in one instance of the *DataSet* object.

Also, I can use any data series as a parameter, for e.g. fertility rate for births’ forecast function.

The database model for DataSets

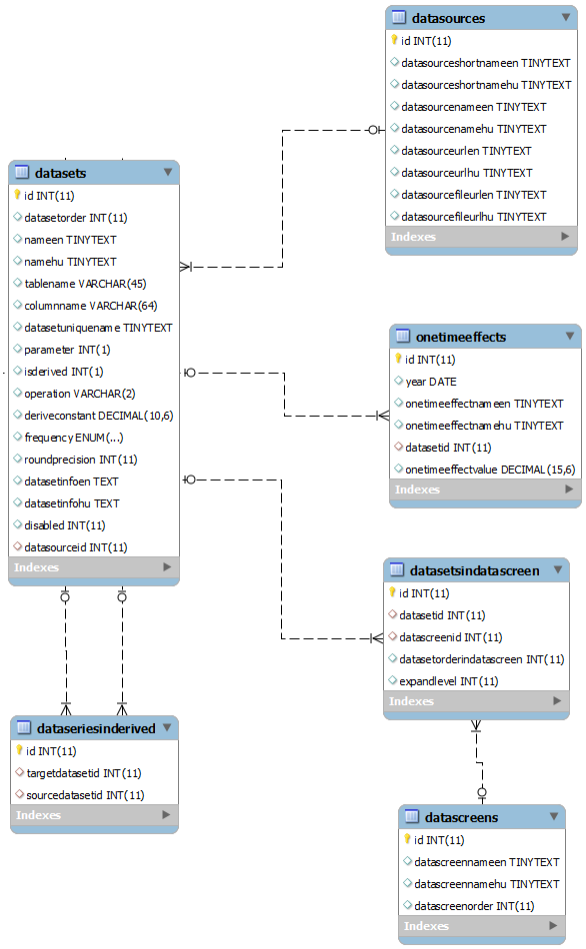

Figure 2 - Database model of DataSets

- *datasources* store the name and urls of the entities, where the data has been acquired from. Such entities are the Statistical Office, or the National Bank of Hungary. Every time, I draw a chart, the '*source*' of the data shown is based on the information stored here.
- *onetimeeffects* store special items, like court decisions, which have altered the raw data, and should be excluded from forecast functions. For example, Hungary had to pay back roughly 140 billion HUF of value added tax to companies in December 2011, which payback has distorted that month's data. Any function, which uses value added tax income as a parameter, should exclude this item.

- *datascreens* are lists of datasets, to help the users understand the structure of the available data. These screens are shown on the GUI, and help navigating among datasets. Every *datascreen* can hold multiple datasets, hence the *datasetsindatascreens* table.
- *datasetsinderived* is a list of datasets being in other datasets, namely in derived data sets. Derived datasets do not use any static data, rather they are calculated from other datasets. For example *unemployment* is a derived dataset, and it is calculated by dividing two datasets: the number of unemployed by the number of economically active. More about derived DataSets can be found in 5.2.

3.3.2. Charts' database model

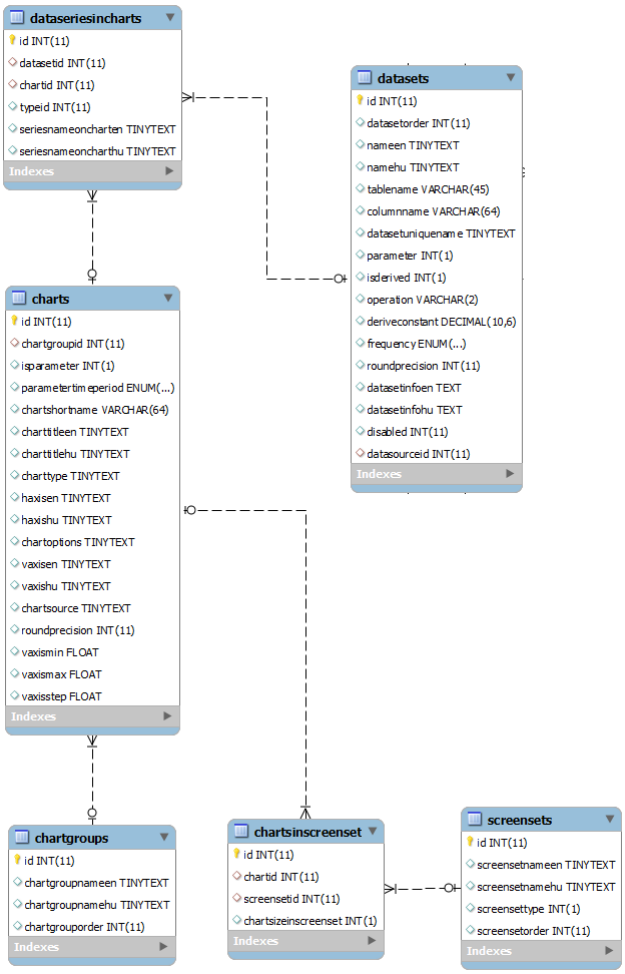

Figure 3 - Database model for Charts

A Chart is nothing else, than a combination of data series, which we want to draw up in the same graphic.

- *dataseriesincharts* is used by every chart as they can contain multiple data series, those which we want to draw up. For example the chart of '*taxes on consumption'* holds seven data series, from *value added tax*, through *excise tax*, till *financial transaction tax*. Through this database table, we can associate multiple data series to one chart.
- *chartgroups* are used to group charts together, to help easier administration.
- *screensets* are similar to *datascreens*, they serve the goal to help users navigate among charts. Every screen set, for example '*Labor´* can contain multiple charts, like employment, economically active, and unemployed. With screen sets, the user can access multiple charts with one click. More about screen sets can be found in chapter 0.

3.3.3. Forecast functions' database model

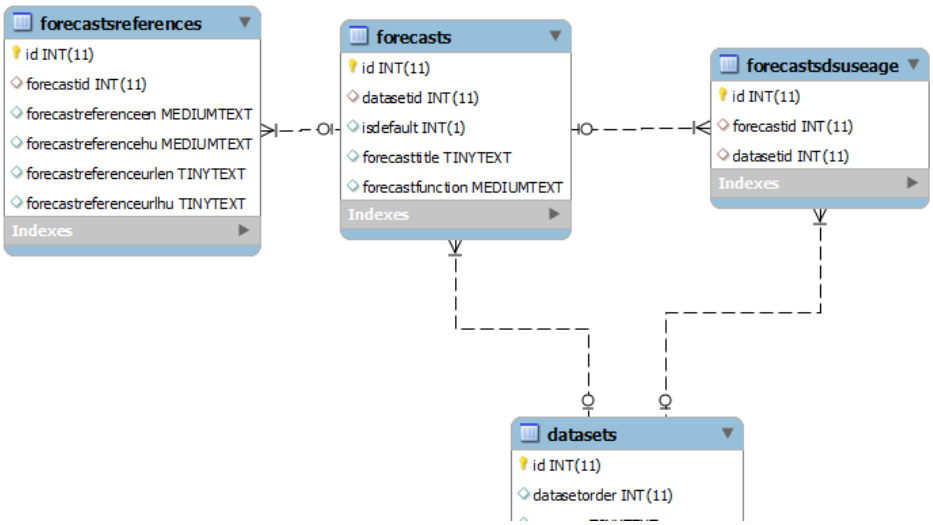

Figure 4 - Database model for forecast functions

- *forecasts* hold the forecast functions for each dataset. Moreover every dataset can have multiple forecast functions.

- *forecastsreferences* serve the purpose to store any kind of article, or other sources, on which the actual forecast is based on. For example, an oil price forecast function can be based on an article, written by the Energy Information Administration (US). This source can be stored as a reference of the actual forecast function. Along with the source of the data, these are also shown with the charts.
- *forcastsdsusage* is a list of datasets which are used by the forecast functions. For example, births use fertility rate and population. This database table serves the goal, to be able to build up a graph for the entire Hungarian macro economy, or any other modeled system thereof, and see which datasets are at the core, or where are the loops in the model.

4. Filling the database with data

After preparing the database, to receive and handle any data series in a unified way, I started to fill it up with data. Adding a multi-column, multi-year monthly series to the system and creating the connecting charts and their related information took only minutes. In this chapter, to demonstrate the process of inserting data, I will walk through the steps of how I inserted the entire Government Budget data, since 2007, into the database.

4.1. Gathering and inserting data

At first, I wanted to download all the budget data from the Treasury's website [4]. To do this nicely, I used '*wget*' [5] limiting downloads to the downloadable links.

```
wget -r -Q100000k --content-disposition -e robots=off -I /letoltesek
http://www.allamkincstar.gov.hu/kincstar/koltsegvetes_merleg_1/2
```

This resulted, all the monthly budget information in excel files arrived to me pretty quickly: *Downloaded: 138 files, 26M in 3.2s (8.05 MB/s)*. These included budget incomes and expenditures from 2002 till today, on a monthly basis. Sadly, some files were in .pdf, instead of .xls.

Doing the same with Social security funds, resulted 206 more files, 64 Mbytes total; continuing with extra budgetary funds, resulted 122 files, 98 Mbytes. I also downloaded Budgetary Institutions, Chapters data, which basically includes the budget of the ministries, 187 files, 34 Mbytes of data. Investments were 135 files, 19 Mbytes. These were also available from 2002, as all the above.

With these all the budgetary information were downloaded. Some local government data is also available on the website but I skipped those, since looking them closely they seemed less important at this stage of the development, so did later. All together 776 excel files, 250 Mbytes combined, were waiting for me to process and put them into the database.

I have also downloaded several other data from the Statistical Office's page, like employment statistics, also from the National Bank's homepage, like the balance of payments. But, in this chapter, I would like to introduce how all the budget data got into the system. It was the same but simpler method for every other data as well.

4.2. Merging and loading data into the database

This has been the first, not strictly IT task on the path to fulfill my goal. Sadly copy-pasting the excel files into MySQL was not a viable solution. There were a lot of different rows in the budget balances from the last 7 years, which had to be merged, before inserting them into the database.

Furthermore, taxation has changed almost every year; not just tax rates but new taxes have been introduced, such as sectorial taxes for banks, telecom companies, while others have disappeared, like payments from the pension reform fund.

It seemed the easiest way to create a blank excel sheet, including all the items which have ever occurred in the budget during the past several years. Into this document I can also put the fresh data later. Furthermore, this sheet will be the exact mirror of the database's budget table.

A monthly budget report's raw excel file looks like this one: *http://www.allamkincstar.gov.hu/letoltesek/10807*. As one can see, this includes both monthly and cumulative data, but also includes year-to-date combined data for every month, luckily. As a result, I had to open the December balance sheets for every year only, to access every month's data.

Looking a report closely, there are about one hundred rows for income items and about eighty for expenditures. These are going to be data sets in my database, all of them, uniquely, named and associated with charts.

These rows are also the subjects for my simulations and forecasts. Meaning, my goal was to create a forecast for each and every one of them. Do this by using unique functions and other data series, like tax rates, statistical data (retail, trade, forex), and by including one-time incomes/expenditures, such as court decisions or nationalizations.

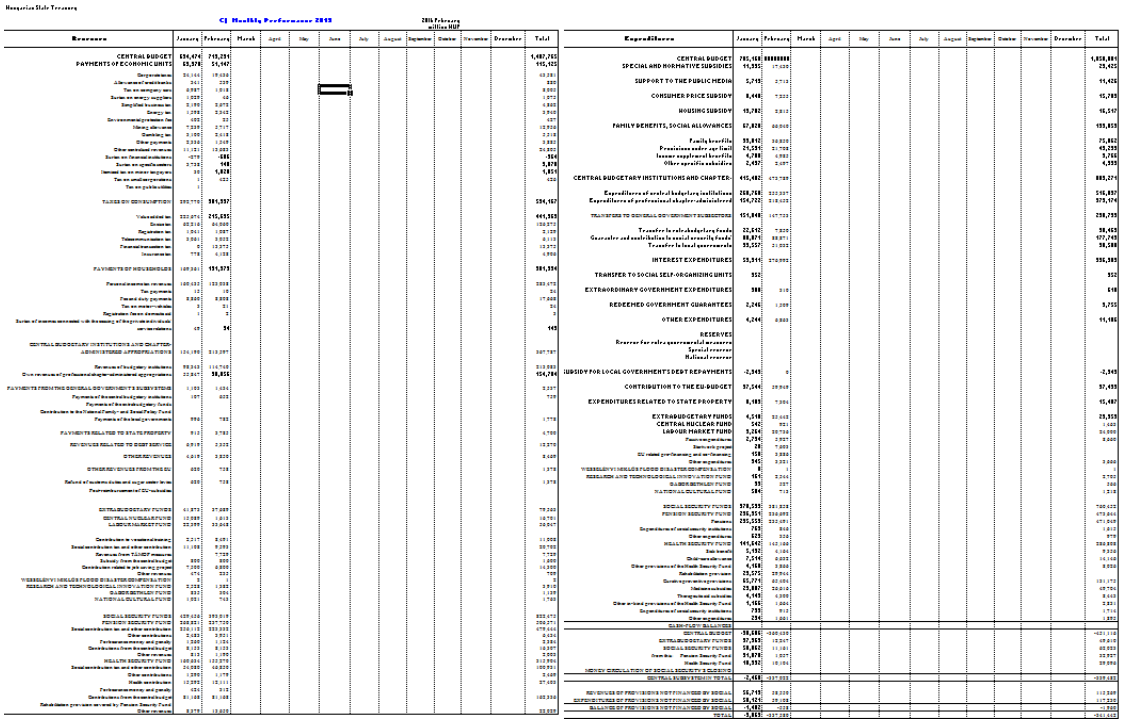

Figure 5 - Budget balance report for 2013 Jan-Feb, incomes and expenditures

4.2.1. Merging data

I started from this year's data, seen on the figure above, and I continued backwards. Meaning I put the 2 columns available (at March) for 2013 into an excel file. Then, I inserted 2012 data before them. This resulted a few rows appeared as shifted. For example, there were no Telecom, Financial transaction and Insurance taxes in 2012, so these rows, as empty ones, had to be inserted for the year 2012, and revenue from the pension 'reform' was non-existent in 2013, so an empty row had to be inserted for 2013. Then, these two years' items became synchronous, as they finally had the same items.

I was doing this for expenditures as well, then continued with the year of 2011 and so on. Meanwhile, after finishing a single year, I performed some random checks, whether the sums of rows equaled the totals, defined originally in the downloaded excels. This was necessary to assure myself that rows did not shifted wrongly and sums were still correct after adding/deleting extra rows. Such checks were also performed after I was able to draw my charts.

Moreover, during the years, some items were merged or separated in the balance sheets. For example in 2010, 'Budgetary Institutions' income was made out of 3 items, two of them were called 'Own revenues' and 'EU support'. However, since 2011, they were merged as 'Own revenues'. As budgetary income is usually analyzed together with budgetary expenditures, it does not really make a difference for me but even if it did, I had no data to be able to separate it again.

Also 'Budget Reserves' were called differently until 2010, than later. In cases like this, I usually used the latest naming, to make it easily identifiable to anyone, who is following Budget Data in general, nowadays.

From 2009, and earlier, Extra Budgetary Funds were not included in the budget balance sheets, therefore, I had to dig out each line separately from those excels which were downloaded earlier and had these data. It was not trivial, since the breakdown of the items was not exactly the same as it is nowadays. The method was, I looked up the 2010 excels, both the budget's sheets and extra funds' sheets, and memorized the data I found. Then I searched for these data in the budget balance sheets, to be able to deduct, which items I needed. From that point I knew, which items to use exactly, from 2009 and earlier, and include in the budget balance, to match nowadays naming.

For example "Other Expenditures" is the sum of "Active Subsidies" + "Vocational training subsidies" + "Rehabilitation purpose job creation subsidy". So these had to be summarized separately and inserted into the budget balance. The same method was applied to Social Security Funds.

Of course another solution could have been to create a separate data set for all the Extraordinary Budget Fund's items, but I did not do that. Later it can be done, if needed, and the Data Set replaced with a derived Data Set.

Sadly, the social security funds' detailed data and the extra budgetary funds' data are not available prior than 2007. Moreover, the format of the budget report is different prior to 2007, so I decided not to store monthly budget data prior than 2007. As my objective is forecasting the future, this did not seem a big sacrifice. Still, they could have been useful for quality assurance purposes.

Merging Budget data for 2007-2013 took a couple of hours, and it was just creation of the excel file.

4.2.2. Loading the data into SQL

I renamed all row names to lowercase, cleared white spaces, quotation marks, etc. Also, I gave prefixes for some of them, such as 'budget_inc_' for budget incomes or 'psf_exp_' for pension funds expenditures. This was necessary to avoid column name duplications. Also, clearing summary rows, like taxes on consumption, were necessary as these became derived data sets.

At the end, I had to transpose the table to fit SQL requirements, so dates will be the rows and budget items will be the columns. It looked like this, with 127 columns.

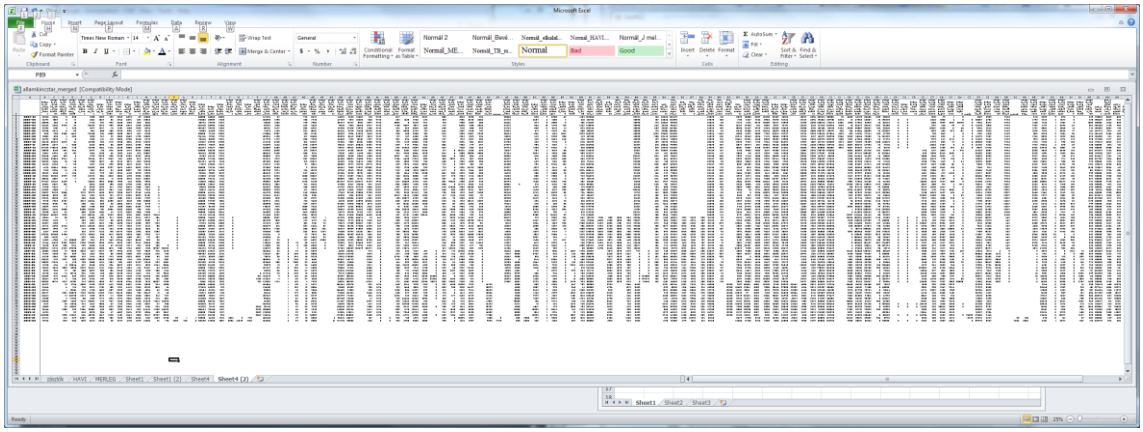

Figure 6 - Budget data 2007-2013 merged into 1 excel sheet

Navicat can import excel files into MySQL with creating the MySQL table as well, so it went smoothly, in a second. All the rest above took about an entire day.

4.3. Creating the Data Sets from static data

Since I renamed all items, I decided to use the column names for the data set names as well. It is not that nice, but simple and fast. Also, data set names do not appear for the users, so it is basically irrelevant, how I call them, as long as the names are unique.

```
insert into datasets (nameen,tablename,columnname,parameter,isderived,frequency)
VALUES ('Budget inc corporate
taxes','budget','budget_inc_corporate_taxes','0','0','monthly');
```

Creating datasets for all the 127 columns took seconds with a few replaces in a text editor. Later, I created separate and visible names as well, like '*Budget – Incomes – Excise Tax*' both in English and Hungarian. Doing that for ~600 data series also took some time. These names appear on the GUI.

4.4. Mass producing charts

After inserting 127 data sets into the database, I did not want to create all charts one-by-one and by hand, nor their thumbnails for the GUI. So I decided to write a few scripts to do these.

First, I had to create charts, basic, line charts, not used as parameters, but containing forecasted data series as well. I selected the new dataset names, then, I created a chart with the same name and added the 2 types of data series onto it. This was done by the following PHP script, which shows the creation of 2 charts only, but I have had 127 of these. The script was created by RegExp search/replace functions mainly in a text editor.

It is worth to note, I used similar techniques in many cases; namely, generating a program code with another program, instead of doing a specific function on a one-by-one basis.

```
$db->rq("insert into charts
(chartgroupid,isparameter,parametertimeperiod,charttitle,chartshortname,charttype,h
axis,vaxis) values (3,0,'monthly', 'Budget inc corporate taxes',
'budget_inc_corporate_taxes', 'ColumnChart', 'month', 'mln huf')");
$chartid=$db->last_id(); $typeid=1;
$db->rq("insert into dataseriesincharts (datasetid,chartid, typeid) values (142,
'$chartid', $typeid)"); $typeid++;
$db->rq("insert into dataseriesincharts (datasetid,chartid, typeid) values (142,
'$chartid', $typeid)"); $typeid++;

$db->rq("insert into charts
(chartgroupid,isparameter,parametertimeperiod,charttitle,chartshortname,charttype,h
axis,vaxis) values (3,0,'monthly', 'Budget inc surtax of corporations',
'budget_inc_surtax_of_corporations', 'ColumnChart', 'month', 'mln huf')");
$chartid=$db->last_id(); $typeid=1;
$db->rq("insert into dataseriesincharts (datasetid,chartid, typeid) values (143,
'$chartid', $typeid)"); $typeid++;
$db->rq("insert into dataseriesincharts (datasetid,chartid, typeid) values (143,
'$chartid', $typeid)"); $typeid++;
```

After having the charts in the database, I made them appear on the screen and saved their thumbnails with a JavaScript function, so the user can see their little thumbnails whenever he/she opens the chart library. To do this, I needed the following JavaScript and a little PHP code to save them. This tool proved to be very useful as I am regularly using it whenever charts change, or new static data is inserted, so new thumbnails are needed. Generating and saving the ~1000 charts takes about 2 hours for the script.

The code for saving the thumbnails can be found in the Appendix. It opens the chart in small, saves it, and closes it, then opens the next one. The image rendering is done by the browser which sends the base64 encoded data to a PHP script, which saves it into a .png file. With this solution, I generated and saved all budget charts in 15 minutes.

4.5. Administering the data

I have created a chart administration interface, which is available for registered users, only with the appropriate rights.

4.5.1. Chart Groups

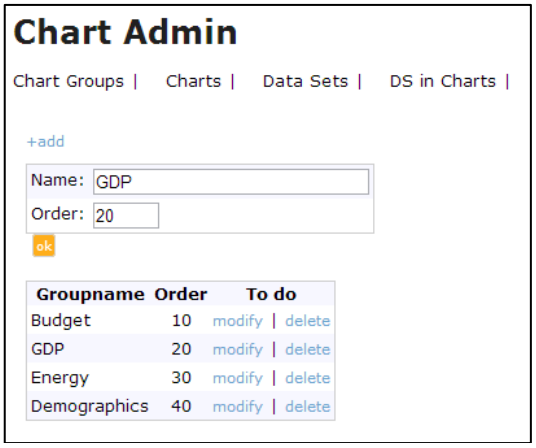

Figure 7 - Chart Groups Admin

First I have to create the chart groups to which certain charts will belong. The purpose is to group charts together. This is easily done by the interface seen on the figure above. All the updates/changes are made through Ajax. Also deleting already used chart groups is impossible as they are connected to 'charts' MySQL table with a foreign key, therefore MySQL restricts deleting this record.

4.5.2. Creating Data Sets and Charts

Then, after having the data series available in the database, I have to create Data Sets, regular and derived ones as well. I can do both on the following screen, where I can easily define Data Sets and whether they are originated from a table's column in MySQL or are derived ones.

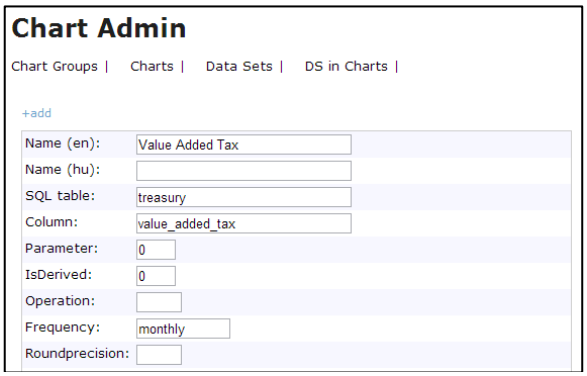

Figure 8 - Defining a new Data Set from a SQL table column

For example, I created 'VAT income' Data Set out of the *budget* table's *'value_added_tax'* column. This Data Set will be added to the Chart of 'VAT income'. As you can see on the figure below, the VAT chart has to be created then the Data Set defined above added to the Chart. Both the actual and the projected data, distinguished by 'typeid' 1 and 2.

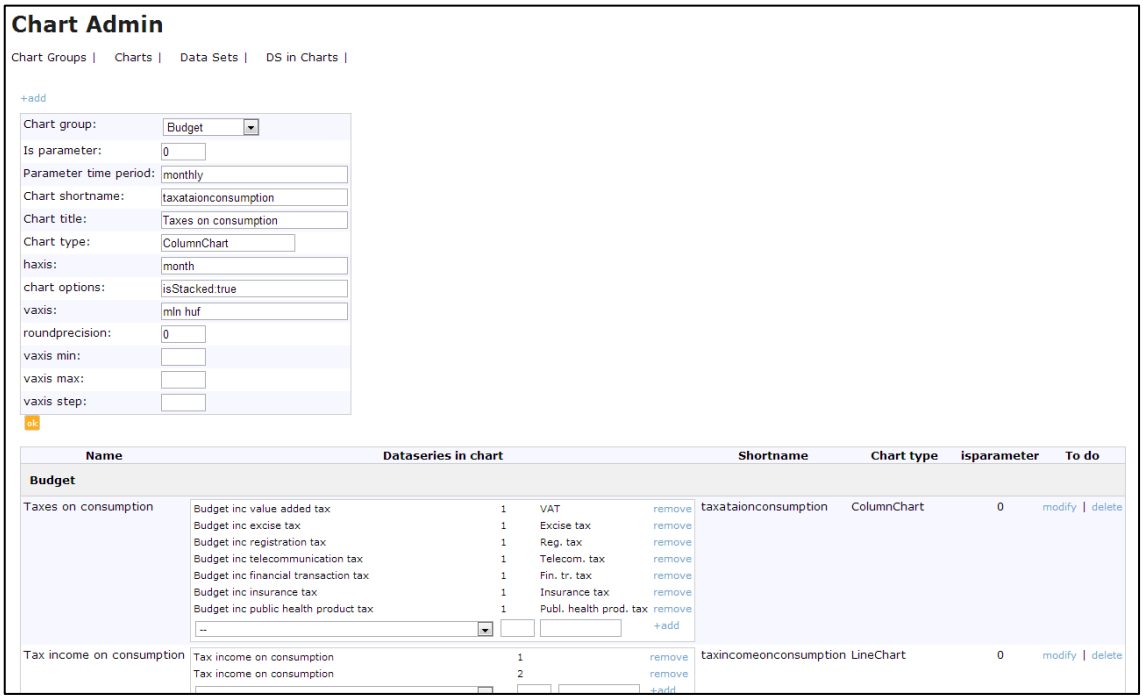

Figure 9 - Creating a new chart and adding Data Set into it.

I also wanted to create a stacked column chart (see below), which includes all consumption related tax incomes. Therefore, I had to add the VAT Data Set to this chart as well (1st chart definition above, on Figure 9). So I can use the same data sets in two charts: first, in the stacked one containing all consumption taxes, second in the one with the forecast for VAT. The purpose is, while on the second one I can show forecasted data for VAT income, I cannot do that on the first, stacked, chart. The problem is: Google Chart Tools is unable to create multi-bar stacked charts. Resulting, if I want to make a forecast on consumption related tax incomes, as a whole, I have to create a 3rd chart as well, containing the combined value of these tax incomes.

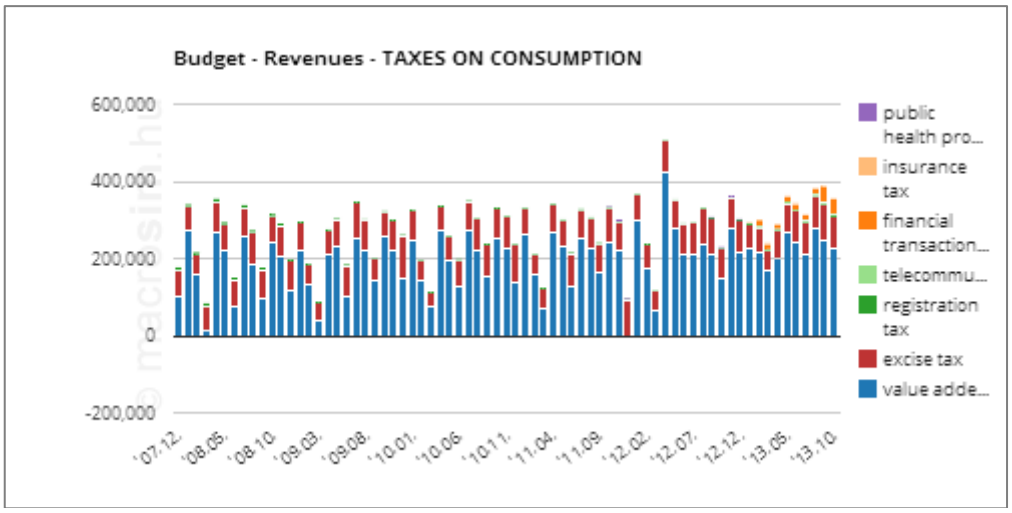

Figure 10 - Stacked column chart on consumption related taxes, actual data

To do that, I had to create a Data Set, a derived one, which is the sum of the consumption related taxes and for which I can create a forecasted column on the chart (2nd chart definition on Figure 9). I created a derived Data Set, an empty one, and added the related datasets into it. The *'operation'* (see on the Figure below) defines what operation is evaluated on the member data series. Then, I can add this new data set to a new Chart which will show the forecasted data for the sum of taxes on consumption as well.

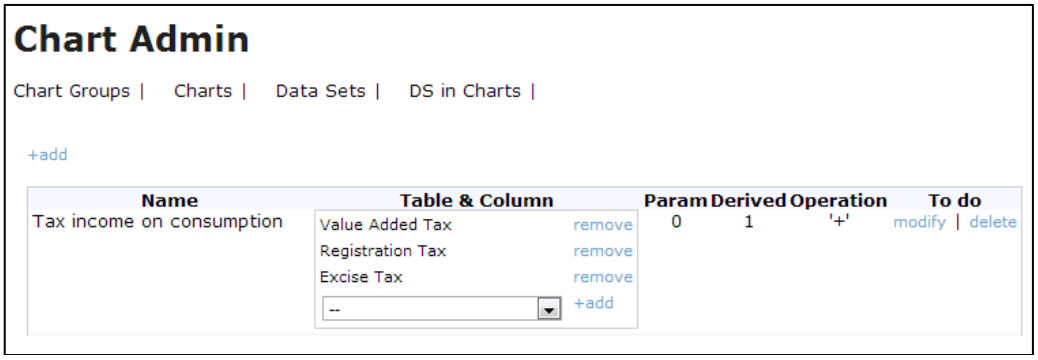

Figure 11 - Creating derived Data Set with any kind of operation on the members

All this is necessary to be able to create a forecast for all budget items later, without knowing what is inside the program code, what exactly I am summarizing. Hopefully, stacked multi bar charts will be available in Google Charts soon.

4.5.3. One time effects

In many cases, actual values are distorted by one-time factors, such as court decisions, changes in laws, privatization or nationalization, etc.

These factors must be excluded from the forecast functions' source data, but they must be included in the forecasted values. To make it clear, in December, 2011 the Government had to pay back a big amount of VAT to companies, because of a Brussels court decision. It caused a big drop in that month's VAT income. However, when I make a forecast for next years' December, I should exclude this item, since I use previous year's data in my forecast function, what should not be distorted by one-time effects when we use it as a base period.

Fun fact, the government itself seems forgetting this obvious rule of excluding one-time effects. They have forecasted 2950bln HUF of VAT income for this year (2013), but they forgot there was a one-time income, about 140bln HUF, in 2012 April, which increased 2012's data to 2750bln. Altogether, the government was extremely optimistic, as it basically forecasted VAT income to increase from 2610bln (without the one-time income) to 2950bln. As of today, after the first 10 months, the VAT income is exactly the same as last year, 2250bln, and assuming slightly better last two months, it will end around 2800bln HUF, missing the target exactly by that one-time amount.

Therefore, when creating the dataset, I subtract one time effects from the 'Actual data', which creates the 'Combined data', from which forecasts are calculated. After the calculation is done, I re-add it to the forecasted value, so the net effect is zero in the result, but in the forecast functions, these factors got excluded. It is like the government should have used 2610bln for forecasting this year's data, instead of the raw data. And when simulating past forecasts for 2012 April for e.g., they should add one-time effects after the calculations are made, to get the right results.

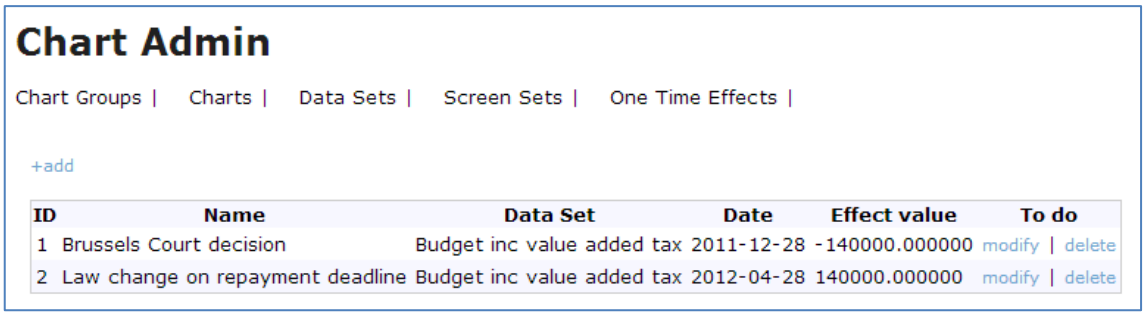

Figure 12 - One-time effects on Value Added Tax incomes

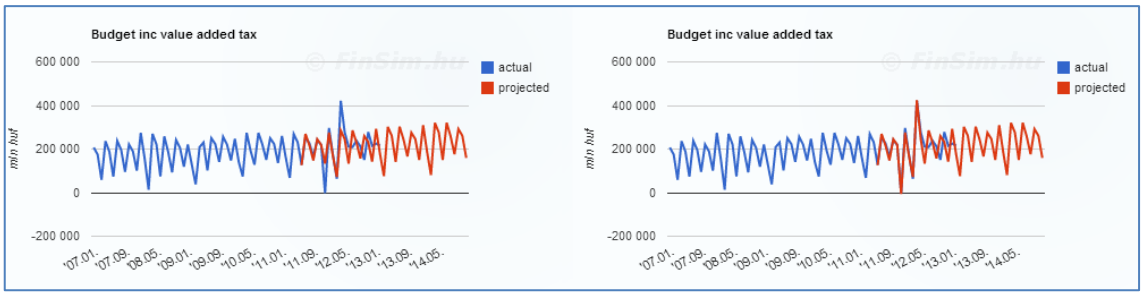

Figure 13 – Without (left) and with (right) one-time effects on VAT

Did the Government lie?

The government in 2011 communicated they had to pay back approximately 240bln HUF of VAT, but as you can see, the forecast function gives close approximation to the actual data with a 140bln HUF one-time amount. This means, either my forecast function is very bad and mysteriously it gives a good result everywhere else, or the government just lied, so they can mask a 100bln HUF hole in the budget and blame it on Brussels.

Furthermore, simulating the law change affecting next April's VAT income, also gives a close approximation with the very same amount. We already knew that it was on purpose to get back the money for the budget they have lost earlier. Just a speculation, but if the government's goal was to get back the same amount of money, all this would make sense. Regardless of the fact, they have communicated differently.

Additionally, if we take a look at the 12 months moving sum of VAT income, we will find, the amount dropped in December, 2011 by 130bln (court decision), and in May 2013 by 122bln HUF (when 2012 April's one time effect flushes out), supporting the ideas above.

5. Creating Object Oriented Environment

I had to create a reliable model in which I can work with many kinds of data easily. In PHP, especially on a webserver, it is not that easy to keep objects alive. Well, basically it is impossible, but we can at least make it look like that a running program is in certain state.

On a webserver every page request through HTTP Protocol makes PHP to recompile the source code and run it from the beginning. Therefore, objects cannot exist between these sessions. We can save objects' data on the server between these sessions, but that is not exactly the same.

Also, my objects should reflect the database model's structure; otherwise I made a mistake designing either the database or the objects, since the database generally serves to store data for objects.

5.1. What objects shall I create

At first, I wanted to create an object for every data set that I thought belong together; like putting births, population, fertility rate, etc… all into one object and name them 'Births', 'Population', 'Fertility rate' respectively. But, as my code evolved, I had to drop this idea. I was trying to create an object called 'DataGroup', and make Population as an extended class of it, but even that proved to be useless.

As I already stated, every data I am using is basically a single 2 columns data series, either they are monthly, quarterly or yearly, they all are the same. So I could not find any reason not to handle them the same way. I decided to make 3 objects as the core of my simulator; these are: DataSet, Chart and SimulationModel objects.

5.1.1. The Objects' class diagram

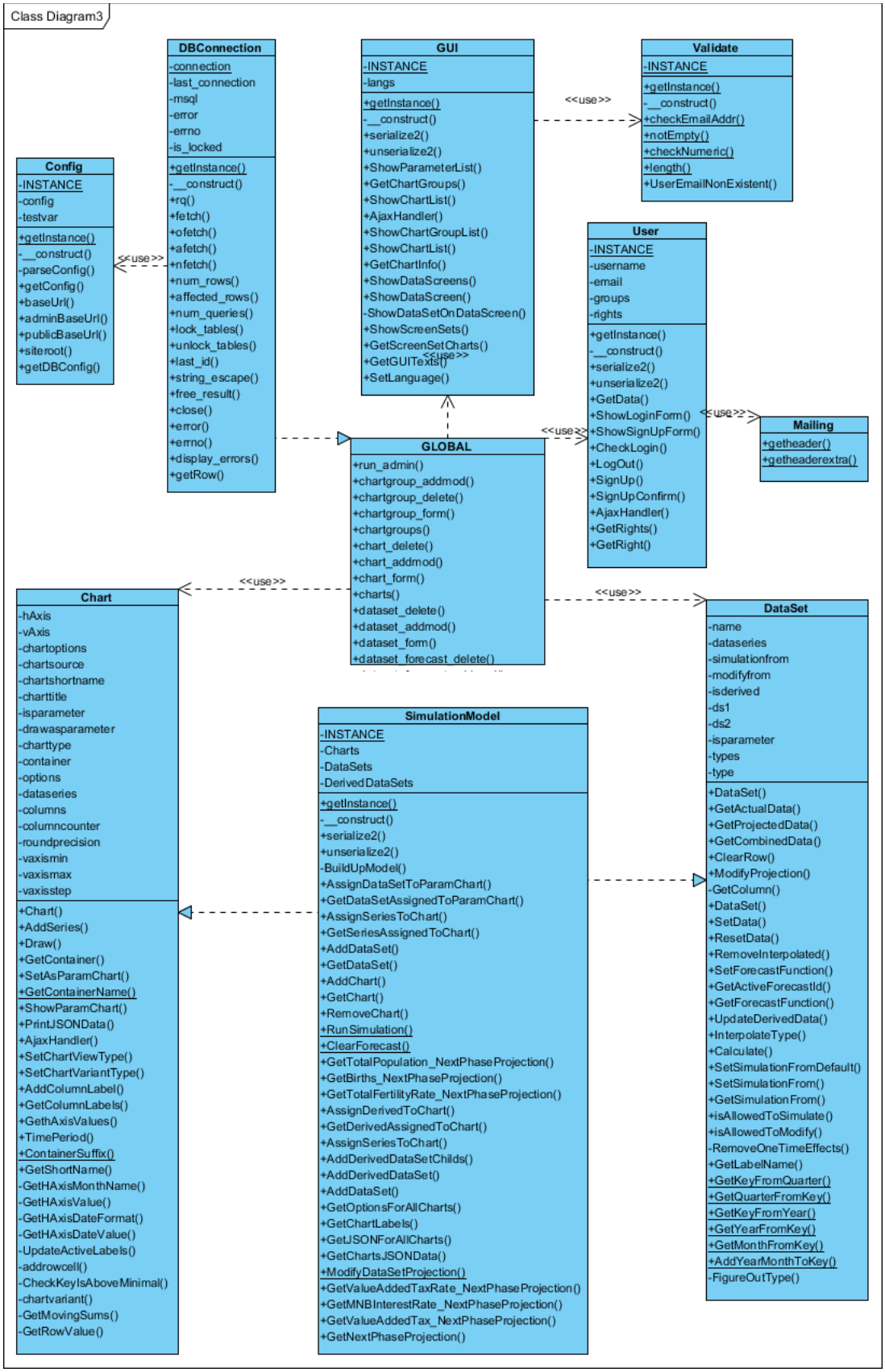

Figure 14 - Class Diagram

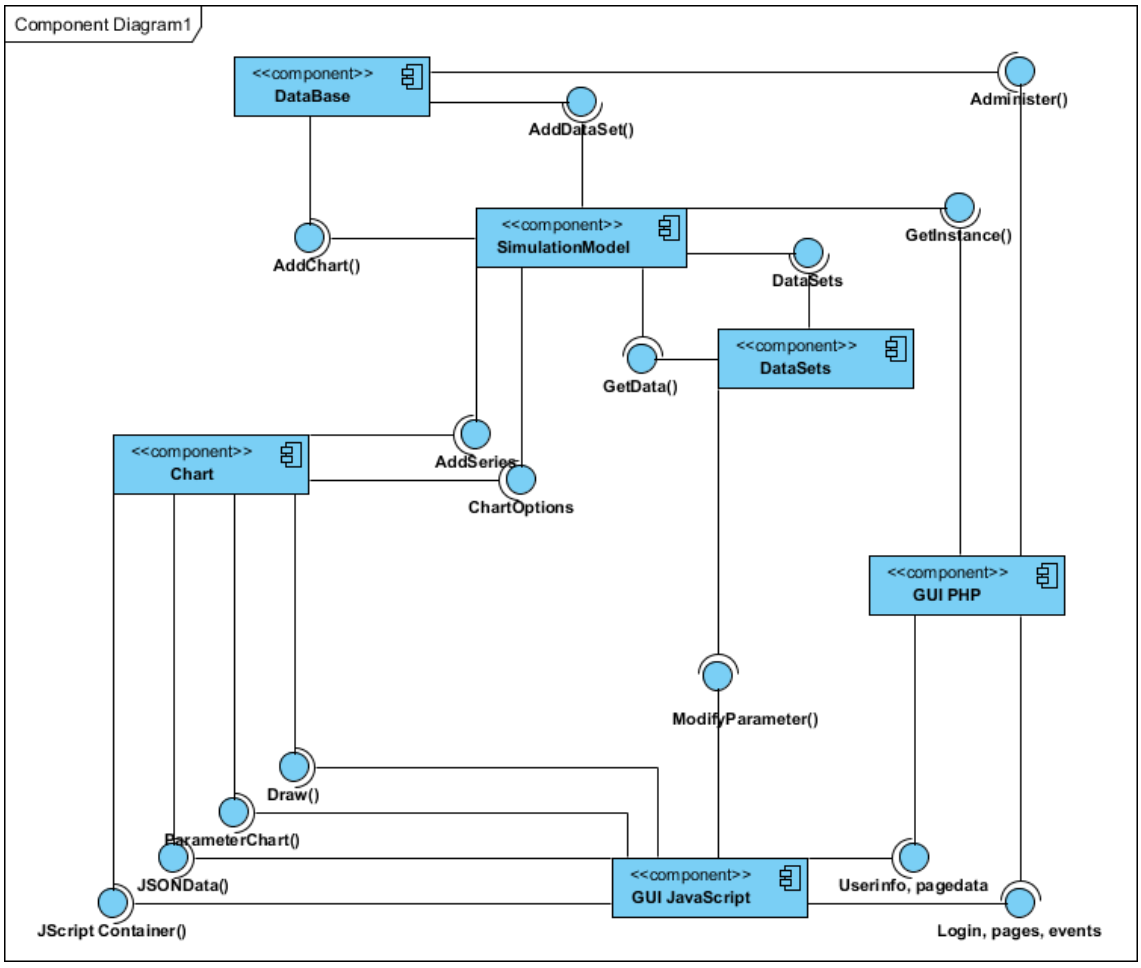

Figure 15 - Component Diagram

The general flow of data is counter clockwise on the diagram. The SimulationModel creates the Datasets and Charts according to the DataBase information. Charts receive the DataSeries from the SimulationModel (which gets them from the DataSets). Then Charts provide the information for the GUI. While the GUI can modify the DataSets and restart the cycle.

5.1.2. Saving Singleton Object

Before creating my objects, the biggest challenge was to keep a Singleton Object, an object which cannot be instantiated more than once, alive through page loads. Generally PHP can store any data between the sessions in the *$_SESSION* global variable. However storing objects is not that simple. The two main problems are: private variables and private functions are not seen from the outside (depends on PHP version), and saving resource type variables, such as database connections and file handlers, through sessions are forbidden [6].

Therefore the database connection had to be initialized every time, and its handler put into a global variable to be accessible everywhere.

About saving objects, it was tricky. You can *serialize()* any kind of object, but since you might be unable to see inside the object, I had to create a *serialize()* and *unserialize()* function inside every object, which I wanted to keep alive. After, I can put this string into the *$_SESSION* and restore it later. This will not keep my objects alive, but at least, it will make them look alive.

With this all I had to do was to initiate the saved objects on every page load, and save them again at the end. A similar solution can be found at [7], whereas my implementation is shown in the Appendix A.1.

5.2. The DataSet Object

This is not a singleton, moreover, it will be instanced in the SimulationModel object, therefore it is enough to keep the SimulationModel alive across the sessions and it will automatically keep the DataSets alive too.

A dataseries is a simple 2 columns array, where the left (first) column contains the keys, namely dates, and the right (second) column contains the appropriate values. DataSet contains some Data Series, usually the versions of one series. Every DataSet has a name, which should be unique, but the parent object will guarantee that, when it creates them.

DataSet reads the actual data from the database with a private function guaranteeing no one else can access the database, names the fetched data as ‘actual’, and also creates ‘projected’ and ‘combined’ arrays, the latter with references to the actual series. The object is also responsible for creating monthly, quarterly, and yearly versions out of any kind of data series. It has also a clearing and modification function to alter all but the actual data.

The purpose of the first two series (actual and projected) is to put them on charts. The combined series is used for simulations, as it has to be the combination of these two. Generally forecasted data is added to the actual data, and where they overlap, the latter is used. So, if forecasted data overlaps with the actual (in cases when I date back the start of simulation), then I use the forecasted to make further forecasts. I needed this, to be able to make forecasts starting from earlier dates, and base later forecast values on already forecasted data. For example, if I have a forecasted data for 2013, I use it as a basis to forecast 2014, instead of using the actual data from 2013. This is on purpose to see how accurate the forecasts would have been for 2014, if I had run them at earlier time, like in 2012.

Derived Data

Data Sets can be created as derived ones out of other Data Sets, for example Births/Population, Debt/GDP. To achieve this, a slight adjustment was needed. The constructor has to be able to handle the extra parameters defining member data sets. Also Simulation does not (and should not) update derived data.

I had to make a function inside the DataSet object, which updates data on demand (for example: when JSON data is needed to draw charts). So I do not recalculate them every time. Also SimulationModel has to maintain a list of *DerivedDataSets* as well, in order to know on what objects to call the *UpdateDerivedData()* function. While MySQL tables had to be changed to store the extra information, but the way of assigning these type of data sets to charts remained the same.

Additionally, the *UpdateDerivedData()* function has to be recursive in case the derived DataSet is built from other derived DataSets. An example for this is *Unemployment Rate,* which uses *Economically Active* data set which is also a derived one. In the GUI admin panel, they look like this:

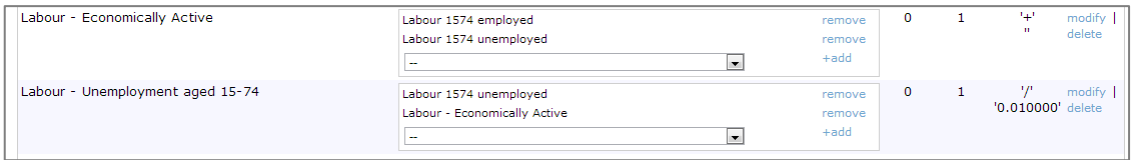

Figure 16 - Derived data set as member of another derived data set

Derived data sets may also use *Interpolated()* data. For example GDP / capita is a quarterly series even though Population is yearly by default. To solve the situation, I had to implement an interpolation function inside the DataSet which calculates the missing data. I was thinking a lot whether to make it a spline interpolation a.k.a. C2 continuous curve [8], but at the end I decided to 'keep it simple' and make it linear.

The downside is interpolated data can consume memory which would not be a problem in normal case but here, were we save/load the entire model with its data at every HTTP page request, it may count. So I also implemented a *RemoveInterpolated()* function which recursively calls itself on it children datasets (from which it is derived) and clears the interpolated data. Of course interpolated data must be produced on demand again.

The quarterly interpolated data for population is about 10Kbytes and this is just the actual data, not the forecasted. Speaking about 10s of charts this could have been easily up to the megabytes territory. The Simulation itself may also use the interpolated data, therefore removing it on Chart destroy might be a bad idea, as it may spoil other forecast functions. It is easily solvable with a lock flag inside the object. Or, another solution is to calculate the interpolated data inside the forecast functions, which would like to use them. It saves memory, but consumes processing time instead, as we may actually calculate the very same interpolation multiple times in one round of simulation.

To make creation of derived datasets easier, their definitions were also moved to the MySQL. This has caused some problems, because some of the derived data sets are derived from other derived data sets, where the latter are defined later in the database. Hence, when creating a derived data set, children data sets might not exist at that point. The solution was to build up derived data sets recursively, by creating children first, if they do not exist.

5.3. The Chart Object

The Chart object is a little more complicated. It is designed to fit Google Chart Tools' needs. It has all the variables needed to fulfill chart creation, and it also has the functions to provide the data needed to draw charts.

Charts are stored in MySQL, their names, titles, axis texts, etc. They are empty charts with all the necessary information beyond the data series themselves.

Therefore, once a new Chart is instanced, also from SimulationModel, it has to be filled with data. The solution is SimulationModel has an *AddSeries()* function which receives DataSet types. More specifically: 2 column data series.

Every chart could operate as a parameter chart, if the database information allows. This means, I can create 2 charts from the very same data series but I have to administrate, whether it is a parameter chart or a regular one. This is needed for the JavaScript functions, as both might be visible at the same time.

To fulfill this duty Chart has a defined container in JavaScript, which depends on the *'isparameter'* setting. Note: I can make a call to draw the chart as parameter but if the database does not allow, the JavaScript will not be able to draw. When a Chart is instanced in JavaScript, its *container* value is set properly depending on whether it is a parameter chart or not, and the container name is returned. So the JavaScript knows exactly, where to draw the newly instanced chart.

Chart object also handles the *Ajax* answers for any chart related information request, such as JSON data for Google Charts or the HTML code of a parameter chart. Since the Chart object knows which series are assigned to itself, it can provide these answers. The parameter chart, as a whole, was created by me, in order to be able to modify bar charts, hence the HTML code is the return value, when creating such a chart.

After all this the creation of the Total Population chart looks like:

```
$SimulationModel=SimulationModel::getINstance();
$SimulationModel->AddChart(”totalpopulation”,$_REQUEST["isparameter"])->Draw();
$SimulationModel->GetChart(”totalpopulation”)->AddSeries
('actual', $SimulationModel->GetDataSet(”totalpopulation”)->GetActualData(), 0);
$SimulationModel->GetChart(”totalpopulation”)->AddSeries
('projected', $SimulationModel->GetDataSet(”totalpopulation”)->GetProjectedData(), 1);
$SimulationModel->GetChart(”totalpopulation”)->PrintJSONData();
```

In this case the DataSet was named the same as the Chart, but that is not necessary. The hard coded “actual” and “projected” variables are the label names appearing on the charts. This is hard coded only one place, hence no need to make a variable for it.

The three most important functions are: *PrintJSONData()* (see Appendix) which provides the data for the charts, *GetContainer()* which returns the container name for the JavaScript, and *SetAsParamChart()* which sets it to appear as a parameter chart in case it is allowed and called as a *‘paramchart’*. This function is also responsible for naming the container properly.

```
function GetContainer() {
    return $this->container;
}

public function SetAsParamChart()   {
   if($this->isparameter && $this->drawasparameter)    {
       $this->charttype='parameter';
       $this->container= self::GetContainerName($this->chartshortname, 1);
       return true;
   }
   return false;
}
public static function GetContainerName($chartname, $isparameter)   {
   $container=$chartname;
   if($isparameter)    $container.="-parameter";
   return $container;
}
```

5.4. The SimulationModel Object

After dropping earlier ideas and implementations, I created this object, which is a Singleton, and handles DataSets, Charts, and the simulation itself.

In its constructor it creates the datasets, all of them, but does not create any charts. Those are instanced on demand in order to lower memory usage.

```
private function __construct(){
    $this->TotalFertilityRate=self::AddDataSet('population', 'totalfertilityrate');
    $this->TotalPopulation=self::AddDataSet('population', 'totalpopulation');
    $this->Births=self::AddDataSet('population', 'births');
}
```

After all the datasets are instanced, I can run a simulation, which fills up projected and combined datasets accordingly. The object also guarantees no two DataSets or Charts are created with the same attributes. Once we have instanced a DataSet with a name, it cannot be instanced again from this Object.

```
public function AddDataSet($table='', $column='')    {
    $NewDataSet=New DataSet($table, $column);
    if(!isset($this->DataSets[$column])) $this->DataSets[$column]=$NewDataSet;
    return $this->DataSets[$column];
}
public function GetDataSet($datasetname)     {
    return $this->DataSets[$datasetname];
}

public function AddChart($chartname, $drawasparameter=0)     {
    $NewChart=New Chart($chartname, $drawasparameter);
    if(!isset($this->Charts[$NewChart->container])) $this->Charts[$NewChart-
>container]=$NewChart;
    return $NewChart->container;
}

public function GetChart($chartname)     {
    return $this->Charts[$chartname];
}
```

In case of Charts, we return the container name, this is important because that is the unique identifier for them, and the identifier on the webpage.

With this I was ready with the main objects needed to handle any kind of datasets in one common way.

5.5. Other Objects created

I needed a few extra objects like GUI or User. GUI is handling the graphical interface, and most of the ajax calls. The latter handles user logins, registrations. These are not really objects since all the used functions could be outside of object context. Yet, I created a .class file for them for easier human readability. Also the class diagram includes them, which makes it easier to understand.

I also created a DB, Mailing, Validate, and Config classes, which are basically just collections of functions. Yet DB is a singleton and since it is a resource type, it is instanced as a global variable, but it is still not more than a bunch of DB handling functions.

5.6. Speeding up things

At this point I was able to speed up two things. First, the Charts were accessing the PHP engine every time they were resized. Redrawing is essential on resize, but getting new dataset is not. Therefore, I modified the JavaScript to remember the JSON data it has received earlier and return it from the *getJSONData()* JavaScript function. The data size for a chart is around 1-2 Kbytes, but the http connection can lag, causing sensible time delay.

Second, I realized every time, when I instantiate my SimulationModel, I not just rebuild the model, but I access the database itself as well. This could have been a huge mistake and probably would have surfaced anyway sooner, rather than later. The solution was to put all the Model related data into a function, *private function BuildUpModel()* (see Appendix), which creates all the data sets, charts, etc., and call this function only if the *unserialize()* did not build up the data. Therefore, modifying the constructor was enough. My *debug.php* assured me the MySQL queries run only once, at first run, and neither the ajax handler, nor the JSON data handler accesses the database.

```
private function __construct(){
    if(isset($_SESSION["obj"]["simulationmodel"]))
              self::unserialize2($_SESSION["obj"]["simulationmodel"]);
    elseif(count(array_keys($this->DataSets))<2) self::BuildUpModel();
}
```

Both ‘problems’ rooted in the web environment, where we cannot maintain program states properly between sessions; hence we rebuild the objects every time. Both solutions helped speeding up chart refreshment.

5.6.1. KISS – Keep it simple stupid

The well-known [9] principle helped me a lot of times. Not just when I decided to handle everything with three classes, but also when I treated everything as a data series, even parameters.

Continuing on this path, I decided to free up some memory usage for future times. Therefore, I changed 'combined' data series to a reference type, what is pointing to either the appropriate actual data or to the projected one.

Furthermore the *GetXxxxData()* functions return reference types [10], which are received by Chart's *AddSeries()*, so every time the simulation changes the data, no update is needed for the Chart objects as they access the data through references. Of course the chart on the web still needs to be updated. PHP uses a 'Copy-on-Write' method, meaning it copies data into the new variable, to which it was assigned, only when it is changed (written into), till then, the new variable is only a reference type. More about PHP's variable handling can be found in [11] and [12].

PHP can treat strings as numbers and vice-versa, but since I only use numbers (at least in DataSets) I converted everything to number. This is a one-time measure, done after reading the database. Also, by setting floating number precision to 6, instead of 50, keeps memory usage low.

With these solutions, I approximately halved the amount of data stored in the serialized SimulationModel object.

5.6.2. Techniques to accelerate processing

As I progressed further, I saw charts are updating slowly. There were about 7-8 charts opened at the time and it took 7-8 separate HTTP requests. I decided to request all opened charts' JSON data at once. With a slight adjustment in the JSON handler and in the Chart Object's *PrintJSONData()* function, it was easily solvable. Only a short loop was needed in SimulationModel to rush through the instanced Charts. It is worth noting, if the user removes a Chart from the GUI, the SimulationModel Object destroys it as well. Also, the amount of data for one chart varies between 2-8 Kbytes. As a result, building up the HTTP connection still took more time, than the data transfer. Since the parameter charts also receive JSON data format, I included them as well in the response.

5.6.3. Assignments

I separated the DataSet names from the DataSeries names, resulting the latter can be assigned to the Charts (even before they are created). As mentioned earlier DataSeries are just references to the appropriate data. Passing their references later to the Charts is possible, without moving substantial amount of data or having the data ready at all, before creating the chart. I can assign empty data series references to the charts, and change the data in the referenced variable later.

In case of parameter charts, I had to assign the DataSet, not the series, to the chart. That is because parameter charts must be able to modify datasets, so the Chart has to know, which DataSet it is modifying.

These assignments were moved to MySQL later, so the code became clearer also it became easier to add new charts. At the same time, a MySQL view has been created to access all the needed data at BuildUp phase easier.

6. Drawing charts

In this chapter I will show how to draw charts and handle all the data described in the previous chapter by these charts. First, I will introduce the method to create charts, both the regular ones and the modifiable ones. Later, I will explain what kind of software development was needed to serve users' needs.

6.1. Creating Charts

I used Google Chart Tools as my chart representation engine. This is a JavaScript based software package designed for drawing different types of charts, mostly the ones we are used to, like simple line and column charts.

6.1.1. The first Chart

The drawing itself is a simple JavaScript function, which gets the data as a parameter, what can also be the return value of another JavaScript function. The latter can be a JQuery function calling a server side script, PHP in my case, which returns the data in the proper format. For Google Chart Tools this format is called JSON [13]. All I had to do was to generate a JSON formatted text out of the MySQL results. Later, these MySQL results were placed into the DataSet Objects.

My first chart looked like the one below. Its corresponding HTML, JavaScript, and PHP codes can be found in the Appendix with a partial JSON data set.

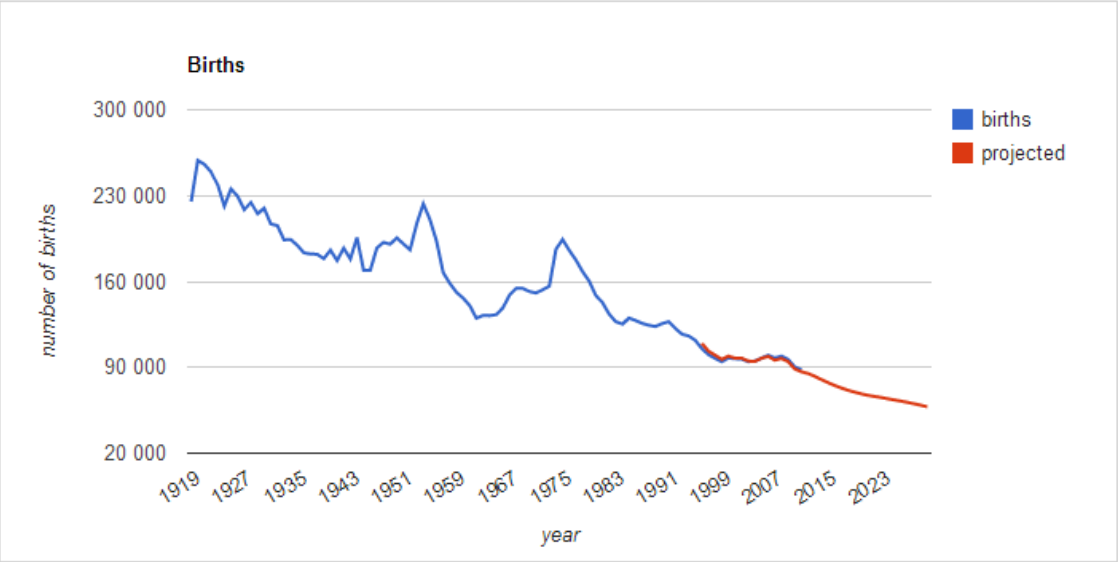

Figure 17 - My first Chart drawn with Google Chart Tools

The Chart Object's '*PrintJSONData()*' generates the proper JSON format from all the data series attached to the specific chart and returns the output. Right before the chart wrapper (re)draws the chart. In the Appendix you can find the PHP code which is inside the Chart Class objects and uses its data series. That data series is generated by creating a new Chart and adding the data series into it.

Note: There is a *json_response()* function in PHP but for Google charts I needed a more complex one, in which I can handle column names, notes, etc. For example, looking at my first stacked column chart closer, I figured out it adds values as strings, so "3"+"4" becomes "34" instead of 7. This has required a slight adjustment, using *is_numeric()*. Source code can be found in the Appendix.

With implementing all the above, I was able to draw a Chart with simply calling the Chart Objects *Draw()* and *PrintJSONData()* functions. The first one initiates the chart, the second fills it with data. Of course both functions are called from JavaScript functions and their outputs are received by the callers, to be able to draw up the chart in the user's browser.

6.1.2. Multiple charts, AJAX refreshment

After accomplishing drawing my first chart, I needed to be able to draw many of them and being able to refresh their inner data without page loading. Both were solved simply. All I had to do was to create a list of charts in the JavaScript, which are already shown. Then, every time I created new data for these charts, I just called the *updatecharts()* JavaScript function, which goes through all the visible charts and redraws them. This was easy but essential to be able to set parameters on the air.

6.1.3. Setting parameters for the Charts on the air

This has been the most important part regarding charts, without question. I was already able to draw and update multiple charts but I was not able to set parameters easily. The main problem was with the parameters themselves, to know which can be set or vary by time. Speaking of births for e.g., Total Fertility Rate varies by time. It is inadequate to ask for one parameter only when I want to make a 15-20 year long forecast. The goal was to make it easy to set different values for every period during the entire forecast time. In case of fertility rate, user has to be able to provide a number as parameter for each year, separately, for the next 15-20 years.

My dream was to make the charts interactive, meaning the user can move the data points on the charts and set the values manually and easily. To achieve this goal I had to create a bar chart manually. For this, I used JQuery UI's slider function [14], which is a simple slider being able to set a value in a range. Multiplying this slider it creates a bar chart in which the user is capable of moving each and every column separately. By default the values are the forecasted values of the appropriate data series. This is on purpose to show the user if somebody had known the values back in time, then what would have been the simulation result. Also, a parameter can depend on its earlier values, like tax rates do, so if we change a tax rate in 2013, we can assume it will be the new value in 2014 as well. Therefore, the values of these columns have to be the forecasted values, until the user changes them.

The parameters, like 'Total Fertility Rate' in the example below, are the same kind of data sets just as births or total population. It is just a decision, whether I use them as a changeable parameter or just a forecasted data set, which might be used in other data set's forecast functions of course.

The screenshot shows a manual change of fertility rate and its effect on the births chart. The refreshment of the births chart happens in real time, almost immediately. Additionally, any other charts refresh, such as total population, which uses a more complex method calculating the data. Since the number of births is certainly among the needed data for forecasting population, hence changing TFR affects Total Population. As a result, when I reduce the number of births for the year 2021, then the forecasted population line becomes steeper, as smaller population will be expected.

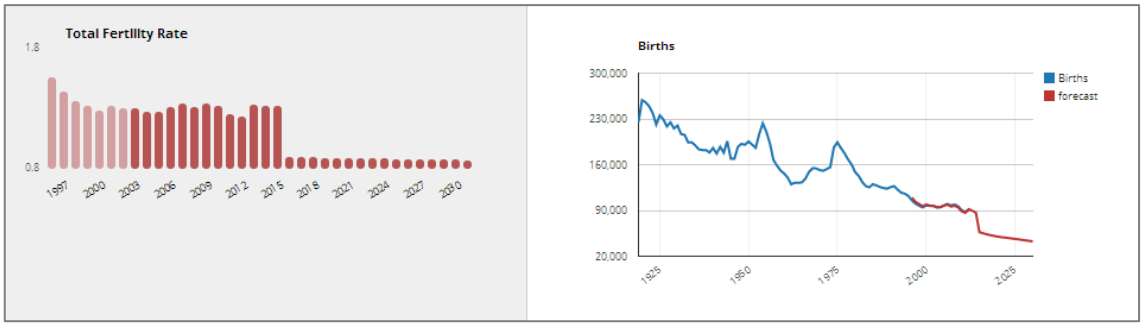

Figure 18 – A parameter change and its effect

From this moment I was able to set any data series as parameters. So I can use '*totalferilitityrate[$year-$month]*' as a parameter in any forecast function. Note: this became the following later, as Objects had been created.

```
&$SimulationModel->GetDataSet("total_fertility_rate")->GetCombinedData();
```

6.2. Improving charts' user interface

Even though I will introduce the GUI in chapter 0, I would like to describe the necessary developments made with Google Chart Tools in this chapter. These were programming tasks, in order to make the GUI work as it is working. First, I will show how I applied certain Google Charts functions onto my charts. Later, I will demonstrate how moving averages and year-on-year comparison were made available on these charts.

6.2.1. Chart Range Filter

Chart Range Filter is a built-in function of Google Charts, which allows the user to change the visible time range of the charts, and show a smaller or wider time period. The only condition is that the horizontal axis has to be a continuous (time or number) scale. However, this condition engendered a lot of other problems, for example, a monthly chart, especially a column chart, is not divided into equal spaces in case of a continuous axis, rather, the spaces follow the lengths of the months. This means, if I draw 3 columns on a chart, Feb-Mar-Apr, then the column of March will not be in the middle, but a bit closer to February, as the second month of the year is only 28 days, while April is 30.

To solve this problem horizontal axis had to be string type, so the spaces will be equal. However, it contradicts with the original condition, that the axis must be continuous. Luckily, the values written on the chart can be converted into string and still used as date type.

With this solution I was able to apply the Chart Range Filter on every chart, making it possible for the user, to narrow the time period of the charts accordingly.

6.2.2. Show/hide columns

While Chart Range Filter has a built-in graphical interface in Google Charts, showing/hiding columns has not. Moreover, when I hide a column, the colors of the remaining data series change on the chart. So my task was not just to create an interface, but to handle coloring as well.

For example, I have a stacked chart on 'taxes on consumption' (VAT, excise tax, etc) and I want to hide VAT, which is the first data series. After removing that, excise tax becomes the first series, and the first available color is assigned to it. This is very confusing, when we show/hide columns and colors change all the time. Just imagine a line chart, where we hide the blue line, and suddenly all other lines, which do not move at all, change their colors, and one of them will be the blue one.

As a solution, I had to create both a list for colors, and a list for the columns. Whenever I hide a column, I re-assign the original colors to the remaining (visible) columns according to their original position. Furthermore, I had to forbid hiding all columns, at least one must be visible all the time, otherwise Google Charts throws an error.

6.2.3. Group by months

Many analysts like charts where they can compare years to each other. In order to make this available, I had to group each data series by months and redraw the chart where every year is a separate data series (from the chart's point of view).

The big question was where should I create this new data; in the Chart object or inside the DataSet object? I chose the former, as I think this service is part of the Charts, and it has nothing to do with the original data. For example VAT income data does not change at all, just because I group them by months, this is only a visibility service, hence it is enough to change the already introduced *PrintJSONData()* function, and return the very same values in a different order.

As easy as it sounds, it took about a whole day to alter the function accordingly, and to create the necessary new functions. Additionally, the colors of the data series changed again, as a simple VAT data suddenly became 7 different data series, one for each year from 2007 till 2013. This was not a problem at first, but I also made both column and line charts available for this type of grouping, causing a little confusion. When I change between column/line charts, while grouped by months, I should keep hidden columns hidden. But when I change back to the original view, then I have to show all columns again. Moreover, this service can easily produce 10-20 or more data series on one chart, which results errors in Google Charts. Therefore, the service of grouping had to be limited for those charts only which were monthly, hence containing data from 2007, and which are not stacked.

6.2.4. Moving averages

Many analysts also like moving averages. Just like in case of the monthly groups, I considered moving averages as a different type of data representations, which has nothing to do with the original data itself. This resulted I built this function into *PrintJSONData()* as well. This one also took about an entire day, as many new problems have arisen again.

First, the first few data rows had to be excluded from the chart, as I cannot calculate a 12 months average for e.g. for November 2007, since I have no actual data from 2006, to make that calculation. However, if I exclude (hide) a data series, then colorings change again, so I decided not to hide them, just to empty them. This results, the chart shows the data series for 2007, but the values are empty records. I had to keep the series for 2007, so I did not have to administer which was the earlier state for the column of 2007, hidden or shown. So the user can switch back to raw data, or 3 months averages, while keep the already set visibility for the 2007 series.

The second problem was, this service had to work with both the original view and the monthly grouped charts, but it had to also work with stacked charts, for what the monthly grouping was not available. Luckily this proved to be the smaller problem.

6.2.5. An example of the charts' interface

To demonstrate the capabilities I place a few screenshots below to show the how a chart can be altered by the functions described above.

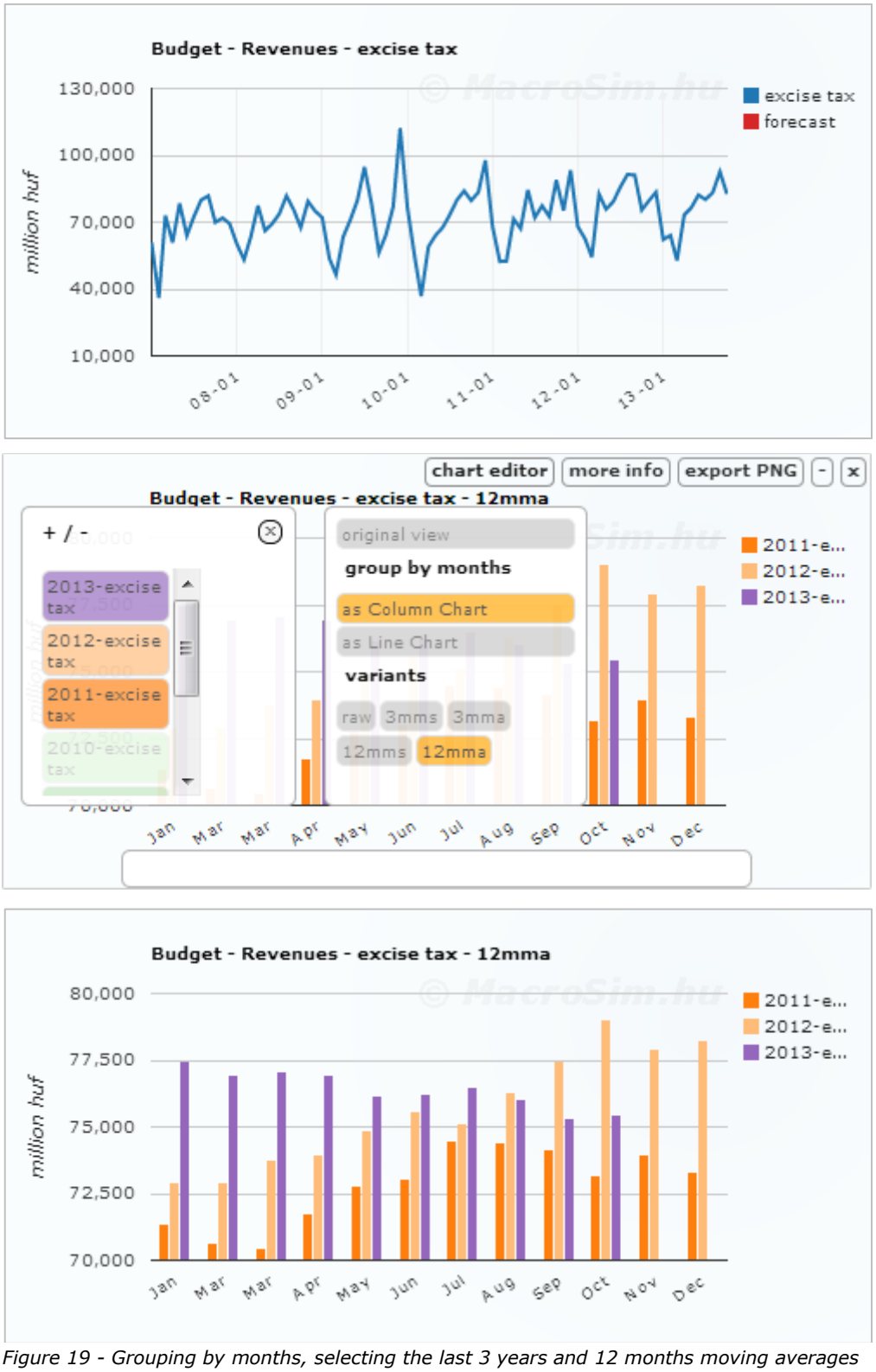

Figure 19 - Grouping by months, selecting the last 3 years and 12 months moving averages

7. Forecasts

Three goals had to be met by the forecast functions. First, they should be easily modified in order to make quick changes in these functions. Second, I have to be able to make 'what-if' type simulations. Whenever the government plans a new law, or multiple scenarios have to be compared, these functions have to be accessible simultaneously. Leading to the conclusion, one data set must have multiple forecast functions. Third, the functions must store what data sets they use, so a usage graph can be built up, to see the loops and causality of these functions.

7.1. Handling forecast functions

Each dataset can have multiple forecast functions, thus making one of them the default one. Also, each function must have a separate list of data sets, the ones it uses. While this has no functional need, the advantage is, I can build up the global model, without understanding it thoroughly. Finally, each function should contain a list of references, where the specific functions are described. Typically these are news articles, law drafts, etc., resulting every 'what-if' simulation will have its own description.

7.1.1. Storing forecast functions

Any user, if he has the rights, can open a modal window with the corresponding functions of the particular dataset. Here, we can define which one is the active function, and we can add/modify/remove functions and their references.

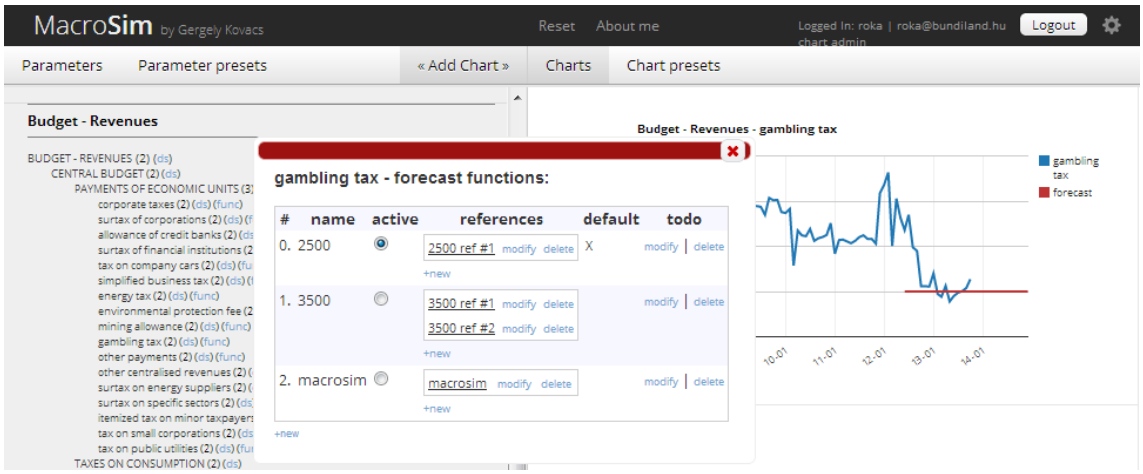

Figure 20 - Forecast functions for gambling tax

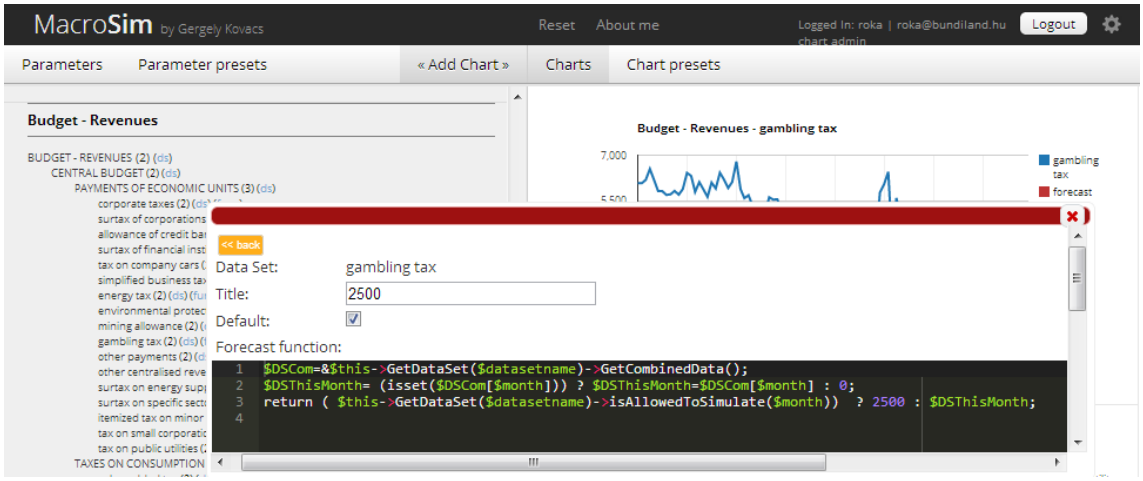

Figure 21 - Modifying the '2500' named forecast function of gambling tax.

As you can see, the forecast function is similar to the one shown earlier but this time, it is in the database. The list of the datasets, which are used by this particular function, can be set also in this window. With this solution all of my goals were solved at once. Generally, anyone could create his own forecast function and set it active, making any kind of simulation not just possible, but storable for later use.

The used program code editor is called Ace Code Editor [15], which had to be applied to the modal window.

7.1.2. Applying forecast functions to the simulation

In order to avoid frequent database accesses, each data set stores its own and active forecast function within itself. It is created during initialization, and can be called anytime, for example when changing the active function. After changing the active forecast function, all I have to do, is to run the simulation again, and the newly activated forecast function will be used.

```
//inside DataSet

function SetForecastFunction()     {
       global $db;
       $res=$db->rq("select forecastfunction from forecasts where isdefault=1 and
datasetid in (select id from datasets where datasetuniquename='".$this->name."')");
       if($db->num_rows($res))     {
              $r=$db->fetch($res);
              $this->forecastfunction=$r["forecastfunction"];
       } else {
              $this->forecastfunction="return 0;";
       }
}

function GetForecastFunction()     {
       return $this->forecastfunction;
}

//inside Simulation

case "budget_inc_gambling_tax":
       return eval($this->GetDataSet($datasetname)->GetForecastFunction());
```

Note, this solution is a huge security risk, as it allows users, to insert code into the program, and run it. Since I do not plan to allow any users to access this feature, it does not seem to be a risk at all. At least, it is not any bigger then my own mistakes. Moreover, it is possible to create a new, pseudo language in order to allow any user to write their functions. Then, both security issues and publicly available functions were solved. However, creating a new programming language is certainly outside the scope of this thesis.

The whole idea behind was, to make the forecast variants available to everyone, and be able to create new ones on demand. So anyone can see the different outcomes, without having access to the source code.

7.2. Defining forecast functions

When I finally had all the data set in my database and I could draw them as charts, including projected data, I could start making projections for each and every one of them.

My goal, as I stated it earlier, was to make a decent budget projection therefore I decided to make an individual projection for every item found in the balance sheet of the budget. Some of them were quite easy, some were not. For example the Central Nuclear Fund receives a big amount of money each January then a small, almost fixed, amount every other month. The same applies to NGOs with the distinction they receive bigger money every quarter. Forecasting these is easy. But making a VAT income projection is rather hard, very noisy and seasonal. Even worse, it is the single biggest item in the budget, hence quite important to make a good projection for it

7.2.1. The easy forecasts

Some data series, like the Central Nuclear Fund, or the National Bank's gold reserves are very easy to forecast. Typically these data never change, or they follow a distinct pattern. Moreover, these data are usually only a small part of the entire budget, therefore making a 1-2 billion forint mistake, on a 12 months period, does not really affect the entire budget number, what is well above 10 thousand billion/year. Of course, I can always adjust the forecast to the official budget in these cases, making it more accurate, as the Ministry usually does not miss these types of data. The forecast function for the Central Nuclear Fund is basically multiplying the 12 months earlier amount with inflation. To be more precise, it is 1.611 billion/month except January, when it is inflation*previous January.

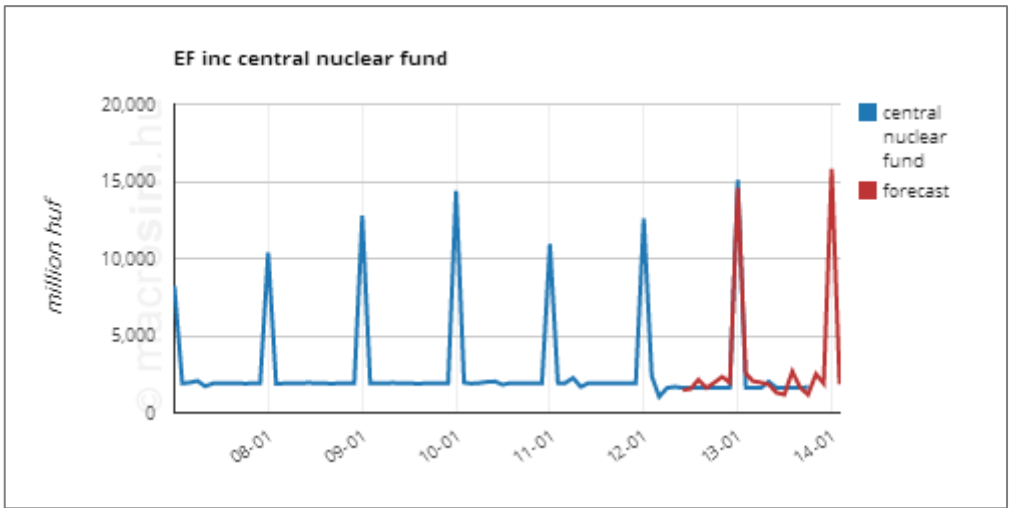

Figure 22 – Corporate taxes - revenues

Data series considered to be easy to forecast include:

- Corporate taxes
- Tax on company cars
- Simplified business tax
- Surtax on energy suppliers
- Itemized tax on minor taxpayers
- Registration tax
- Financial transaction tax
- Surtax of private individuals
- Registration fee on domestic aid
- Private individuals service relations
- Tax payments of households
- Gold, SDR, and IMF reserves
- Population and births
- Economically active 15-74 years old population

These are typically either a constant, like households' tax payments , or at very low levels, like the gold reserves, or following a distinct pattern, like corporate taxes. All this results, their forecast functions can be created in seconds basically.

7.2.2. Not that easy forecasts

Looking at the expenditure side of the Central Nuclear Fund, it is all but easy to forecast. The good news is, it is still a small amount, the bad news is, it shows a rather random pattern. Luckily, since this is a budgetary institution, it has a publicly available budget at the atomic energy authority's website [16]. Here we can find both the last year's budget, what is useful to compare it to the actual data, and this year's. Since last year's budget matches the actual data, I can assume this year it will match too, spending 6.489 billion HUF. Moreover, I also assume the increment will be somewhat similar to next year as well. To put it simple, I basically divided the planned expenditures with 11, as the pattern clearly shows no expenditures in December, and that became my forecast function.

Administering these forecasts in the database

Here, I could finally use the storage method for forecast functions. First I stored a forecast function. Then, I have added the data source, namely the website address of the nuclear fund's budget, to the forecast function's references. Finally, I applied this information to the chart's 'more info' section. Figures follow respectively.

Figure 23 - Forecast function for the Central Nuclear Fund's expenditures.

Figure 24 - Defining the references for the nuclear fund's forecast function.

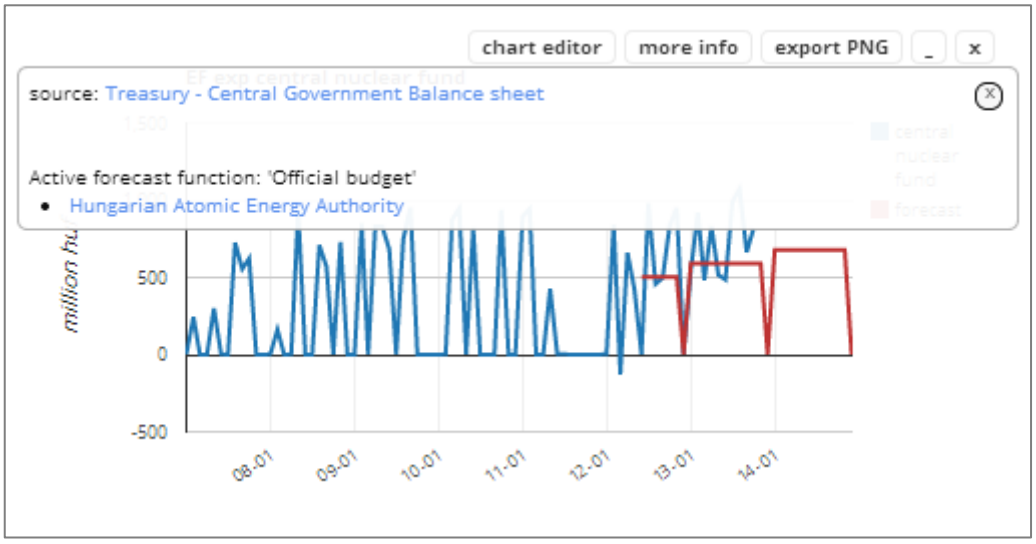

Figure 25 - The chart and its information on nuclear fund expenditures.

7.2.3. Hard to forecast data series

Reverse engineering inflation basket

The statistical office provides inflation data for 12 groups, such as food, housing or education. But the average of them is weighted differently, so the inflation at the end will be a number which is not the simple average of the 12 groups.

The solution was quite simple, I looked up a matrix solver on the internet, [17], copy pasted 12 columns of the inflation record into it and copied another 12 result data, and it gave mi back the basket.

	2013. Mar (1)	Basket weights (1)	(1)*(2)
Food and non-alcoholic beverages	103.3	0.2117	21.86861
Alcoholic beverages, tobacco	113.5	0.0946	10.7371
Clothing and footwear	99.5	0.0479	4.76605
Housing, water, electricity, gas and other fuels	95.9	0.1578	15.13302
Furnishings, household equipment and routine maintenance	101.2	0.1709	17.29508
Health	103.0	0.0605	6.2315
Transport	100.5	0.1298	13.0449
Communication	103.2	0.0696	7.18272
Recreation and culture	101.0	0.084	8.484
Education	104.1	-0.0154	-1.60314
Restaurants and hotels	103.3	-0.2223	-22.96359
Miscellaneous goods and services	106.5	0.207	22.0455
	Off. Data		Calculated:
Inflation Total	**102.2**		**102.22175**

From this point I can make separate projections for each group and also a projection for the overall inflation.

Forecasting Debt

Hungary's public debt is calculated as the ratio of the consolidated debt of general government / current GDP.

The first data comes from the NBH (National Bank of Hungary) which is calculated not that simply. The non-consolidated data is the sum of central government + local governments + social security funds' debt. It is quite simple to calculate and check from the data available at [18]. But the consolidated data excludes those debts where the local governments owe the central government, etc. Also treasury bills are evaluated as the payback value, while the Government Debt Management Agency [19] evaluates them at the face value. The difference is the yearly interest paid on these, roughly 5% on the 2000bln HUF T-bills.

After many discussions with the NBH over the phone, it turned out also some of the government owned companies are included, for example public transportation. Also Gripen-leasing is considered as a debt, even though it is a leasing.

All in all, since I do not have all this data available chose a simpler solution. The SSF and local governments' debt is quite simple, and consolidated values are really close to the not-consolidated ones. So the task is to forecast central government's debt. I chose to do this from GDMA's data, since the difference between the consolidated debt and the debt published by GDMA varies around 550 billion forints +- 80 billion (0.3% of the GDP)

Meaning if I simply add this amount to the GDMA debt I can forecast central government's debt pretty close. Adding SSF and local governments' debt to it will give me a pretty good estimate. Worth to note the revisions usually alter the data with bigger difference. Still, if I can find reliable data how this roughly 550 billion (or less because of T-bills interest) builds up, I will change it.

All in all forecasting the debt is calculated from the GDMA published debt (which generally grows with the deficit in a year) local governments' and SSF debt. The first two is also affected by FX ratios, luckily foreign currency loans are known within these data. The first one is also affected by interest rates, especially on the long therm.

On the other hand I have to make a GDP forecast, for the current price as well. The 4 quarter moving sum at SO's (Statistical Office) web page [20] is used as the GDP when debt is divided. To prove this I checked the official NBH debt data and I calculated the debt for each quarter with the 4Qs GDP and got the table below. It is clearly visible the NBH calculates the Maastricht debt [21] this way.

quarter	current GDP	NBH cons. debt	4Q moving sum of GDP	ratio	official value
2011Q2	6828192	21,306.7	27157659	78.46%	78.5
2011Q3	7125577	22,955.8	27508950	83.45%	83.4
2011Q4	7751079	22,690.7	27886401	81.37%	81.4
2012Q1	6350176	22,397.5	28055024	79.83%	79.8
2012Q2	6973411	22,171.5	28200243	78.62%	78.6
2012Q3	7234861	22,205.0	28309527	78.44%	78.4
2012Q4	7717553	22,380.9	28276001	79.15%	79.2

Of course GDP forecast is not this simple. Anyway, to be able to make a forecast for the debt, I had to reload the Financial Assets of the government and the households as well, using the non-consolidated data (because values can be added unlikely in the consolidated case). Also I had to load in the GDMA debt data as well.

Fun fact, I found an error in the NBH's debt data, a sum was miscalculated. They were also surprised when I called their attention, but it did not make any harm at all at the end.

7.3. One solution above all, SPSS

All the work so far was done for the sole purpose to be able to build up my entire macroeconomic model automatically, without any particular knowledge in economics, and to be able to update the entire model in minutes, if needed. So I can refresh my model every month, for e.g.

As you can see, there were many types of data to forecast in many ways. However, my ultimate goal was to create all forecast functions atomically. I wanted to make IBM's SPSS to find all the regression functions among these data, and create a comprehensive model for the entire Hungarian macro economy. To do this, many steps had to be done.

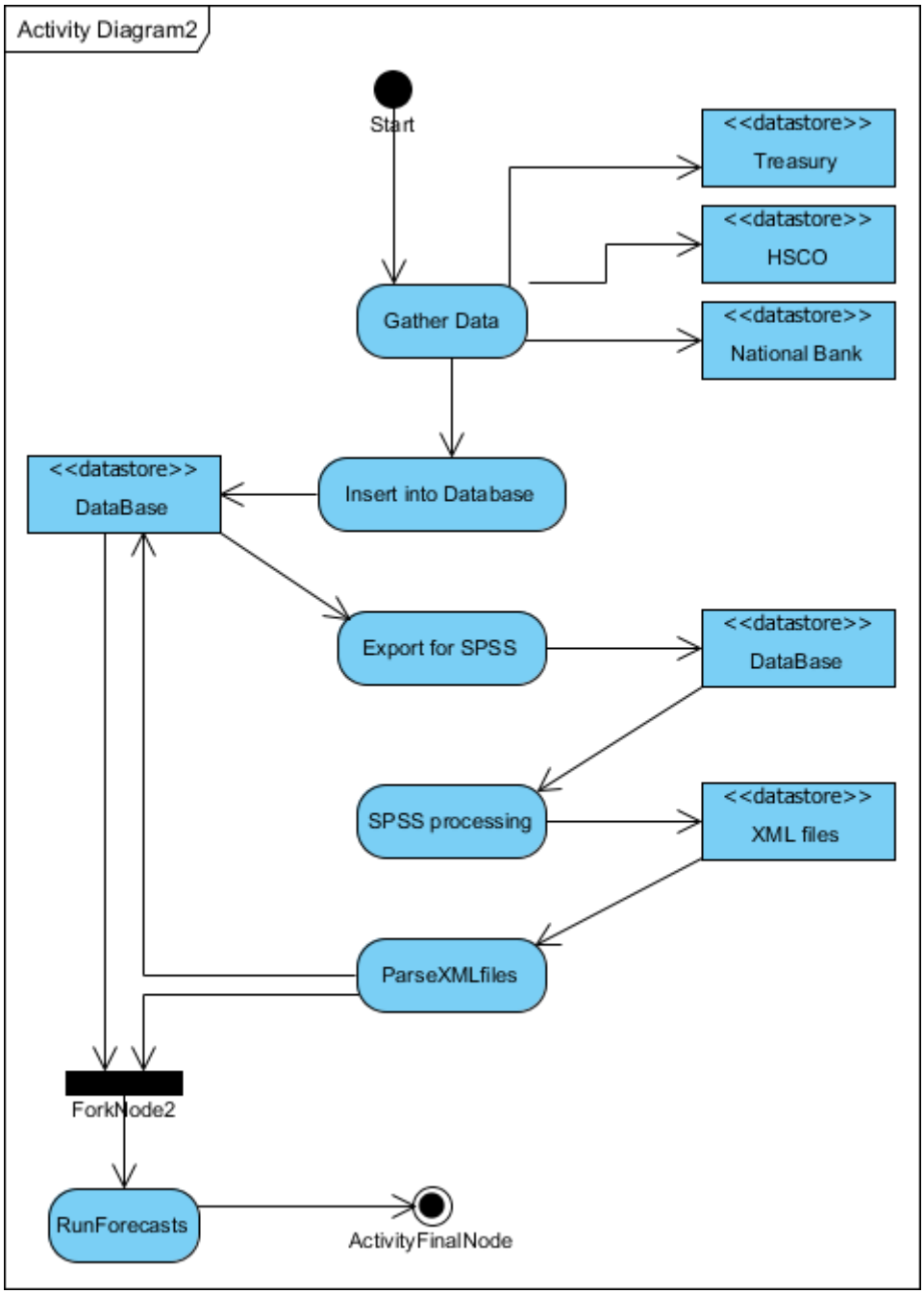

Figure 26 - Activity Diagram for creating the forecast functions

First, I had to figure out what kind of regression functions I am willing to use, and how can I include other types of functions as well. Then, I had to create a database which the SPSS can use. I also had to solve to process the results automatically and put them into my database as forecast functions. Finally, I had to build up a directed acyclic graph to determine in what particular order I must evaluate my forecast functions. In this chapter I will walk through all these steps.

7.3.1. Regression functions

SPSS can handle many types of regression functions; however it cannot handle those, which I wanted to use. These are the ones, where columns are multiplied with each other, like multiplying last year's same month's VAT income with the actual 12 months price index, and use that value as a regression variable. Even if SPSS were capable of doing this, the number of variables would jump from the hundreds to the ten thousands.

Without a question the fastest and simplest regression function is linear regression, with 'stepwise' algorithm. However, I cannot be sure that all my forecast functions will be linear; neither does this algorithm provide a solution to multiply variables with each other.

To solve my problems at once, the trivial solution was to create a database into which these new variables, both non-linear and 'cross-multiplied' are generated. This means, I not just put all my datasets into one single database, but I also generated their inflation-multiplied values into the database. Furthermore, I put the preceding 12 months' value into the same row, along with the values, which were multiplied by the ratio of working days at 12 months ago and now.

To summarize, let's take a look at VAT income's line for Jul 2013. The database contained:

- the raw value for this month: 211,950
- the raw value of last year, Jul 2012: 209,089
- inflation*last year's value: 1.009 * 209,089 = 210,970
- lasts year's square value : 209,089^2
- last year's multiplied by working days ratio: 209,089*23/22

And many other values could be generated, including GDP index multiplication, tax rate differences, and so on. With this solution, a simple linear regression function can investigate non-linear relations, as the values' squares are included; also some multiplication among variables is possible.

The limits are high, as SPSS can handle up to 2 billion columns in every database, whereas Excel for e.g. can handle only 255. Theoretically, I could multiply all my datasets with up to two other in every combination, resulting about 20 million variants. Then I could create another hundred columns for each of them with the method above, including last year's same month's data, or their squares, and so on. For now, I chose to create this few thousand variables 'only'.

7.3.2. Creating SPSS database

SPSS seemed to be the best choice, as many other programs or functions, are not that fast or unable to handle millions of variables. Furthermore SPSS has a script language and XML output, making it the perfect candidate.

To create a database for SPSS all I had to do was to simply export all my datasets into one .csv file, where the rows contain the values for the same month (or quarter). This was rather easy with a small PHP script, which has saved a few hundred Kbytes sized .csv file.

Then importing this .csv into SPSS was simple.

7.3.3. Running regression functions automatically

SPSS is able to run any of its functions in a syntax and/or script window. A simple regression function on corporate tax incomes, for all the columns, and saving it into XML file looked like this:

```
DATASET ACTIVATE DataSet3.
REGRESSION
  /VARIABLES=ALL
  /MISSING LISTWISE
  /STATISTICS COEFF
  /CRITERIA=PIN(.05) POUT(.10)
  /ORIGIN
  /DEPENDENT budget_inc_corporate_taxes
  /METHOD=STEPWISE
  /OUTFILE=MODEL('D:\works\macrosim\spss\budget_inc_corporate_taxes.xml').
```

As it has turned out, using '/variables all' option was a bad choice, as it starts using the variables from the first row for each dependent variable. This means, even though I have last year's same month's data right after each column, SPSS tries the first column first as part of the solution, instead of the most likely one.

To solve this problem, I had to drop '/variables all' option and name each column in order, which I want the algorithm to use. It gave me a lot better results. I also had to drop using constants in the solutions, because without them solutions were also closer. After all this, the syntax saved an XML file containing the regression function I needed.

```
<ParamMatrix >
	<PCell  parameterName="ds142_12m"  beta="0.771271290332966" />
	<PCell  parameterName="ds154_12m"  beta="2.39343217872263" />
	<PCell  parameterName="ds155_12m"  beta="-0.252012588743006" />
	<PCell  parameterName="ds212_12m"  beta="1.19953885365845" />
</ParamMatrix>
```

As you can see, all I had to do was to parse the XML file with php, for which I used *simplexml_load_string()* function, and with a little extra code, it has generated me the following forecast function, which I could place into the database. This code contains the human readable variable names also.

```
$DSCom=&$this->GetDataSet('budget_inc_corporate_taxes')->GetCombinedData();
$DSThisMonth= (isset($DSCom[$month])) ? $DSThisMonth=$DSCom[$month] : 0;
$Constant=0;
$i=-1;
$DS[++$i]=&$this->GetDataSet('budget_inc_corporate_taxes')->GetCombinedData();
$diff[$i]=-12;
$Beta[$i]=0.771271290332966;
$DS[++$i]=&$this->GetDataSet('budget_inc_surtax_on_energy_suppliers')-
>GetCombinedData();
$diff[$i]=-12;
$Beta[$i]=2.39343217872263;
$DS[++$i]=&$this->GetDataSet('budget_inc_surtax_on_specific_sectors')-
>GetCombinedData();
$diff[$i]=-12;
$Beta[$i]=-0.252012588743006;
$DS[++$i]=&$this->GetDataSet('hsf_inc_health_contribution')-
>GetCombinedData();
$diff[$i]=-12;
$Beta[$i]=1.19953885365845;
$ret=$Constant;
for($j=0; isset($DS[$j]); $j++)  {
	$month2=DataSet::AddYearMonthToKey($month, 0, $diff[$j]);
	$ret+= (isset($DS[$j][$month2])) ? $DS[$j][$month2]*$Beta[$j] : 0;
}
return ( $this->GetDataSet($datasetname)->isAllowedToSimulate($month))	? $ret
: $DSThisMonth;
```

I was ready with my forecast function generation for corporate taxes.

The original SPSS output was:

Coefficients[a,b]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
4	ds142_12m	.771	.038	.808	20.312	.000
	ds154_12m	2.393	.356	.254	6.722	.000
	ds155_12m	-.252	.054	-.143	-4.656	.000
	ds212_12m	1.200	.299	.122	4.014	.000

a. Dependent Variable: ds142_raw

b. Linear Regression through the Origin

All I had to do was to write a script, which generates all the regression syntaxes for SPSS for all needed variables and saves them into different xml files. This script can be found in the Appendix.

Finally, applying this function to the chart, resulted the following:

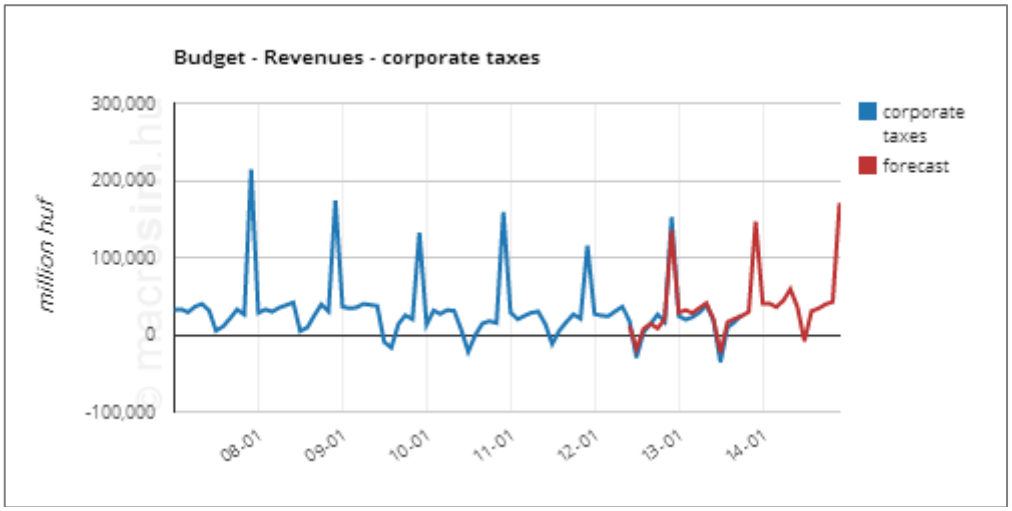

Figure 27 - SPSS's forecast function for budget income on corporate taxes

7.3.4. Building up the graph, and dropping it

Since I had the used dataset names in the xml files, I could easily inserted the dataset ids into the forecasts' *forecastsdsusage* MySQL table, shown in 3.3.3. Once I had all functions' dataset usage stored it took no time to figure out which datasets must be calculated first.

This had to be done, because if a variable uses data from the same month, then every other datasets, which are used, must be calculated first. To be able to do this a DAG (directed acyclic graph) had to be built.

As I found loops I figured out, some forecast functions must be written without using other datasets. Luckily many forecasts could be written this way, as already introduced in '7.2.1 The easy forecasts'.

Yet this was not enough, as SPSS built up a system, which had many loops, meaning it has contained a lot of regression functions with same month's data. After many tries, I could reduce the number of variables to 60, out of 300, which couldn't be forecasted. At this time, the number of levels in my DAG was around 10. The bad news was, stepwise algorithm's PIN/POUT parameters had to be reduces, to achieve this goal, resulting it has dropped a lot of columns.

As a solution I have decided to drop the idea of using columns from the same month, then I do not have to build a DAG at all. I generated the previous month's data into the SPSS input file, and it was enough.

After all this, I had to apply additional modifying functions to various dataset, like tax rates, or the inflation basket, which were also explained earlier. In some cases, like nuclear fund expenditures, I used manual forecast functions, even though the SPSS offered me another.

As a result I had generated about 300 forecast functions, all in a coherent and interconnected system, which models the entire Hungarian macro-economy without knowing exactly what the true relations among variables are.

8. GUI

Finally I would like to briefly introduce the GUI. Even though it took a lot of time to implement I cannot really show how comfortable it became. Still, I will try to describe what I did and why I did, what the objectives were, and what problems I came through.

I wanted to make as many parameters as possible visible simultaneously, meanwhile, I also wanted to show the result charts at the same time. The objective was whenever the user changes a parameter its effects show up immediately on the charts as well. Of course, the entire simulation has to be run every time a single bar or an active forecast function is changed.

After many variations I decided to put the parameters to the left side. Above it is the "Add Chart" button, which includes all the macro topics, what I have collected: from budget data through employment and prices till various National Bank data. These topics generally follow the way, how Statistical Office and the National Bank present data to us.

Basically the user chooses a topic, like budget, selects the charts which he/she wants to use as parameter or see on the right side, like base rate what can be put onto both sides, and puts them onto the screen. Also if the user has administrative rights he/she can administrate the charts' and data sets' data as well, on the very same screen. Putting all these together resulted something like this:

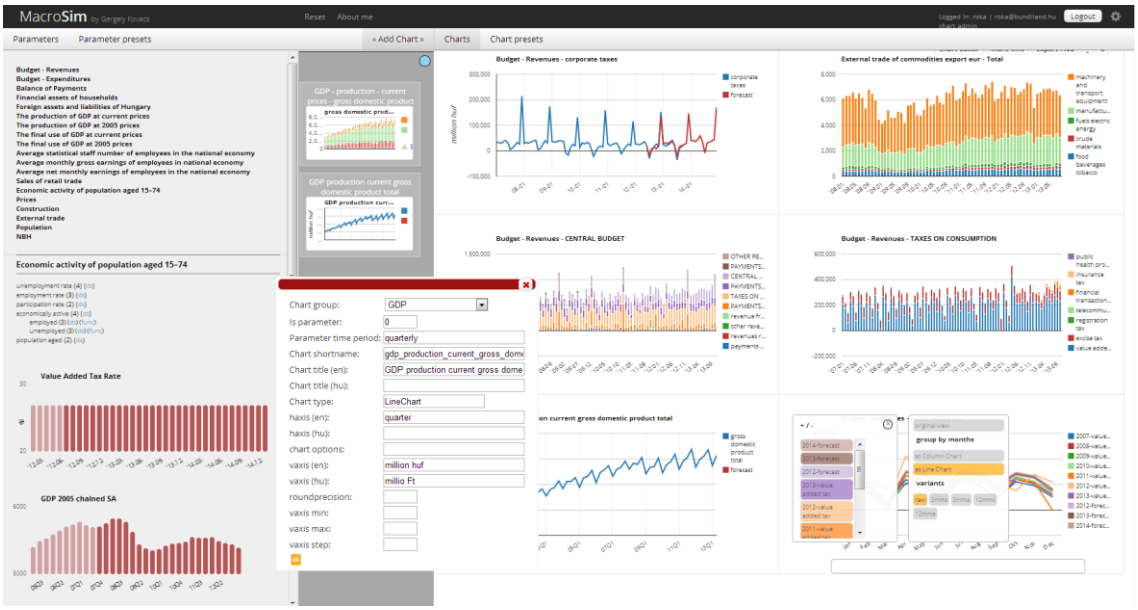

Figure 28 - Screenshot of the GUI

8.1. Adding charts, using topics

After many variations on how to select charts and place them onto the stage, I finally decided to rebuild most of the topics available at the statistical office's and national bank's webpages in a form which is hierarchically equivalent to the data found there. Meaning, I not just present unemployment as a data, but I put it into context how it is calculated, so the user can see the basic connections among these data.

By far, rebuilding the hierarchy of the budget was the biggest challenge, to be able to present the data in the same form as the ministry does. To do that, all the groups and subgroups had to be built up in the database as derived data sets. This was useful not just because I could present the charts in an easy to read format, but because the summary charts and datasets were created at the same time. For example putting budget revenues in a hierarchical structure resulted the budget revenues hierarchy instantly contained all the approximately 80 items and their forecasts as well, as the GUI broke down the derived datasets.

At the end, the roughly 500 charts created so far became easily accessible and understandable to anyone who uses the program. Also the topics are following already seen structures, so whoever has used HCSO's or NBH's webpages before will know where to look for the data, they are interested in.

8.2. Presets

To make the GUI more user friendly, I have created a few sets of charts and parameter charts which can be opened by a simple click, so whenever somebody is interested in rates, or employment data, or budget data, he/she can easily access the most important 5-10 charts and parameters, so he/she does not have to open them one by one. These presets are also administrable via web, so I can add/remove charts into them at any time, or create new sets.

8.3. Chart Editor

This is the most important GUI part, at least for me. Also, this proved to be the hardest one as well. Google Chart Tools is not exactly designed to handle my needs. I wanted to make the charts' timeline selectable, meaning the user can define on every chart, what time window he wants to see. This would have been an easy one, unless I am using Column Charts. More about the topic can be found in [22], where I personally discussed the issue with the developer. The basic problem is GCT cuts first and last columns into halves, since their center is positioned to the exact date. But the view window obviously ends at the last date.

The solution was quite complicated, as described in the above forum, but I made it finally. It was crucial to use it together with my other feature which is adding/dropping data columns dynamically. So if someone wants to see consumption taxes without VAT and Excise tax for example, he can throw out these columns. It looks like this, before and after modification:

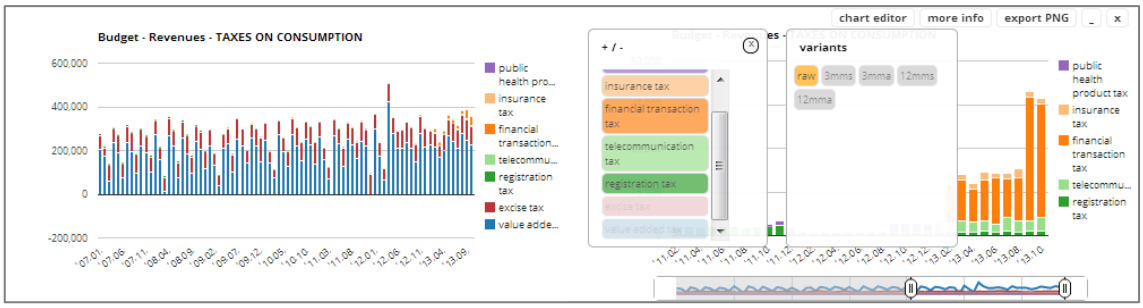

Figure 29 - Using chart editor, to set visible columns and time range

I also included the already introduced variant selector as well, where the user can choose whether he wants to see the data grouped by month, or viewing 3 or 12 months moving averages/sums, or year-to-date comparisons for each year.

9. Summary

My goal was to create a framework and a method to model any kind of complex system, not just macroeconomics. The way of storing, forecasting, and modifying data gives the chance to model other systems as well, from weather to a formula-1 car.

When I started my thesis I had my doubts it will work at all in a way it is working now. However, I trusted my instincts that engineering in fact can offer a new approaches and solutions to known problems. The system I developed may not model the Hungarian macroeconomics as economist would like to see, but it may help us understand the connections among variables we have never seen before. Also it gives us the opportunity to build up models on a scale which is definitely bigger than single individuals can make.

During the development a lot of feature ideas arose, which I hope will be implemented once into this system. These features include from applying linear programming to optimize for GDP growth with constraints on inflation or debt for e.g., through creating a new programming language, till modeling alternative scenarios, such as what would have happened if private pensions had not been nationalized. These features were not part of my task this time but I hope I will have the chance to continue this work and implement these ideas.

Fulfilling my goals feels as a great closure of my studies at the Budapest University of Technology and Economics. I am glad that I had the chance to conclude this work at the Business IT MSc program, what aligns with my interests perfectly. I hope these years prepared me to succeed both at another school and in business life.

Finally, I would like to thank for many individuals for the help and support they provided not just during my thesis work but all these years. Besides my family and friends, I am especially thankful to my supervisor, Dr. Bela Szikora, for his extraordinary patience and trust he has given me.

References

[1] "Google Chart Tools," [Online]. Available: https://developers.google.com/chart/.

[2] Google, "google-visualization-api-issues," Google, [Online]. Available: https://code.google.com/p/google-visualization-api-issues/issues/list?can=2&q=&sort=-id&colspec=ID%20Type%20Status%20Priority%20Milestone%20Owner%20Summary%20Stars.

[3] "Polynomial Regression," [Online]. Available: http://www.drque.net/Projects/PolynomialRegression/.

[4] "Trasury," [Online]. Available: www.allamkincstar.gov.

[5] "Wget - Manual," [Online]. Available: http://www.gnu.org/software/wget/manual/wget.html.

[6] "session_register," [Online]. Available: http://php.net/manual/en/function.session-register.php.

[7] "Storing singleton objects in PHP sessions," [Online]. Available: http://www.robert-gonzalez.com/2012/03/14/storing-singleton-objects-in-php-sessions/.

[8] B. Csébfalvi, "Geometrical Modeling," [Online]. Available: http://sirkan.iit.bme.hu/~cseb/Education/ComputerGraphics/BW/modeling_6.pdf.

[9] "KISS Principle," [Online]. Available: http://www.princeton.edu/~achaney/tmve/wiki100k/docs/KISS_principle.html.

[10] "Reference Type," [Online]. Available: http://php.net/manual/en/language.references.pass.php.

[11] "Copy-on-Write," [Online]. Available: http://www.research.ibm.com/trl/people/mich/pub/200901_popl2009phpsem.pdf.

[12] "schlueters.de," [Online]. Available: http://schlueters.de/blog/archives/125-Do-not-use-PHP-references.html.

[13] "Introducing JSON," [Online]. Available: http://www.json.org/.

[14] "JQuery UI," [Online]. Available: http://jqueryui.com/slider/.

[15] "Ace Code Editor," [Online]. Available: http://ace.c9.io/.

[16] "Hungarian Atomic Energy Authorithy," [Online]. Available: http://www.haea.gov.hu/.

[17] "Linear Equations Solver," [Online]. Available: http://www.bluebit.gr/matrix-calculator/linear_equations.aspx.

[18] "TIME-SERIES OF DETAILED FINANCIAL ACCOUNTS BY SECTORS," [Online]. Available: http://english.mnb.hu/Statisztika/data-and-information/mnben_statisztikai_idosorok/mnben_elv_net_lending/mnben_0602_nemz_modsz_idosorok090107.

[19] [Online]. Available: www.akk.hu.

[20] "Statistical office," [Online]. Available: http://www.ksh.hu/.

[21] "OECD iLibrary," [Online]. Available: http://www.oecd-ilibrary.org/content/data/data-00021-en. [Accessed 03 05 2013].

[22] "Bug: ColumnChart columns halved if hAxis is date," [Online]. Available: https://code.google.com/p/google-visualization-api-issues/issues/detail?id=1212.

List of figures

A. 1. Appendix – Source Codes

Drawing Chart, JavaScript and PHP codes

```
// HTML Code
<div id="births" class="chartdiv"></div>

// Javascript Code calling the chart drawer
AddAndDrawChart('LineChart', 'births', '{
"title":"Births",
"hAxis":{"title":"year"},
"vAxis":{"title":"number of births"}}');

var wrapper;
var charts=Array();

// Adding the chart to the list of active charts
function AddAndDrawChart(charttype, fullchartname, optionstext) {
    var i=charts.length;
    charts[i]=Array();
    charts[i][0]=charttype;
    charts[i][1]=fullchartname;
    charts[i][2]=optionstext;
    drawchart(i);
}

// Drawing the 'i'th Chart
function drawchart(i) {
    drawVisualization(charts[i][0], charts[i][1], charts[i][2]);
```

***

```
// this is the modified original Google Chart Drawing function
function drawVisualization(charttype, fullchartname, optionstext) {
    var optionstext;
    if(optionstext=='') optionstext='{ }';
    wrapper = new google.visualization.ChartWrapper({
      chartType: charttype,
      dataTable: getJSONData(fullchartname),
      options: jQuery.parseJSON(optionstext),
      containerId: fullchartname
    });
    wrapper.draw();
  }

// Loading the JSON data from the PHP
function getJSONData(fullchartname) {
  var jsonData = $.ajax({
      url: "includes/JSON-response.php?fullchartname="+fullchartname,
      dataType:"json",
      async: false
      }).responseText;

  return new google.visualization.DataTable(jsonData);
}
```

***

```
// Chart Object creation with the appropriate data sets and printing the JSON
format

$population=Population::getInstance();
$chart_births=New Chart('births');
$chart_births->AddSeries('births', $population->GetBirths());
$chart_births->AddSeries('projected', $population->GetBirths_Forecast());
$chart_births->PrintJSONData();


//JSON Data output for the Chart

{
"cols": [
       {"id": "0", "label": "year","type":"string"},
       {"id": "1", "label": "births","type":"number"},
       {"id": "2", "label": "projected","type":"number"}
       ],
"rows": [
       {"c":[{"v": "2007"},{"v": 97613},{"v": 95850}]},
       {"c":[{"v": "2008"},{"v": 99149},{"v": 97216}]},
       {"c":[{"v": "2009"},{"v": 96442},{"v": 94659}]},
       {"c":[{"v": "2010"},{"v": 90335},{"v": 88686}]},
       ]
}

// The very first function call which triggers all the rest
$chart_new=New Chart('births');
$chart_new->Draw();
```

***

```
//PHP JSON Data generator

function PrintJSONData()    {
              $GUI=GUI::getInstance();
              $this->activelabels=array();
              $JSON = "{\n\"cols\": [\n";
              for ($i = 0, $columnid=0; isset($this->columns[$i]); $i++) {
                     $yearkey=''; $yearkeys[0][0]=''; $columnid2=0;

                     if($i==0) {
                            if($this->charttype=="SteppedAreaChart" || $this-
>groupbymonth) $type="string";
                            else $type="date";
                     } else {
                            $type="number";
                            reset($this->dataseries[$this->columns[$i]]);
                            while (list($key, $value) = each($this-
>dataseries[$this->columns[$i]])) {
                                   $haxisvalue=$key;
                                   if($this->groupbymonth) {
                                          $haxisvalue = substr($key, 5,2);
                                          $actyear=substr($key, 0,4);
                                          //if(  self::CheckKeyIsAboveMinimal($key,
$i) )  {
                                                 if($yearkey!=$actyear)      {
                                                        $columnid++; $columnid2++;

       self::UpdateActiveLabels($columnid, "$actyear-", $i);
```

```
                                                        $JSON_cols[]="\t{\"id\":
\"".$columnid."\", \"label\": \"".$this->activelabels[$GUI-
>lang][$columnid]."\",\"type\":\"$type\"}";
                                                        $yearkey=$actyear;

        $yearkeys[$i][$columnid2]=$yearkey;
                                                }
                                        //}
                                }
                                $haxiskeys[$haxisvalue]=$key;
                        }
                        ksort($haxiskeys);
                }
                if($i==0 || !$this->groupbymonth)  {
                        self::UpdateActiveLabels($i, '', $i);
                        $JSON_cols[]="\t{\"id\": \"".$i."\", \"label\":
\"".$this->activelabels[$GUI->lang][$i]."\",\"type\":\"$type\"}";
                }

        }
        $JSON .= implode(",\n",$JSON_cols)."\n\t],\n\"rows\": [\n";

        $rowcounter=0;
        while (list($key, $value) = each($haxiskeys)) {
                if(
                        ( $this->variantmode == "raw" ) ||
                        ( $this->variantmode == "ytd"             && $key <=
$this->lastytdmonth       ) ||
                        ( strpos($this->variantmode, "mm") )
                )       {

                        $addrow="\t{\"c\":[{\"v\":
\"".self::GetHAxisDateValue($key)."\", \"f\": \"".self::GetHAxisValue($key)."\"}";
                        for($j=1, $k=1; isset($this->columns[$j]); $j++) {
                                if($this->groupbymonth)     {
                                        for($k=1; $yearkeys[$j][$k]; $k++) {
                                                $addrow.=self::addrowcell($j,
$yearkeys[$j][$k]."-$key");
                                        }
                                } else {
                                        $addrow.=self::addrowcell($j, $key);
                                }
                        }
                        $addrow.="]}";
                        $JSON_rows[] = $addrow;
                }
                $rowcounter++;
        }
        return $JSON.implode(",\n",$JSON_rows)."\n\t]\n}";
    }
```

***

```
// JavaScript

function updateparamchart(dataseries) {
    $.ajax({
      type: "POST",
      url: "/includes/ajax.php",
      data: {getdataseries: dataseries},
      dataType: 'json',
      cache: false,
      success: function(result) {
        $( ".slider-vertical-container" ).each(function() {
            var row=$( this ).children (".slider-haxis" ).text();
            if(typeof result[row] !== 'undefined')  {
                $( this ).children(".slider-vertical").slider({
                  value: result[row]
                });
                $( this ).children (".slider-value" ).text( result[row] );
            }
        });
      }
    });
};
```

Singleton Object

```
include(BASEDIR."/includes/User.class.php");
    $User=User::getInstance();
    if(isset($_SESSION["obj"]["user"])) $User->unserialize2($_SESSION["obj"]["user"]);


class User {

    private static $INSTANCE;
    public $username;
    protected $email;
    protected $groups;

    public function getInstance(){
        if(!self::$INSTANCE) {
            self::$INSTANCE = new self();
        }
        return self::$INSTANCE;
    }

    private function __construct(){
    }

    public function serialize2() {
        return serialize(get_object_vars($this));
    }

    public function unserialize2($data) {
        $data=unserialize($data);
        if (is_array($data)) {
            foreach ($data as $k => $v) {
                $this->$k = $v;
            }
        }

    }
```

SimulationModel's Build Up

```
private function __construct(){
        if(isset($_SESSION["obj"]["simulationmodel"]))
self::unserialize2($_SESSION["obj"]["simulationmodel"]);
        elseif(count(array_keys($this->DataSets))<2) {
                self::BuildUpModel();
        }
}
```

***

```
private function BuildUpModel()       {
        GLOBAL $db;

        $query="SELECT * from datasets where disabled=0 order by isderived";
        $res=$db->rq($query);
        for($i=1; $r=$db->fetch($res); $i++) {
                if($r["isderived"]==0) {
                        self::AddDataSet($r);
                } else {
                        self::AddDerivedDataSetChilds($r["id"]);
                }
        }

        $query="SELECT * from chartsandseries where disabled=0 and datasetid>0";
        $res=$db->rq($query);
        $types[1]="actual";
        $types[2]="projected";
        for($i=1; $r=$db->fetch($res); $i++) {
                $seriesnameoncharten = (isset($r["seriesnameoncharten"])) ?
$r["seriesnameoncharten"] : $types[$r["typeid"]];
                if($r["typeid"]==1)
        self::AssignSeriesToChart($r["datasetuniquename"], "actual", $seriesnameonchart,
$r["chartshortname"]);
                if($r["typeid"]==2)
        self::AssignSeriesToChart($r["datasetuniquename"], "projected",
$seriesnameonchart, $r["chartshortname"]);
                if($r["isparameter"])
        self::AssignDataSetToParamChart($r["chartshortname"], $r["datasetuniquename"]);
                if($r["isderived"])    self::AssignDerivedToChart($r["datasetuniquename"],
$r["chartshortname"]);
        }

}
```

Saving all charts' thumbnails JavaScript

```
getallchartnames();
chartstosave=Array();

function getallchartnames() {
        $.post( "/includes/ajax.php", {getallchartnames: true} ).done(function( data
) {
                chartstosave  = jQuery.parseJSON(data);
                opensaveandclose(0);
        });
}

function opensaveandclose(i)       {
        if(i>0) AddOrRemoveChart(chartstosave[i-1], 0, 0);
        if(chartstosave[i]) AddOrRemoveChart(chartstosave[i], 0, 0);
        i++;
        window.setTimeout("massivesave();", 3000);
        window.setTimeout("opensaveandclose("+i+")", 6000);
}
```

```
function massivesave()      {
       for(var key in charts) {
              if(charts[key][0]!='parameter')
       setTimeout("savechart('"+key+"');",100);
       }
}

function savechart(container)      {
       $.post("savechart.php", { filename: container,
imagedata:rendercanvas(container) }).done(function( data ) {   });
}

function rendercanvas(container)     {
    chart_area = document.getElementById(container);
    svg = $("#"+container).children().children().html();
    canvas = document.getElementById("hiddencanvas");
    canvas.setAttribute('width', chart_area.offsetWidth);
    canvas.setAttribute('height', chart_area.offsetHeight);
    canvg(canvas, svg);
    image_data_uri = canvas.toDataURL("image/png");
    canvas.setAttribute('width', 1);
    canvas.setAttribute('height', 1);
    return image_data_uri;
}
```

Exporting database for SPSS

```
function generate_csv()       {
               global $db;
               $regscript = fopen('output/regrscript.txt', 'w+');

               $imax=0;
               $csvoutput="year,";
               $columnlist=Array();
               $query="SELECT id,tablename,columnname,datasetuniquename from datasets
where frequency='monthly' and disabled=0 and isderived=0";
               $res=$db->rq($query);
               for($i=0; $r=$db->fetch($res); $i++) {
                       $csvoutput.="ds".$r["id"]."_raw,";
                       $csvoutput.="ds".$r["id"]."_1m,";    $columnlist[$i].="
ds".$r["id"]."_1m";
                       $csvoutput.="ds".$r["id"]."_3m,";    $columnlist[$i].="
ds".$r["id"]."_3m";
                       $csvoutput.="ds".$r["id"]."_12m,";   $columnlist[$i].="
ds".$r["id"]."_12m";
                       //generate_regressionscript($regscript, "ds".$r["id"]."_raw",
$r["datasetuniquename"], $columnlist);
                       $query2="SELECT year,".$r["columnname"]." from ".$r["tablename"]."
where year>'2007-01-01' order by year";
                       $res2=$db->rq($query2);
                       for($j=0; $r2=$db->fetch($res2); $j++)       {
                              $key=substr($r2["year"], 0,7);
                              $k=-1;
                              $output[$key][$i][++$k]=$r2[$r["columnname"]];

       $output[$key][$i][++$k]=0+$output[DataSet::AddYearMonthToKey($key, 0, -
1)][$i][0];

       $output[$key][$i][++$k]=0+$output[DataSet::AddYearMonthToKey($key, 0, -
3)][$i][0];

       $output[$key][$i][++$k]=0+$output[DataSet::AddYearMonthToKey($key, -1,
0)][$i][0];
                       }
               }

               $query="SELECT id,tablename,columnname,datasetuniquename from datasets
where frequency='monthly' and disabled=0 and isderived=0";
```

```
		$res=$db->rq($query);
		for($i=0; $r=$db->fetch($res); $i++) {
			if($i>0) {
				$columnlist[count($columnlist)]=$columnlist[0];
				array_shift($columnlist);
			}
			$columnlist2="";
			for($j=0; $columnlist[$j]; $j++)	{
				$columnlist2.=$columnlist[$j];
				if($j%6==5) $columnlist2.="\n";
			}
			generate_regressionscript($regscript, "ds".$r["id"]."_raw",
$r["datasetuniquename"], $columnlist2);
		}

		$csvoutput.="\n";
		$imax = ($i>$imax) ? $i-1 : $imax;
		ksort($output);


		$fp = fopen('output/file.csv', 'w+');
		foreach ($output as $key => $value) {
			$csvoutput.="$key-28,";
			for($i=0; $i<=$imax; $i++)	{
				for($j=0; $j<=$k; $j++)		{
					$csvoutput.=$output[$key][$i][$j].",";
				}
			}
			$csvoutput.="\n";
		}
		fwrite($fp, $csvoutput);
		fclose($fp);
		fclose($regscript);
}
```