

IBD Computer Science

Internal Assessment

# Multilevel Marketing Manager

Application for managing human resources

Candidate Name: **Juraj Masár**

Candidate Number: **000771–022**

Session: **May 2011**

School: **Gymnázium Jura Hronca, Bratislava, Slovakia**

Computer Science teacher: **Mrs. Eva Hanulová**

(Blank page)

## **Acknowledgments**

This work could not have been done without the support of various people.

The author wishes to thank to his friend Daniel Lovásko, who was helpful and offered constructive criticism.

Special thanks also to his family for understanding and support while this report was being created.

## Contents

Acknowledgments .....	3
A: Analysis.....	9
Introduction.....	9
Current system .....	9
Basic functionality .....	11
A1: Analyzing the problem .....	11
Distributors – permanent information.....	11
Permanent information– sample input data.....	11
Distributors - changing information .....	13
Information about distributor for particular month – sample input data .....	13
Calculations on input data.....	14
Extra information for the root distributor – the user of the application .....	16
Other features of the application.....	16
A2: Criteria for success .....	17
Common goals.....	18
A3: Prototype solution .....	19
Login screen.....	22
Main window.....	22
Addition of distributors .....	23
Profile of the distributor.....	24
B: Detailed design.....	25
B1: Data structures.....	25
Data structures stored on HDD .....	25
Password data structure .....	25
Record .....	26
DirectAccessFile.....	26
Distributor data structure .....	27
Month data structure .....	29
Complex abstract composite data structures stored in memory .....	30
Linked list.....	30
Vector .....	33
Hierarchical data structures .....	38
Tree.....	38

B2: Algorithms .....	39
class DirectAccessFile .....	39
Getting the length of the file .....	39
Getting the number of records .....	39
Seeking to a different record .....	40
Writing to record with given position .....	40
Reading from record with given position .....	41
Deleting record with given position .....	41
Appending new record to file .....	42
class Record .....	42
Merging record fields to single string and writing it to DirectAccessFile at given position .....	43
Reading from given position from DirectAccessFile and parsing String into particular record fields .....	43
class Distributor .....	44
Determine the generation of the distributor in the distributors structure .....	44
Get pointers to direct children distributors of the particular distributor .....	45
Get pointers to all children distributors of the particular distributor .....	45
class Month .....	46
Calculation of TotalPw .....	46
Calculation of TotalGw .....	46
Determining the level distributor achieved that Month .....	47
Determining the bonus level distributor achieved that Month .....	47
Calculation of provision .....	48
class Treeltem .....	49
Get number of all children distributors .....	49
Get position of current item and all its children .....	49
class Static .....	50
Customizing String length to unified length .....	50
Cut String if longer than given length .....	51
Number of occurrences of a given char in a given String .....	51
Check if all of the String characters are numeric .....	52
Remove given char from given String .....	52
Parse merged Strings with a given separator to array of String (explode) .....	53
Merge given array of Strings to single String with given separator (implode) .....	53

Conjunction of more given Integer arrays .....	54
E-mail validation .....	54
class List.....	55
Add item to the beginning of the List.....	55
Add item to the sorted List.....	55
Find position by given key in list.....	56
Find positions of all items with key of which given key is substring in a list.....	57
class Vector<O> .....	57
Push front .....	58
Pop front.....	58
Insert given object at given position in the Vector .....	58
Find the index of object given in the vector.....	59
class Dialog .....	60
Display profile of the particular distributor.....	60
class DistributorManager .....	61
Add new distributor .....	61
Get all distributors.....	61
Find distributors by combination of their name, surname and registration number.....	62
class MonthManager.....	63
Get all months for particular user .....	63
Get particular Month object by its year and month combination .....	63
B3: Modular Organization .....	65
C: The Program .....	71
C1: Source code.....	71
File readme.txt .....	71
File Static.java.....	72
File Dialog.java.....	89
File Main.java .....	97
File TextFileWriter.java.....	102
File TextFileReader.java.....	105
File DirectAccessFile.java.....	108
File DesEncrypter.java .....	117
File DistributorManager.java.....	121
File MonthManager.java .....	142

File Password.java .....	151
File Record.java .....	155
File Month.java .....	160
File Distributor.java .....	174
File Vector.java .....	190
File VectorItem.java .....	208
File List.java .....	212
File Item.java .....	227
File ItemRegNum.java .....	231
File ItemSponsor.java .....	233
File TreeItem.java .....	235
File MainWindow.java .....	243
File ProfileWindow.java .....	254
File DistributorTreeDialog.java .....	275
File ChangePasswordDialog.java .....	280
File AddDistributorDialog.java .....	285
File PointsDialog.java .....	297
File ChildrenTreeDialog.java .....	304
File ChildrenTableDialog.java .....	308
File CheckNode.java .....	312
File CheckRenderer.java .....	316
File NodeSelectionListener.java .....	320
File TreeLabel.java .....	322
File RowHeaderRenderer.java .....	327
File NonEditableDefaultTableModel.java .....	329
File Chart.java .....	331
File PlainDocumentMaxLength.java .....	336
File JTextFieldMaxLength.java .....	338
File CustomPasswordField.java .....	340
C: Usability .....	342
C2: Handling errors .....	342
Missing files .....	342
Erroneous/invalid data input .....	343
C3: Success of the program .....	343

D: Documentation .....	345
D1: Hardcopy of program output.....	345
First run of the program .....	345
Logging in to the system.....	346
Main window of the application .....	346
About dialog .....	348
Change password dialog.....	348
Add distributor dialog.....	349
Points management .....	351
Distributor profile .....	353
Excerpt from the program's data file .....	359
D2: Evaluation .....	360
Mastery .....	362



## **A: Analysis**

### **Introduction**

Multilevel marketing is a relatively new concept of entrepreneurship which has become very popular recently, because of its high accessibility. It allows anybody to start their business practically without any starting capital. It offers an alternative approach for a transport of goods and services from producer to consumer.

The classical way when goods move from producer to wholesale, then from wholesale to retail business and consequently to consumer, spends considerable resources on transportation costs and advertisement.

The multilevel marketing, on the other hand, takes these resources which would be spend on advertising, and gives them to “distributors” – salesmen who sale goods directly to the consumer.

The amount of earnings in multilevel marketing systems, therefore, depends on the number of sales the individual has accomplished. However, systems offer a way of building a passive income of money, too. If the distributor registers someone else to the system, they get a bonus every time the newly registered distributor makes a deal and when anybody registered under them makes a deal – according to certain rules.

This is why the managing of human resources is crucial in this kind of business.

I will develop the program directly for the distributor – my mother. It will handle the whole matrix of people involved in the system under particular distributor. It will store all required information about each distributor, the amount of sales each distributor has made and calculate their earnings. Thanks to the system the user will be able to see whole tree of distributors registered under them and easily work with them if necessary.

Even though the principles in the multilevel marketing are more or less similar for every system, each system has its specifics. I will be developing the application for a **Network World Alliance** or **NWA**, a relatively new player on MLM field.

### **Current system**

All information about distributors is currently stored in spreadsheet file. Even though handling the database of people in spreadsheet documents is possible, but it becomes highly impractical as the database grows. The disadvantages are clear: Limited options for graphical representations of data, impractical access to tree-style structure of the database, strictly limited organization of data.

Those are the reasons why a need for new system exists.

Number	Name	Surname	Company	Registration number	Sponsor
1.	Marianna	[blurred]	[blurred]	161	[blurred]
2.	Jozef	[blurred]	[blurred]	161	[blurred]
3.	Peter	[blurred]	[blurred]	161	[blurred]
4.	Viera	[blurred]	[blurred]	161	[blurred]
5.	Mária	[blurred]	[blurred]	161	[blurred]
6.	Jozef	[blurred]	[blurred]	186	[blurred]
7.	Peter	[blurred]	[blurred]	187	[blurred]
8.	Katarína	[blurred]	[blurred]	183	[blurred]
9.	Zdenka	[blurred]	[blurred]	186	[blurred]
10.	Eva	[blurred]	[blurred]	186	[blurred]
11.	Ľuboslava	[blurred]	[blurred]	199	[blurred]
12.	Renáta	[blurred]	[blurred]	187	[blurred]
13.	Margita	[blurred]	[blurred]	187	[blurred]
14.	Jaroslav	[blurred]	[blurred]	181	[blurred]
15.	Anna	[blurred]	[blurred]	181	[blurred]
16.	Anna	[blurred]	[blurred]	181	[blurred]
17.	Oľga	[blurred]	[blurred]	188	[blurred]
18.	Zita	[blurred]	[blurred]	184	[blurred]
19.	Anna	[blurred]	[blurred]	188	[blurred]
20.	František	[blurred]	[blurred]	186	[blurred]
21.	Jaroslava	[blurred]	[blurred]	186	[blurred]
22.	Mária	[blurred]	[blurred]	183	[blurred]
23.	Mária	[blurred]	[blurred]	183	[blurred]
24.	Eduard	[blurred]	[blurred]	183	[blurred]
25.	Martin	[blurred]	[blurred]	186	[blurred]
26.	Monika	[blurred]	[blurred]	184	[blurred]
27.	Anton	[blurred]	[blurred]	184	[blurred]
28.	Juraj	[blurred]	[blurred]	184	[blurred]
29.	Eva	[blurred]	[blurred]	187	[blurred]
30.	Andrea	[blurred]	[blurred]	191	[blurred]
31.	Eva	[blurred]	[blurred]	191	[blurred]
32.	Antónia	[blurred]	[blurred]	196	[blurred]
33.	Juraj	[blurred]	[blurred]	[blurred]	[blurred]

Figure 1: Previous system - spreadsheet document

## Basic functionality

Since the program is meant to work with personal data, the access has to be **secured by password**.

The most fundamental entity for the whole program is **distributor**. There is number of information stored about each and every distributor; some are **permanent** (name, registration number,...) whereas other **change every month** (points earned, estimated revenue, etc). Every distributor in the system has to be **registered under other distributor** – except for the user of the application – this creates a **tree-like structure** of the database of distributors.

The application is then able to produce various kinds of statistics for each distributor and their sub-network of distributors. When a new distributor is registered in the network, the user of the application adds them to the system. After month ends, the user of the application inserts data (points earned, etc) about each and every distributor to the system.

## A1: Analyzing the problem

As I mentioned previously, there is a number of information which is to be stored by the program. Therefore, the application has to work with a database; according to an interview with the user, we will distinguish **permanent information** and **information different for every month**.

All input information is to be inserted to the application through user-friendly interface with validation. Simple distributor entry should look like following:

### Distributors – permanent information

#### Permanent information– sample input data

Sponsor*	Registration number*	Country*	Date of registration*	Status	Name*	Surname*	Date of birth	E-mail	Telephone	Note
-	10000	SK	1/10/2010	Manager, Salesman, Consumer	John	Cash	1/1/1980	john@cash.com	00421907123123	The famous singer!
10000	10050	SK	10/10/2010	Salesman	Bon	Jovi	2/12/1975	bon@jovi.net	004412332112	
10000	10100	CZ	11/10/2010	Manager	Amy	Mac	7/5/1970	amy@gmail.com	00441221221221	
10050	10200	SK	20/10/2010	Consumer	Petr	Bily	10/11/1965	petr@pobox.sk	00421264281231	
10050	10300	SK	25/10/2010	Salesman, Consumer	Jozo	Raz	12/11/1960	raz@elan.sk	00421123321122	

\* is compulsory, the rest is optional

**Registration number (compulsory)** is a *unique identification number* each distributor registered in the system possess – it is assigned by the particular multilevel marketing company. Example: 10000, 10500.

**Sponsor (compulsory)** is a registration number of the distributor under the particular distributor is registered. Except of the *root distributor* in the system – the user of my application – every user has to have a sponsor. Example: 10000, 10500.

**Country (compulsory)** refers to *country code* of the country the distributor is registered in in the system. Example: SK, CZ, HU.

**Date of registration (compulsory)** is a date in which the user was registered to the multilevel marketing system (not added to the application!) Example: 1/1/2010

**Status (optional)** is a reason why individual distributor decided to join the multilevel marketing company. It is not official information; it has just an informative character for the user of the application. Every distributor can have any of the following statuses: *Manager* – the distributor is interested in registering other people to the system, because of provisions; *Salesman* – distributor wants to directly sell goods and services to customers; *Consumer* – distributor themselves want to consume goods offered by the multilevel marketing company and they registered in order to get better prices. The user wants this feature to be implemented using checkboxes.

**Name (compulsory)** is the name of the distributor. Example: John, Brittany, Bridget.

**Surname (compulsory)** is the family name of the distributor. Example: West, Hemmingway, Jewell.

**Date of birth (optional)** of the distributor. Example: 10/12/1980

**E-mail (optional)** is a valid e-mail address of the distributor. Example: west@annie.com

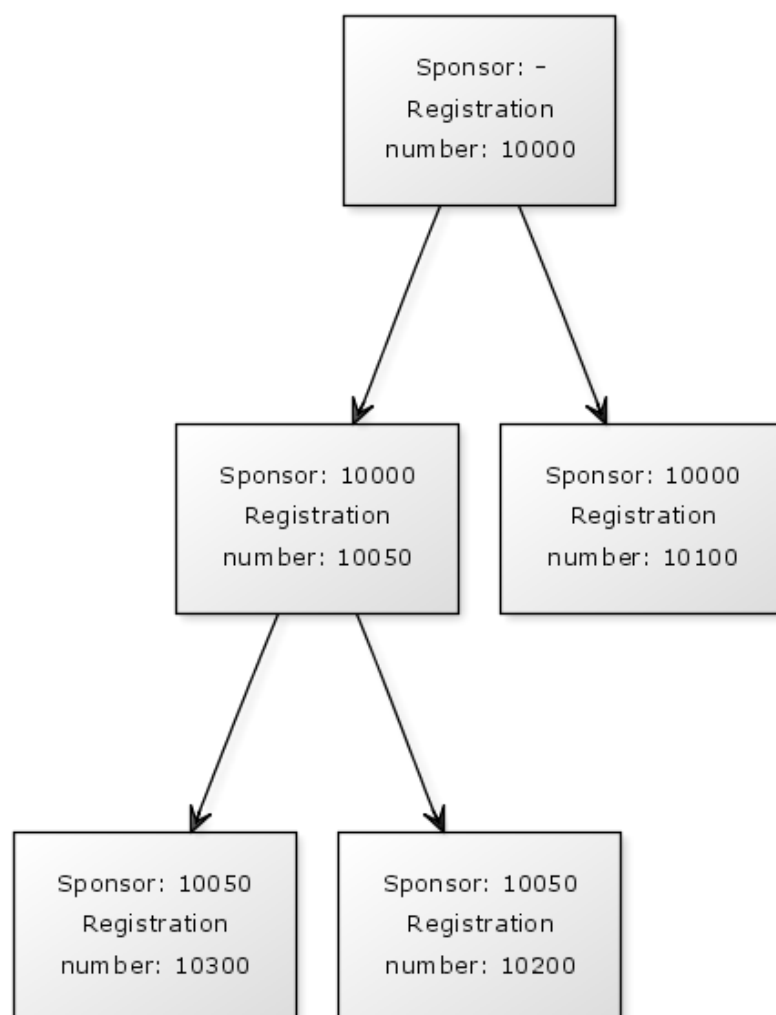


Figure 2: Tree-like structure of distributors in MLM system

**Telephone (optional)** is a phone number of the distributor: Example: 00421907321123

**Note (optional)** is meant to be an optional multiline text field suitable for any other notes about particular distributor.

### Distributors - changing information

In addition data already mentioned there is other information required to be stored by the application, which is different for every month. It is typically related to the amount of money earned by particular distributor in particular month.

This functionality should work the principle that if user of the application wants to calculate earnings and keep track of earned points of particular distributor, they firstly select these distributors using checkboxes. Since the calculation of earnings depend on data of the direct child-distributors (those whom sponsor is the distributor whose earnings the user wants to calculate), the program then generates a table with names of all distributors which has to be fulfilled with data as following example:

#### Information about distributor for particular month – sample input data

Registration number	PW	Group-PW	GW	Group-GW
10000	0	1200	0	597
10050	100	800	47	396
10100	300	0	146	0

**PW**<sup>1</sup> and **GW**<sup>2</sup> are different types of points the distributor receives from the multilevel marketing company. The difference between them is out of our interest; the only thing we have to know is that we will some *calculations* based this information.

**Group-PW** is the sum of PWs of all “children” of particular distributor – all distributors which are in the structure below particular distributor (not only directly).

**Group-GW** is the sum of GWs of all “children” of particular distributor – all distributors which are in the structure below particular distributor (not only directly).

Even though I was not eager to input Group-PW and Group-GW I have changed my mind after a dialogue with the user. The thing is that Group-PW could be easily calculated by the program itself if PW of every distributor in the application was known. However, it could be very time consuming for

---

<sup>1</sup> **PW** is earned as following: for every sale of good which the final price (including provision and tax) of N €, the distributor gains 2N PWs (1€ = 2 PW).

<sup>2</sup> **GW** is the net turnover the distributor has made (without provision and tax) in €.

the user to insert PWs of all distributors every month, which would be necessary even if the user is interested only in statistics of points of particular distributors.

That is why we agreed on alternative approach – it will be possible to insert Group-PW and Group-GW every month for every distributor as well as PW and GW, even though this information is redundant if PWs and GWs for all distributors are known. This will, however, allow calculating earnings even only for a small part of all distributors the user is interested in. In statistics the program will simply display N/A (not applicable) for the rest of distributors for which points are not known.

Therefore, when inserting data for particular month, **user does not have to insert for all distributors**; only necessary information about some distributors is required – according to user's wants.

### Calculations on input data

Firstly, the user wants the program to calculate **Total-PW**, which is the sum of the distributor's PW and Group-PW.

**Total-GW** is defined in a very similar way, as the sum the distributor's GW and Group-GW.

Secondly, the program should be able to determine the **Level** for the distributor. The level is a percentage of the distributor's net turnover (=GW) which will be paid to the distributor from multilevel marketing company (however, the calculation of the provision itself is a little bit trickier than this). Level depends on the Total-PW of the distributor and can be determined from the following table:

Total-PW	Level
2 – 499	3%
500 – 999	6%
1000 – 1999	9%
2000 – 3999	12%
4000 – 7999	15%
8000 – 11999	18%
12000 and more	21%

There is one more way of rewarding distributors in the NWA multilevel marketing system, besides the provision from the distributor's net turnover. It is called the "bonus from differences" and it refers to provision earned from child-distributors registered directly below particular distributor.

In order to illustrate it painlessly, let's work with data we have from input – so we need now to calculate all information we have already introduced.

Registration number	PW	Group-PW	GW	Group-GW	Total -PW	Total-GW	Level
10000	0	1200	0	597	1200	1797	9%
10050	100	800	47	396	900	443	6%
10100	300	0	146	0	300	146	3%

It works as following: firstly, it is required to determine the Level of each of these child-distributors. Then, for every one of them, the provision from particular child-distributor is calculated as **the level difference** (=the difference between distributor's level and their child-distributor level) \* **Total-GW of the child-distributor** (Total-GW represents the net turnover of the whole branch of distributors registered under the child-distributor we are dealing with and child-distributor's own GW).<sup>3</sup> The bonus from differences is therefore the sum of all such provisions from all direct child-distributors.

Example: The "bonus from differences" for distributor 10000 is calculated as following:

$$(9-6) \% \text{ from } 443 + (9-3) \% \text{ from } 146 = 0.03 * 443 + 0.06 * 146 = 22 \text{ €}.$$

There are some further marginal cases, however. If the level difference is 0 and the distributor's level is less than 21%, there is no provision from particular child-distributor.

If the level difference is 0 and the distributor's level is 21%, there is a provision from such child-distributor and it is determined as following:

If the distributor whose earnings we are interested in has more than 3 child-distributors with level of 21%, the **Bonus Level** is determined from the following table.

Number of direct child-distributors on Level 21%	Bonus Level
1	7.0%
2	7.5%
4	8.0%
6	8.5%
8	9.0%
10	9.5%
12 and more	10.0%

If there is 1 to 3 child-distributors with level of 21% and distributor's **Total-PW – Total-PW's of all 21% child-distributors** is more than 6000, the bonus level is determined from the previous table as well.

<sup>3</sup> Level difference and only positive or zero, because the distributor's level is always at least as high as their child-distributor's level – because of the way the level is determined.

If there is 1 to 3 child-distributors with level of 21% and distributor's **Total-PW – Total-PW's of all 21% child-distributors** is more than 4000 but less than 6000, the bonus level is simply 3%.

If there is 1 to 3 child-distributors with level of 21% and distributor's **Total-PW – Total-PW's of all 21% child-distributors** is less than 4000, the bonus level is simply 0%.

So the last part of provision is the "bonus from differences from child-distributors with level of 21%" which can be calculated as **Bonus Level** \*(sum of Total-GWs of all child-distributors with level of 21%).

Finally, the application calculates **provision**. It can be calculated upon previous information as

**Level \* (GW of the distributor) + "the bonus from differences" + "the bonus from child-distributors with level of 21%"**

### **Extra information for the root distributor – the user of the application**

There is some additional information, however. The user wants to plan their achievements in the multilevel marketing system as well. Therefore, the root distributor is special, because there is extra data.

The user of the application should be able to insert the **Planned-PW** and **Planned-Group-PW** to the record of the root distributor – points for that particular month they aspire to earn.

Moreover, the user should be able to insert the **planned number of new registrations** to the record of root distributor, which is the planned number of the new children of the root distributor in the MLM system in comparison to the previous month.

### **Other features of the application**

The program should provide easy-to-use interface for searching distributors. The user wants to **search** for distributors **according** to their **registration number, name, surname**.

It should be also able to display a **profile** for each and every distributor. Such profile would display all **permanent information** about the distributor as well as **calculated data** for months, if applicable.

For every month, it would display the **newly registered distributors** both directly<sup>4</sup> and indirectly<sup>5</sup>; **table of points** for past months and **graphical representation** of these data.

If it would be the profile of the root distributor, the user of the application, it would also display the **planned data** for every month.

Last but not least, it would be possible to **view all distributors which are registered below** the particular distributor – **in a table** with various sorting options and filters; and **in tree-like structure** according to who is whom sponsor.

---

<sup>4</sup> Directly – the child-distributor's sponsor is the distributor whose profile is displayed

<sup>5</sup> Indirectly – the child-distributor is below the distributor whose profile is displayed in the structure but they do not have to be directly their sponsor



## A2: Criteria for success

Upon multiple dialogs with user the set of following criteria has been created:

- 1. Program should intuitive and suitable for using by non-skilled PC user. It works on Microsoft Windows XP and newer.**

Application should be as simple and straight-forward as possible. It should work on the operating system which the user currently uses and support also newer versions in case of future updates of the operating system.

- 2. Program is supposed to work with private data – the access to all information should be protected**

Since the application is designed only for a single user, a password authentication without username is suitable. The program, however, has to therefore offer an additional functionality to change the password.

- 3. Inserting new distributors to the system should be painless and intuitive with proper validation**

Application should include an easy-to-use interface for inserting information about new distributors to the system. This interface should offer quick addition of many distributors, so that inserting of distributors will not be time consuming.

- 4. Inserting points earned by distributors to the system should be painless and intuitive with proper validation**

The same as written above applies also for inserting data about distributors every month. The interface has to be designed the way that inserting points for every month will become an effortless routine for the user of the application. The user is not forced to insert this data for every distributor in the system; however, if they choose to insert data for particular distributor, they have to insert all types of data (PW, Group-PW and GW).

- 5. Application should allow an intelligent search for distributors by various properties**

The program should offer an efficient searching functionality which would be able to operate even with no precise input – the search is supposed to be case-insensitive, support search both with diacritic and without marks (when searching for “Masar” and the actual surname of the distributor is “Masár”, the program is supposed to find him without problems). It should be possible to search by several properties – the registration number, name, surname and number of points.

- 6. Profile of each distributor should display well-arranged material about the distributor and allows the user to immediately access all important information**

- 7. Application should offer multiple ways of finding out the changes in productivity of the distributor (tabulated data, graphs)**

Graph of changes in points should make picturing of the progress of individuals easier. The tables, on the other hand, are supposed to offer various functionality; sorting by different columns, filtering.

**8. The program should offer an easy-to-use interface for displaying and browsing the structure of distributors registered below any distributor**

Actually, user requested two ways of browsing the structure of distributors registered below any distributor: using a table (with sorting and filtering features) and using dynamic tree-like component (known from left panel of Microsoft Windows Explorer).

**Common goals**

- Program should be easy to work with
- Interface should be designed intuitively
- Program should deal with data efficiently during calculations (both memory and CPU consumption should be minimal)
  - The number of entries in database should be limited only by the limits of operating system
- Data should be stored on HDD efficiently

### A3: Prototype solution

The following flowchart represents the basic functionality the user agreed on.

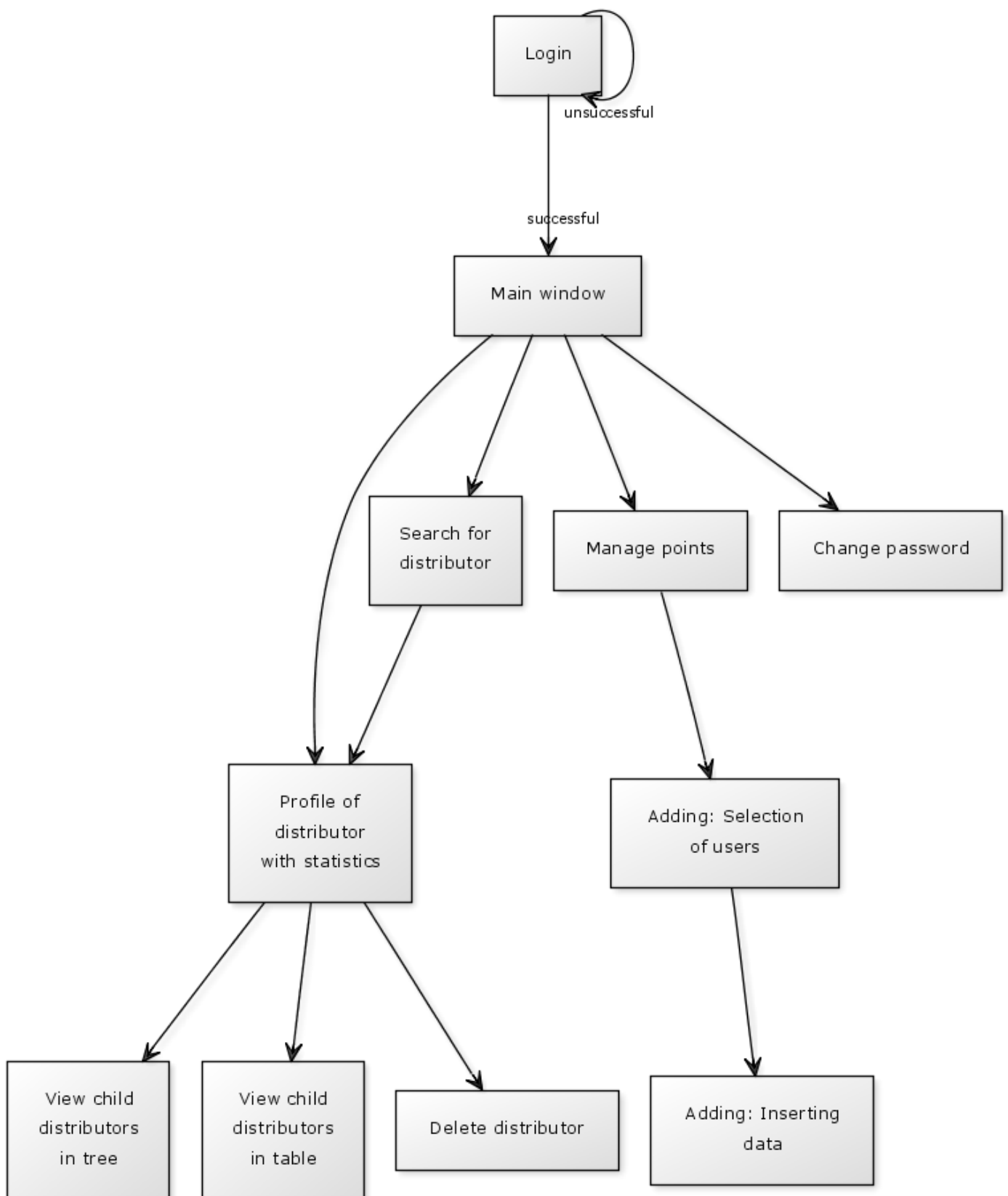


Figure 3: Flowchart describing functionality of the application

During multiple talks with the user we have also produced several sketches of the program interface using pen and pencil. I have consequently designed the interface using Java swing library, which I will be using in producing of the interface for the application so that the user could have a quite precise picture of the program at this point.

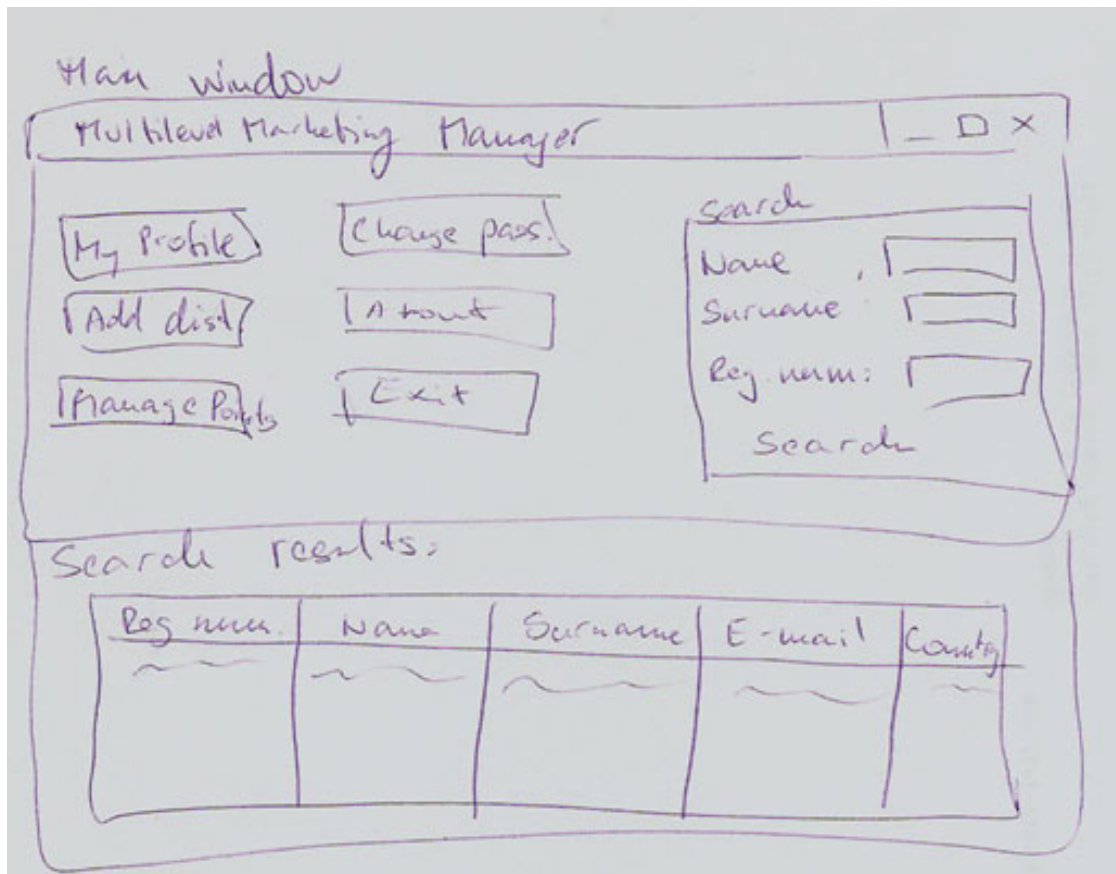


Figure 4: Initial drawing of the main window of the application done during interview with user

Profile Window

Profile of Juraj Masár

Name:  Surname:  Date of birth:  E-mail:  Telephone:  Note:

leg name:  country:  Date of regist.:  Generation:  sponsor:

Save Delete OPEN TREE OPEN TABLE

Statistics

Months

Criteria

Chart

Figure 5: Initial drawing of the profile window of a distributor done during interview with the user

I will now show the final version of the most important windows designed for particular tasks one by one.

## Login screen

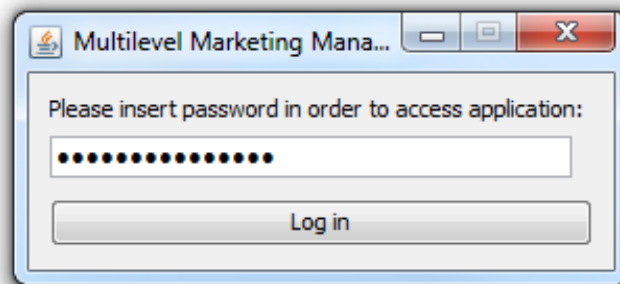


Figure 6: Logging into the application

If the user inserts the correct password the main windows of the application appears. Otherwise, a graphical message informs them about the unsuccessful login.

## Main window

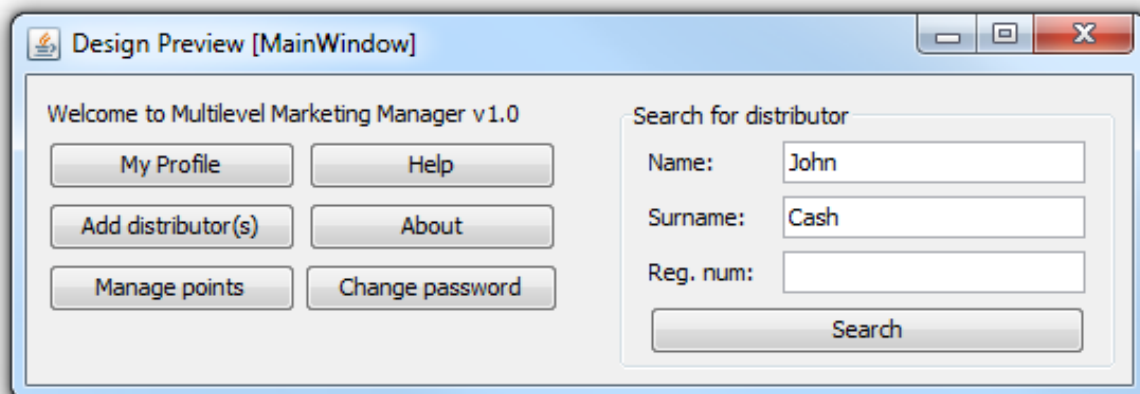


Figure 7: Main windows of the application

Main window offers direct access to most of the application's functionality: direct access to profile of root distributor using "My Profile", accessing dialogs for addition of new distributors to the system, managing points, changing password and other.

Notably, there is a dialog for distributor searching. If the user fulfills the details and presses "Search" button, the window is enlarged and search results are displayed comfortably directly under the menu.

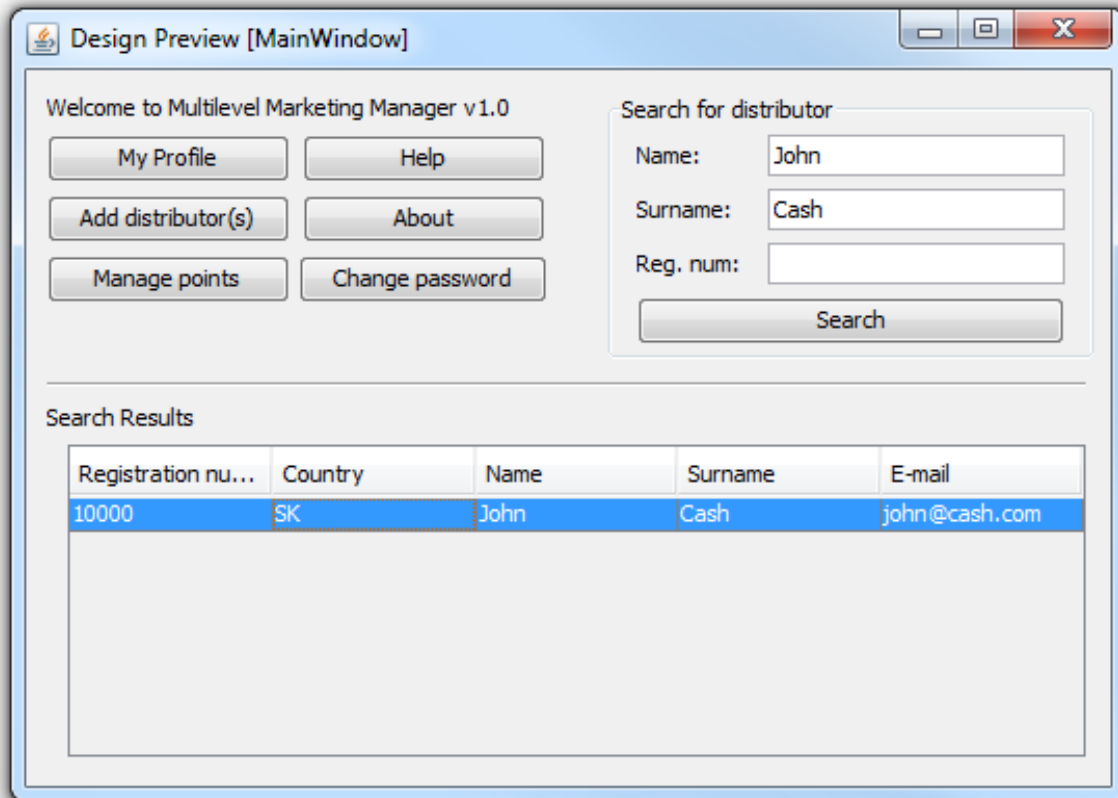


Figure 8: Main window of the application with displayed search results

Double clicking on entry in search results opens the profile window of particular distributor.

### Addition of distributors

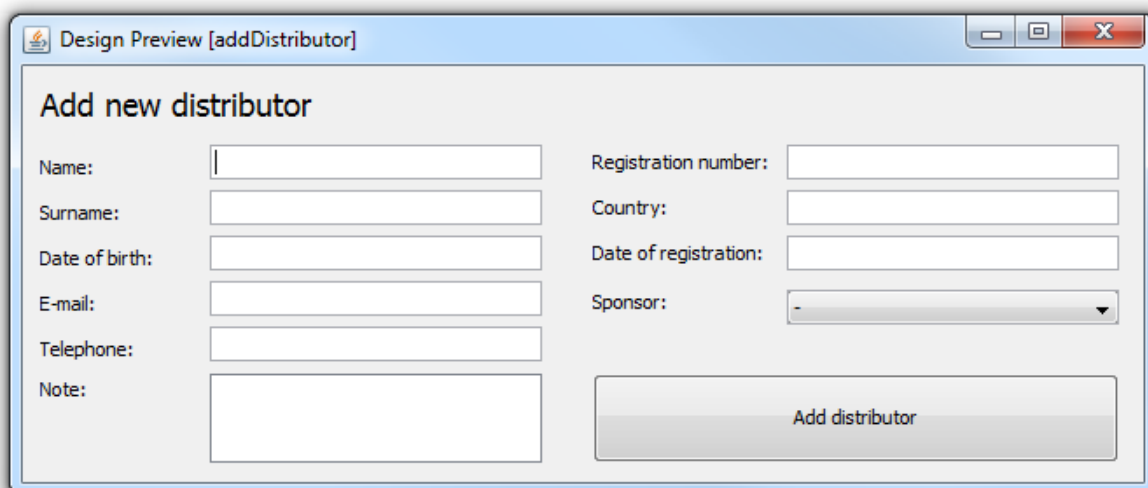


Figure 9: Dialogue for adding distributors to the system

Using this dialogue the users inserts the basic information about the distributor. They also select the distributor's sponsor using Combo Box (or pull-down menu) in which all distributors which are currently in the system appear.

## Profile of the distributor

Design Preview [Profile]

### Profile of John Cash

Name:	John	Registration number:	10000
Surname:	Cash	Country:	SK
Date of birth:	1. 1. 1980	Date of registration:	1. 10. 2010
E-mail:	john@cash.com	Generation:	0
Telephone:	00421907123123	Sponsor:	-
Note:			

Save changes
Open child distributors in table
Delete distributor
Open child distributors in tree

### Statistics

	May	June	July	August	September	October
Total distributors	-	-	-	-	-	4
New distributor...	-	-	-	-	-	2
New distributors	-	-	-	-	-	4
PW	-	-	-	-	-	0
Group-PW	-	-	-	-	-	1200
GW	-	-	-	-	-	0
Group-GW	-	-	-	-	-	597
Provision	-	-	-	-	-	22

Legend:

- Total distributors
- New direct distributors
- New distributors
- PW
- Group-PW
- GW
- Group-GW
- Provision

Figure 10: Profile of the distributor with comprehensive information about their performance in both tabular and graphic form.

User of the application can instantly update any personal data of the distributor directly in the profile, view structure of distributors registered under particular distributor in both tree-style view and tabular view and see the performance results of the distributor for past months.



## B: Detailed design

### B1: Data structures

#### Data structures stored on HDD

Firstly, I had to identify the entities using which the program will work. This was not a difficult task, since because of the clarity of information provided by user in the Analysis stage, it was straightforward that storing of two main entities will be required – **distributor** and **month**. Moreover, program will also have to take care of storing the **password** required from user for its launch.

#### Password data structure

When ruminating over the appropriate type of a data structure or a file, it is important to bear in mind the whole picture of implementation of particular functionality. For instance, when talking about storing password, we need to know that we will be storing not the password inserted by user directly, but its hash instead.

Therefore, we can rely on facts that each hash will have a constant length and form – according to hashing algorithm used. Since the length is constant, in the end we are storing only one line of characters. Such file will always be read from the beginning till the end and therefore, no direct access is required.

So a simple access to **sequential file** is enough to store and read/check the password. This reading and writing within sequential file will be done using classes similar to IBIO class which will encapsulate the Java's stream and buffer classes – `TextFileWriter` and `TextFileReader`.

The simple logic which will work with these general classes for working with sequential file will be in the class **Password**.

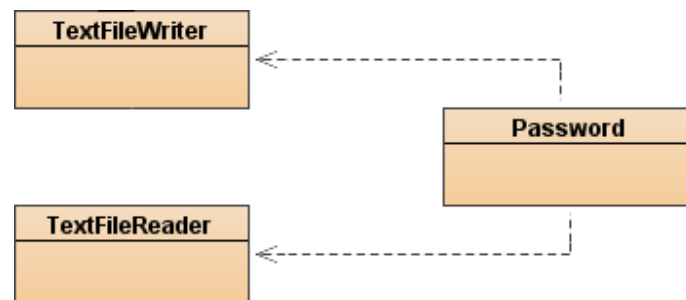


Figure 11: Password data structure organization

The organization of the password data structure is illustrated in the following picture.

## Record

Lots of information the application is supposed to work with has a modular character. Instead of working with each of the fields of particular entity (e. i. working with distributor name and surname separately), it is practical to organize this information into one data structure and operate with all information concerning particular distributor together.

Class which will be operating direct access to file will be given a record about the distributor as a whole. It will be a String constituting of other Strings (particular fields of distributor record) separated by a separator – chosen character (in my case ;).

The example of a String which has been created by compiling all the records fields of a record is shown below:

```
10002;10001;SK;3.12.2010;ABC2;name 2;surname  
2;17.11.1957;2@nwaobchod.sk;0906248216;note about 2;
```

The Record class will have to take care of creating this single String out of all the record fields of the entity before passing it to DirectAccessFile class, and parsing the single String after being read from the file to particular object variables.

The Record class itself is meant to be abstract. It will contain a single object variable – the array of Strings data[], which is supposed to store values of individual fields of the record.

When the Record is extended getters and setters for particular indexes are to be created, therefore, the programmer does not have to bear in mind particular indexes and the integrity of data stored in the String[] is controlled by the Record class only.

Last but not least, we have to still intensively think of the user requirements, even of those which were not stated directly – because the user typically does not understand the technical issues connected with software design.

Since the user wants the application to be protected by a chosen password, we can assume they want **all data to be protected** – secured. Password authentication for the application entrance is one thing, however, if data should be protected they can't be in stored in files (does not matter whether DAF or sequential file) in a plaintext. They have to be somehow encrypted.

If the data structure organization is designed with this in mind, the implementation of encryption of all information in the database can be painless – it will be implemented at the lowest level – directly in the Record class. Therefore, the rest of the application does not have to cope with the idea of encrypting information at all.

## DirectAccessFile

This DirectAccessFile class will encapsulate the RandomAccessFile class from Java, providing the necessary operations with RandomAccessFiles. It will be working with records without distinguishing particular record fields within the records. It will support low-level data operation, such as adding new record to a file, searching existing records, editing existing records and deleting existing records. Since its nature will be general to a great extent, it will be easily reused for all entities which require direct access to files.

DirectAccessFile this application will work with will not be ordered. Therefore, **adding new record** to the DirectAccessFile is a simple matter of prolonging the file and inserting the newly created record at the end of the file.

Seeking through the file will be done using the **seek** method of the RandomAccessFile class. Each instance of DirectAccessFile will be given a record length in the constructor's parameters. This record length will be stored as an object variable of the DirectAccessFile instance. It will allow us to seek the file not by the number of bytes, but more simply – just by record numbers. Particular bytes will be calculated upon the record number and the record length in bytes of a single record.

Reading will be done using the method **readUTF** of the RandomAccessFile with catching the potential I/O exceptions. The String we get is then parsed so that the individual record fields of the record are directly accessible. It will have constant, e. i.  $O(1)$  time complexity.

Editing the particular record will be, on the other hand, done by overwriting particular record by record with updated information. Writing will be done using **writeUTF** method. This method will write the content of the String compiled by all the values of the individual record fields (the whole data array). This String will be customized to have a unified record length, so that the integrity of Random Access File in which the record will be stored is preserved. It will have constant, e. i.  $O(1)$  time complexity.

**Deleting the record** in a DirectAccessFile will work as following. Since the file is not ordered, the last record will be read to memory and it will overwrite the record which is to be deleted. Then, the file will be shortened using the method **setLength**. Thanks to this approach, the deleting record on a known position will have a constant time complexity  $O(1)$ .

### **Distributor data structure**

Storing distributor information has to be much more sophisticated than storing a simple password. Information about each distributor has a modular character, which has been already described in the stage A – columns of table of distributors.

Since many actions will be done on this data, such as adding new data, finding data and editing existing data and removing existing data a direct access to particular records = information about particular distributor, has to be possible. That is the reason why the Distributor data structure will extend the generic Record data structure and implement all its properties.

### ***Distributor record***

<b>Name</b>	<b>Type</b>	<b>Validation</b>	<b>Example</b>
<b>Registration number</b>	String	Required, can contain only numeric characters. Unique. Length <= 10 characters	10000
<b>Sponsor</b>	String	Required, can contain only numeric characters. Length <= 10 characters	10001
<b>Country code</b>	String	Required, Length <= 3 characters	SK
<b>Registration date</b>	String	Required, has to be valid in format dd.MM.yyyy	10.10.2010
<b>Status</b>	String	Length <= 5 Each status displayed as checkbox is represented by a unique character	CSM
<b>Name</b>	String	Required Length <= 20	Juraj
<b>Surname</b>	String	Required Length <= 20	Masar
<b>Date of birth</b>	String	Required, has to be valid in format dd.MM.yyyy	23.06.1992
<b>Email</b>	String	Valid email address – presence of @ and tld	juraj.masar@gmail.com
<b>Telephone</b>	String		00 421 907 123 123
<b>Note</b>	String		The computer guy.

These fields describing aspects of a distributor entity will be grouped to a record and saved to a DirectAccessFile (DAF) as a record. Each distributor has their **registration number which is unique** and could be, therefore, used as a **key** when working with distributors.

Records of distributors will be stored in the following file:

```
{programDir}/data/distributors.dat
```

Each Distributor object will also have its own pointer to MonthManager, an interface for managing Month records for particular Distributors. I will explain this more thoroughly in the B3 section, since it is more related to the modular organization of the application than to data structures.

Last but not least, each and every Distributor object will also have a pointer to its corresponding Treeltem, an object which helps to represent the hierarchical structure in the MLM network in the program. I will write additional few words about Treeltem in this section.

### Month data structure

Working with month entities will be very similar to working with distributor entities. Each month for particular user will be represented by a month record stored in a file containing month records for particular distributor. Therefore, **a different file containing month records will exist for every distributor**. Each month record has pointer to the Distributor object it belongs to. The month entity has been well described in the analysis stage.

#### Month record

Name	Type	Validation	Example
Year	String	Required, valid year in format yyyy.	2011
Month	String	Required, valid month in format MM.	01
PW	String	Required, can contain only numeric characters. The integer value has to be positive. Length is not limited.	1005
Group PW	String	Required, can contain only numeric characters. The integer value has to be positive. Length is not limited.	1254
GW	String	Required, can contain only numeric characters. The integer value has to be positive. Length is not limited.	200
Group GW	String	Required, can contain only numeric characters. The integer value has to be positive. Length is not limited.	54

There is no need for Month record to include the distributor registration number since in particular file all records will belong to a single distributor.

The file of Month records for particular distributor will be

`{programDir}/data/dist{registration number}.dat`

The cooperation between Month entity, Distributor entity, general Record class and DirectAccessFile is illustrated in the following diagram.

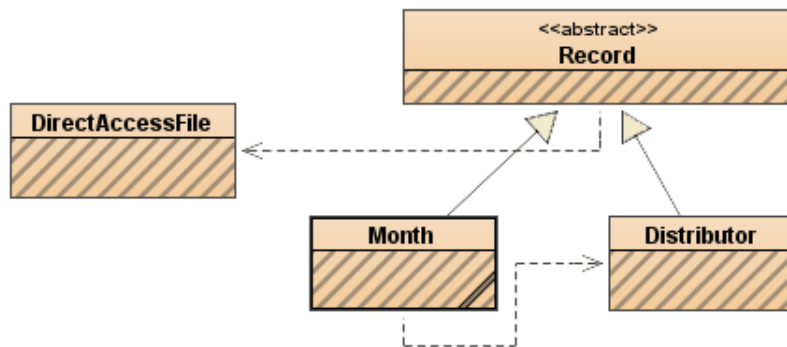


Figure 12: File entities organization

### Complex abstract composite data structures stored in memory

Additional data structures are, however, required for data manipulation within the application. An array is not sufficient in many cases because of its static nature and lack of advanced functions. Therefore, I will have to implement my own data structures with support of dynamic allocation of memory.

#### Linked list

Firstly an apparent need for the simplest dynamic data structure exists – a linked list. Because a single Distributor or Month record usually contains many data, certain actions with this data such as searching for record with specific information in a key field become very time consuming.

Consider the following example. A simple database program wants to display the whole content of its database – let's say a list of Distributors, sorted by their unique identification, in our case, the registration number.

If the only data structures providing working with this data would be those mentioned previously (DirectAccessFile, general Record and customized extended Record), the program would have to load the whole database to the memory, sort it and display it. In case it would be too large to fit the memory, it would have to go through the whole file many times, searching for the nearest higher value, which would be critically time consuming.

A simple solution which solves this issue is using a simple dynamic data structure for indexing all the records in the file by appropriate key. Each key would have the position of the records it is representing associated with it. Therefore, it would be possible to sort this partial data by key (working only with the minimum amount of data possible) and consequently access records directly using DAF from their positions.

Simple data structure which fits these needs is called **Linked list**.

**Linked list** is a composite data structure which consists of *Items*. I will firstly describe what an Item is.

An **item** is representing single record in a file. Therefore, variables of an Item have corresponding variables.

Name of variable	Type	Meaning
Key	String	Contains information we want to work in the list with. Example: an unique ID of the record according to which we want to sort records
Next	Pointer to Item	Pointer to next Item object in the List. This allows dynamic allocation of a memory and easy reordering of Items. However, it prolongs the access time for Nth object in a list from $O(1)$ in array to $O(N)$ , since all previous records have to be accessed since the content of Nth Item is accessible.
Position	Long	Contains the position of the record represented by this Item in the file. Long is used instead of Integer so that storing higher values (and therefore having more records in a single file) is possible.

**Linked list** uses Items and provides its functionality in a way that the existence of Items is not apparent outside of the Linked list object.

Its variables are as following:

Name of the variable	Type	Meaning
First	Pointer to Item	It provides access to the first Item in the "chain" of Items.
sorted	Boolean	Indicates whether the chain of linked Items is sorted or not.

The linked list typically offers the following methods:

Name of the method	Meaning	Return type	Time complexity
<b>&lt;constructor&gt;(isSorted)</b>	Initializes the object variables and sets the sorted indicator	N/A	O(1)
<b>IsSorted</b>	Returns whether array is sorted or not	Boolean	O(1)
<b>Add (key, parameter)</b>	Adds a new object to the Linked list structure by creating appropriate Item and linking it to existing Items	Boolean (true if successful)/void	In unsorted Linked list: O(1) In sorted Linked list: O(n)
<b>getPosition(key)</b>	Returns the position of the record which contains the given key	Long = position of the record	O(n)
<b>getKey(position)</b>	Returns the key of the record according to its position	String = key of the record	O(1)
<b>Delete (key)</b> <b>Delete(position)</b>	Deletes corresponding Item in the Items' chain	Boolean (true if successful) / void	O (n)
<b>getPositionBySubstring(key)</b>	The same as getPosition, however, the given key does not have to be equal to key of the Item, it can be its substring = we need this for implementing search which is required by the user	Long = position of the record	O(n)

The searching for records with certain values can be, therefore, implemented via Linked list. We create lists with all necessary keys, and then use them to find corresponding positions. Consequently, we read all the information we need from the records using direct access. This procedure significantly lowers the time required for this action.

Such linked list implementation has also other advantages which have not been mentioned yet. For instance, if we want to use a similar list, but we need to store additional objects for individual record as well, all we need to do is to extend the Item object and add appropriate variable definitions and getters and setters.

This whole organization of **linked list** is illustrated on the following diagram.



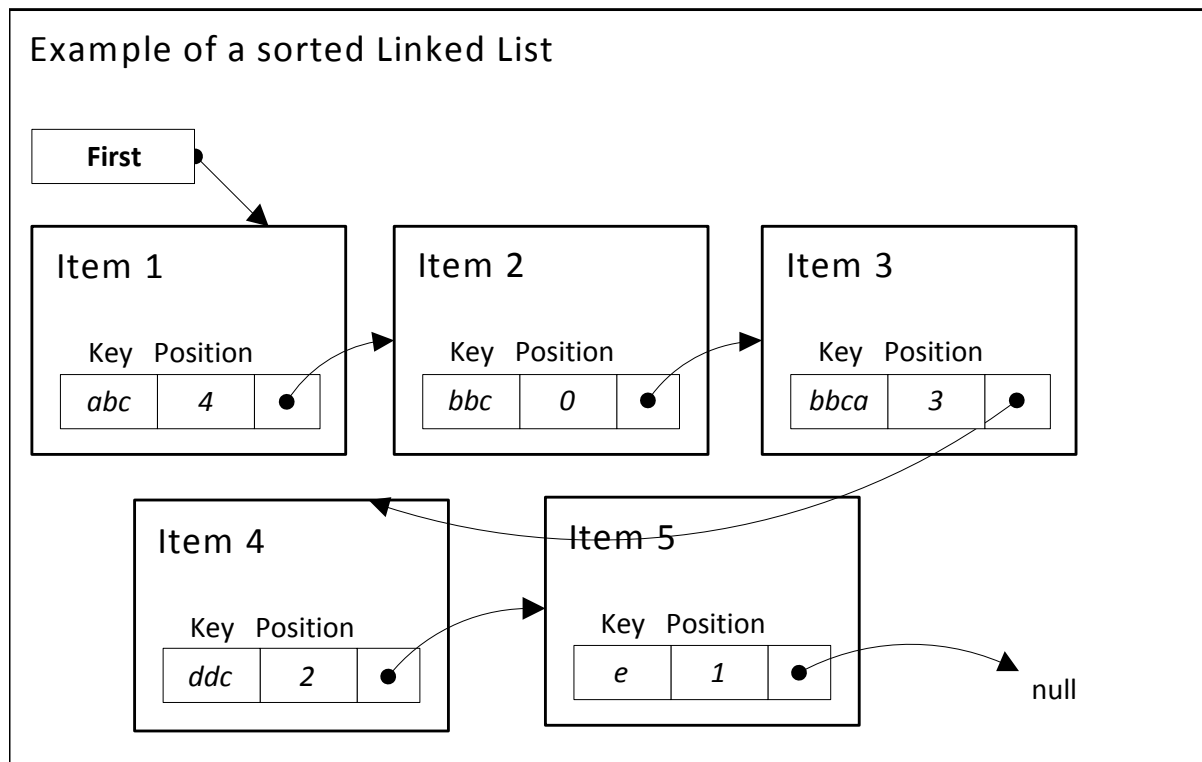


Figure 13: Illustration of Linked List

The use of the list in our application is pretty straightforward. For instance, the user wants the program to search in records according to 3 different criteria (registration number, name and surname), therefore, three lists with appropriate keys will be implemented. Also, all tables / graphs / other components representing data stored in the program which require sorting records by particular fields will have to use lists.

### Vector

The use of abstract data structure can be directly connected to their storing in files, but does not necessarily have to be. Having an enhanced dynamic data structure is very handy when it comes to many algorithms.

The linked list does provide a dynamic allocation of memory, but it is meant to be implemented in a way to easily suite its main purpose – indexing records in DirectAccessFiles.

It would be very useful to have additional dynamic complex abstract data structure which would support some typical methods.

Instead of having more data structures based on the same principle but having different operation methods implemented, I have decided to implement the most possible generic data structure – Vector.

Vector works on the same principle of linking VectorItems with each other as the linked list does. However, VectorItems are double linked – each of them points to the next VectorItem in the chain, but also to the previous VectorItem. The other difference is that its content does not necessarily

have to be String – using generics it is possible to implement a general object type so that we could use Vector for working with any variable types. Moreover, the range of methods provided by Vector for working with VectorItems is much wider than the range provided by other typical dynamic data structures such as Stack, Queue or Double linked list.

I will now briefly describe the **variables of the VectorItem<O>**:

Name of the variable	Type	Meaning
<b>Object</b>	O, e. i. type given using generics in variable definition	The content of the VectorItem variable which stores the particular object stored in VectorItem
<b>Next</b>	Pointer to VectorItem	Points to the next instance of VectorItem class – creates chain of VectorItems
<b>Previous</b>	Pointer to VectorItem	Points to the previous instance of VectorItem class – creates chain of VectorItems

The object variables of a Vector<O> are:

Name of the variable	Type of the variable	Meaning of the variable
<b>First</b>	VectorItem<O>	Pointer to the first/front item in the chain of VectorItem objects
<b>Last</b>	VectorItem<o>	Pointer to the last item in the chain of VectorItem objects
<b>Count</b>	Integer	Stores the number of items which are present in the Vector. Since this value is stored, the count does not have to be computed recursively every time it needs to be known, therefore, finding out the number of items in a Vector is much less time consuming – O(1)

Using these variables and the nature of VectorItem objects, the **Vector<O>** should offer the following basic functionality:

Name of the method	Meaning	Return type	Time complexity
<b>&lt;constructor&gt; (n, object)</b>	Initializes the object variables to default values and inserts the initial data – N times it inserts the given object	N/A	O(n)
<b>Size()</b>	Returns the number of elements in the Vector – return the value of the object variable count	Integer	O(1)
<b>IsEmpty()</b>	Checks whether the size is 0 or not	Boolean	O(1)
<b>Clear()</b>	Deletes all data in the Vector by initializing Vector's object variables to default values	void	O(1)
<b>First()</b>	Returns the first/front object in the Vector	O – the generic object type of the Vector	O(1)
<b>Last()</b>	Returns the last/back object in the Vector	O – the generic object type of the Vector	O(1)
<b>pushBack(object)</b>	Puts the object given as the last element/to the back of the VectorItem chain by changing the Last pointer of the Vector and adjusting Pointers within the last and newly added VectorItems.	Void	O(1)
<b>pushFront(object)</b>	Puts the object given to the front of the VectorItem chain by changing the Front pointer of the Vector class and adjusting the pointers within the first and newly added VectorItems.	void	O(1)
<b>popBack()</b>	Returns the last object in the Vector and removes its VectorItem from the chain by setting the last pointer of the Vector to previous	O	O(1)

	VectorItem and also adjusting the Next pointer of this item.		
<b>popFront()</b>	Returns the front object in the Vector and removes it from the VectorItem chain by changing the First pointer of the Vector to the second VectorItem, and also adjusting the Previous pointer of this VectorItem.	O	O(1)
<b>Get(index)</b>	Returns the object in the Vector at the given position or null if the index is out of bounds. It goes through the chain of VectorItems till the index is not equal to index wanted.	O	O(n) – VectorItem has to be found by traversing VectorItems from the beginning or the end
<b>Set (object, index)</b>	Sets the object given to the Vector at position given. It goes through the chain of VectorItems till the index does not correspond to the index given and then changes the object value of the VectorItem.	Boolean – true if successful	O(n) – VectorItem has to be found by traversing VectorItems from the beginning or the end
<b>indexOf(object)</b>	Returns the index of the object given in parameters in the Vector. If the object is not present in the Vector, -1 should be returned. It compares the given object with objects in the Vector one after.	Integer – the index value	O(n) - VectorItem has to be found by traversing VectorItems from the beginning or the end
<b>insertAt(object, index)</b>	Inserts a new element in the chain of VectorItems at the position given in parameters. It creates new VectorItem object and customizes the pointers of the	Boolean – true if successful	O(n)

	VectorItem with neighboring indexes.		
<b>Before(object)</b>	Returns the object O which is at the position just before the object given	O	O(n)
<b>After(object)</b>	Returns the object O which is at the position just after the object given	O	O(n)
<b>insertAfter(object, object)</b>	Inserts the given object right after the other object given	Boolean – true if successful	O(n)
<b>insertBefore(object, object)</b>	Inserts the given object just before the other object given	Boolean – true if successful	O(n)

The **Vector data structure** is also illustrated at the following diagram:

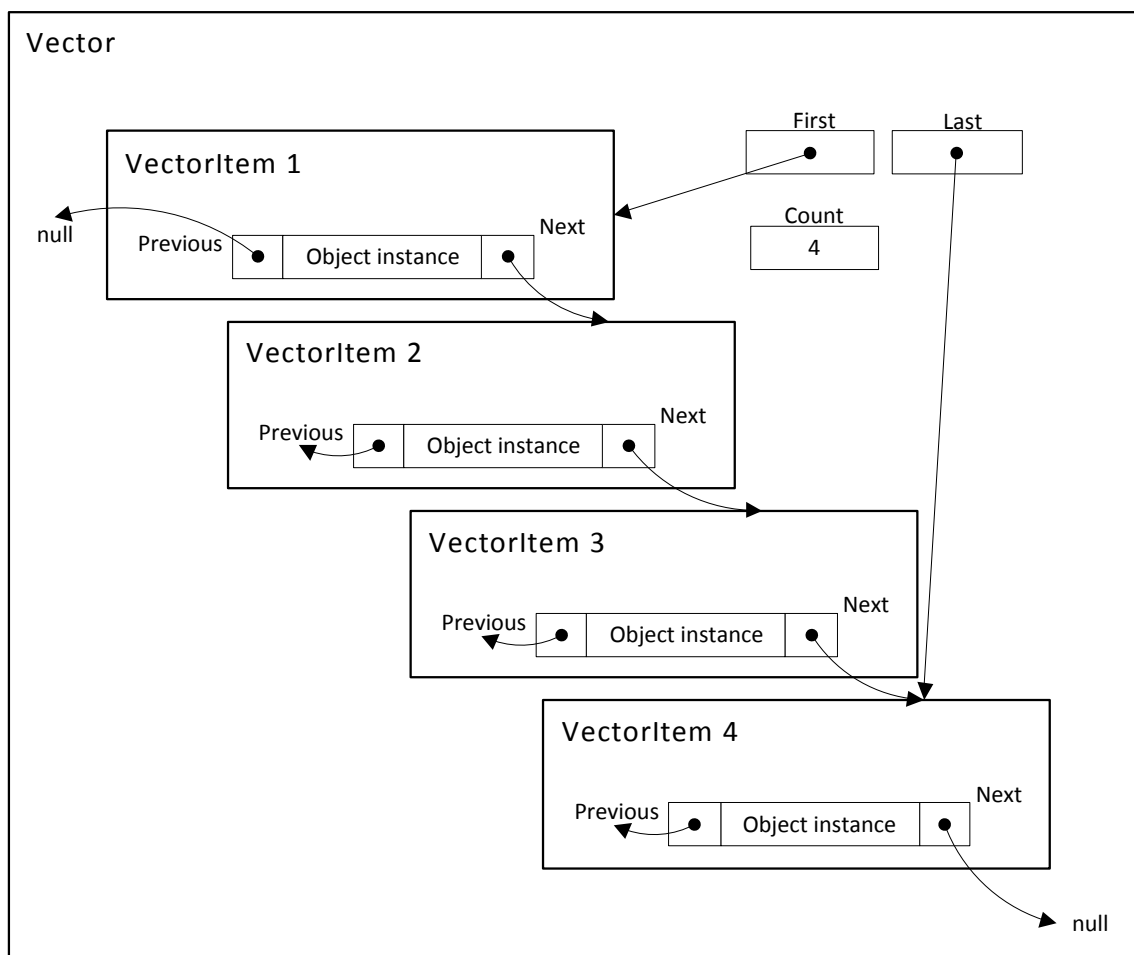


Figure 14: Illustration of Vector

This kind of a general data structure can be used for various purposes. Since elements can be freely inserted and popped from the beginning as well as from the end, it fully substitutes the Stack and Queue. Generally speaking, it is required everywhere where the static allocation of memory is insufficient. For instance, since the number of direct children is in time of initialization of object variables of a Treeltem unknown (more below), a dynamic allocation of memory is the only way to go.

## Hierarchical data structures

### Tree

This is not an implementation of a binary tree. It is just a basic tree consisting of linked items. Therefore, in comparison with a binary tree, each node can have as many children as required. Each Treeltem (Node) will store the following information:

Name of the variable	Type	Meaning
Parent	Pointer to Treeltem	Pointer to parent item in the Tree data structure.
Children	Array of pointers to Treeltem	Pointer to children items in the Tree data structure
Key	String	Representation of the record in particular instance of Tree
Position	Long	Position in file where the whole record is situated

Possibly, in our implementation we had better use one Tree with more key values, instead of more Trees with identical structure, because the tree structure can be very memory consuming.

We need such structure for representing the hierarchical structure of the network of distributors in the multilevel marketing system. Each distributor has only one sponsor (in this case their parent), but may have as many children (in this case children) distributors as possible. Therefore, if a Treeltem represents particular record of distributor a hierarchical structure of distributors can be easily represented using described Tree structure.

Implementing this structure alternatively without linking items with each other is possible, but definitely would be ineffective and impractical.

## B2: Algorithms

In this section I will briefly describe my proposal of design of algorithms which will have to be used in the program – because of the user's wishes stated in the analysis stage. I will describe neither getters and setters, nor stereotypical features such as traversal searching or sorting.

### class DirectAccessFile

DirectAccessFile is a class which encapsulates the Java's RandomAccessFile class, providing direct access to records in a file. In order to implement this class, I have to implement the methods with following functionality:

- Getting the length of the file [parameters: none; returns: long]
- Getting the number of records [parameters: none; returns: long]
- Seeking to a different record [parameters: Long position; returns: void]
- Writing to record with given position  
[parameters: String content, Long position; returns: boolean]
- Reading from record with given position [parameters: Long position; returns: String]
- Deleting record with given position [parameters: Long position; returns: none]
- Appending new record to file [parameters: String content; returns: boolean]

### Getting the length of the file

#### Pre-conditions:

- The file pointer to RandomAccessFile object has been initialized and it points to a working file

#### Description:

Finds out the length of the file from the RandomAccessFile and returns it. Since working with files is never risk-free, the method has to be encapsulated in the try-catch statement.

#### Pseudo-code:

Method getLength() return Long

Try

Return (file length determined from the RAF pointer)

Catch (IOException)

Display error message

Return value indicating that an error occurred, e. g. -1

#### Post-conditions:

- If the returned value was positive the length of the file has either been successfully returned
- If it was negative an error in determining the length occurred

Time-complexity: O(1)

### Getting the number of records

#### Pre-conditions:

- The file pointer to RandomAccessFile object has been initialized and it points to a working file
- The object has been successfully initialized, so the object variable indicating record length is set to valid length

Description:

It computes the number of records present in the RAF by dividing the length of the file by the length of a single record.

Pseudo-code:

```
Method numberOfRecords() return Long
    return (length of the file determined from the RAF pointer)/(length of a single record);
```

Post-conditions:

- Valid record length has been returned.

Time-complexity: O(1)

### Seeking to a different record

Pre-conditions:

- The file pointer to RandomAccessFile object has been initialized and it points to a working file
- The position in the file we want to seek to is within the range of existing records in the file.

Description:

If the position given in parameters is possible to seek to (e. i. it is positive number or 0), the method tries to move the pointer of the RAF using its seek method. I/O exception can occurred so it has to be handled properly.

Pseudo-code:

```
Method goToRecord (long n) return void
If n >= 0 then
    Try to
```

```
        Seek the RAF to the position n
```

```
Catch IOException
```

```
    Display warning
```

Post-conditions:

- If successful, the position of the file pointer in the instance of RandomAccessFile used by our DAF class has changed.

Time-complexity: O(1)

### Writing to record with given position

Pre-conditions:

- The file pointer to RandomAccessFile object has been initialized and it points to a working file
- The position in the file we want to write to is within the range of existing records in the file; or greater by one
- A valid String object is available in parameters.

Description:

The method firstly moves the file pointer to the wished position and then writes the content of the String given in parameters to that position using the method writeUTF provided by RAF pointer.



Pseudo-code:

```
Method writeToPosition (String, position) return void
    goToRecord(position)
    try to
        write the String to the RAF file using RAF pointer
    catch IO exception
        display error
```

Post-conditions:

If successful:

- The record at given position has been changed to given string.
- The position of the seeker in the RAF file has changed to the position given.

Time-complexity:  $O(1)$

### **Reading from record with given position**

Pre-conditions:

- The file pointer to RandomAccessFile object has been initialized and it points to a working file
- The position in the file we want to read from is within the range of existing records in the file

Description:

The method seeks to the position given in the file using the RAF pointer. It then reads the content of the particular record using the RAF pointer's readUTF method and returns it. A IOException which can occur has to be caught.

Pseudo-code:

```
Method readFromPosition(position) returns String
    If position >= 0 then
        Try to
            Return (Read from RAF file from position given)
        Catch IOException
            Display error
```

Post-conditions:

If successful:

- The record from wished position has been successfully returned.
- The position of the seeker in the RAF has changed to position given.

Time-complexity:  $O(1)$

### **Deleting record with given position**

Pre-conditions:

- The file pointer to RandomAccessFile object has been initialized and it points to a working file
- The position in the file we want to delete is within the range of existing records in the file

Description:

The file itself is not sorted so the order of files is not important. This method, therefore, reads the content of the last record in the file and stores it in the temporary variable. Then it overrides the

record with given position by the content of the last record. Lastly, the file is shortened by setting the length of the file using the RAF pointer.

Pseudo-code:

Method delete (position) returns void

```
Temp = readFromPosition (numberOfRecords()-1)
writeToPosition(temp,position)
```

```
set file size using the RAF pointer to (numberOfRecords()-1)*recordLength
```

Post-conditions:

- The record at given position does not exist anymore.
- The file has been shortened by recordLength
- The pointer of the RAF is set to the position of the deleted record

Time-complexity: O(1)

### Appending new record to file

Pre-conditions:

- The file pointer to RandomAccessFile object has been initialized and it points to a working file
- A valid String object is available in parameters.

Description:

This method writes the content of the String given in parameters to the last position in file+1 so that file is prolonged by the recordLength.

Pseudo-code:

Method append (String) returns void

```
writeToPosition (String,numberOfRecords())
```

Post-conditions:

- The file has been prolonged by a recordLength
  - The number of records has changed
- The pointer of the RAF is pointing to this newly created last record

Time-complexity: O(1)

### class Record

Record is a parent class of all particular record-style entities, in our case of Distributor and Month. It implements the common features of a record, such as:

- Merging record fields to single string and writing it to DirectAccessFile at given position  
[parameters: DirectAccessFile f, String[] fields, Long position; returns: void]
- Reading from given position from DirectAccessFile and parsing String into particular record fields  
[parameters: DirectAccessFile f, Long position; returns: void]

### **Merging record fields to single string and writing it to DirectAccessFile at given position**

#### Pre-conditions:

- The fully functioning file pointer to DirectAccessFile has been passed to the function in parameters
- The position in parameters is appropriate to the file length ( $0 \leq \text{position} \leq \text{numberOfRecords}()$ )
- Data array has been initialized and contains String values.

#### Description:

This method will use a general method *implode* from class static. Why? Because of the principle DRY (don't repeat yourself). Since the *implode* functionality is general and reusable, I will put it into the Static class and just use it out of the Record class.

Except for imploding the record fields, we have to think about security – this is the level at which we want to implement data encrypting.

All in all, the procedure will create the merged string by calling a method *implode* from static class, encrypt it using a method *encrypt* from static class and write it to given DAF at given position.

#### Pseudo-code:

Method *writeToPosition* (DAF, position) return void

```
String temp = Static.implode (object data array)
temp = Static.encrypt (temp)
daf.write to position (position)
```

#### Post-conditions:

- The file has changed because of the newly written record
- The pointer of the RAF is pointing to this newly created record

Time-complexity:  $O(\text{number of fields in data})$

### **Reading from given position from DirectAccessFile and parsing String into particular record fields**

#### Pre-conditions:

- The fully functioning file pointer to DirectAccessFile has been passed to the function in parameters
- The position in parameters is appropriate to the file length ( $0 \leq \text{position} \leq \text{numberOfRecords}() - 1$ )

#### Description:

It will read the String from given DAF pointer from the given position. Consequently, the string will be decrypted and the method will use the static *explode* function, which will be defined in the static class to convert the merged String into an array of Strings. This array will then override the object's array.

#### Pseudo-code:

Method *readFromPosition* (DAF, position) return void

```
String temp = daf.read from position(position)
```

```
temp = Static.decrypt (temp)
object data array= Static.explode (temp)
```

Post-conditions:

- The record fields of this record in the memory have changed to read values
- The pointer of the RAF is pointing to this recently read record

Time-complexity:  $O(\text{number of fields in data})$

## **class Distributor**

Distributor is an inherited class from general Record object. It works on the basis of Records but customizes concrete variables to distributor's record fields. Moreover, it provides some logic, when it comes to working with Distributor object.

- Determine the generation of the distributor in the distributors structure  
[parameters: none; returns: Integer]
- Get pointers to direct children distributors of the particular distributor  
[parameters: none; returns: Distributor[]]
- Get pointers to all children distributors of the particular distributor  
[parameters: none; returns: Distributor[]]

### **Determine the generation of the distributor in the distributors structure**

Pre-conditions:

- Fully functioning pointer to the corresponding Treeltem accessible as the Distributor's object variable
- The initialized structure of Treeltems

Description:

The method will access the Treeltem instance through the object variable. It will then repeatedly traverse the Treeltem structure to the parent of the actual Treeltem, gradually increasing counter of generation. This counter will be eventually returned.

Pseudo-code:

Method getGeneration () return Integer

```
    Generation = 0
```

```
    CurrentTreeltem = object's treeltem
```

```
    While CurrentTreeltem is defined
```

```
        Generation = generation +1
```

```
        CurrentTreeltem = CurrentTreeltem.parent
```

```
    Return generation
```

Post-conditions:

- The generation value is returned

Time-complexity:  $O(\text{generation})$

### **Get pointers to direct children distributors of the particular distributor**

#### Pre-conditions:

- Fully functioning pointer to the corresponding TreeItem accessible as the Distributor's object variable
- The initialized structure of TreeItems

#### Description:

This method will get the array of positions of records in file at which the direct children distributor of the particular distributor are situated. It will then read the records from these positions into an array and return it.

#### Pseudo-code:

```
Method getDirectChildrenDistributors () return Distributor[]  
    positions[] = treeItem.getDirectChildrenPositions()  
    distributors[] = load distributors using DistributorManager class from particular positions  
    Return distributors[]
```

#### Post-conditions:

- The array of distributors is returned

Time-complexity:  $O(\text{number of direct children distributors})$

### **Get pointers to all children distributors of the particular distributor**

#### Pre-conditions:

- Fully functioning pointer to the corresponding TreeItem accessible as the Distributor's object variable
- The initialized structure of TreeItems

#### Description:

This method will get the array of positions of records in file at which the all children distributor of the particular distributor are situated from the treeItem object representing the hierarchical structure of the distributors in the MLM system. It will then read the records from these positions into an array and return it.

#### Pseudo-code:

```
Method getAllChildrenDistributors () return Distributor[]  
    Positions[] = treeItem.getAllChildrenPositions()  
    Distributors[] = load distributors using DistributorManager class from particular positions
```

#### Post-conditions:

- The array of all distributors is returned

Time-complexity:  $O(\text{number of all children distributors})$

## **class Month**

Month is an inherited class from the Record object. It also works on the basis of Records but customizes concrete variables to month's record fields. Moreover, it provides some useful logic, when it comes to operating the Month object.

- Calculation of TotalPW  
[parameters: none; returns: Integer]
- Calculation of TotalGW  
[parameters: none; returns: Integer]
- Determining the level distributor achieved that Month  
[parameters: none; returns: Double]
- Determining the BonusLevel distributor achieved that Month  
[parameters: Integer numberOf21Children; returns: Double]
- Calculation of provision  
[parameters: none; returns: Double]

### **Calculation of TotalPw**

#### Pre-conditions:

- The object variables of the Month object have been properly initialized and filled with valid data

#### Description:

The method calculates the TotalPW as the sum of PW and GroupPW of particular Month record. This value is then returned.

#### Pseudo-code:

Method getTotalPw() return Integer

Return (pw + groupPw)

#### Post-conditions:

- The calculated value is returned

#### Time-complexity: O(1)

### **Calculation of TotalGw**

#### Pre-conditions:

- The object variables of the Month object have been properly initialized and filled with valid data

#### Description:

The method calculates the TotalGW as the sum of GW and GroupGW of particular Month record. This value is then returned.

#### Pseudo-code:

Method getTotalGw() return Integer

Return (gw + groupGw)

Post-conditions:

- The calculated TotalGW value is returned

Time-complexity: O(1)

**Determining the level distributor achieved that Month**

Pre-conditions:

- The object variables of the Month object have been properly initialized and filled with valid data

Description:

The method calculates the Level according to criteria given by the user in the analysis stage.

Pseudo-code:

Method getLevel() return Integer

```
If totalPw < 2 then return 0
If totalPw < 500 then return 3
If totalPw < 1000 then return 6
If totalPw < 2000 then return 9
If totalPw < 4000 then return 12
If totalPw < 8000 then return 15
If totalPw < 12000 then return 18
return 21
```

Post-conditions:

- The calculated value of Level is returned

Time-complexity: O(1)

**Determining the bonus level distributor achieved that Month**

Pre-conditions: none

Description:

The method calculates the BonusLevel from the number of direct children distributors with level of 21, according to criteria given by the user in the analysis stage.

Pseudo-code:

Method getBonusLevel(numberOf21children) return Integer

```
If numberOf21children < 1 then return 0
If numberOf21children == 1 then return 7
If numberOf21children < 4 then return 7.5
If numberOf21children < 6 then return 8
If numberOf21children < 8 then return 8.5
If numberOf21children < 10 then return 9
If numberOf21children < 12 then return 9.5
return 10
```

Post-conditions:

- The calculated value of BonusLevel is returned

Time-complexity: O(1)

**Calculation of provision**

Pre-conditions:

- The object variables of the Month object have been properly initialized and filled with valid data
- The month's distributor pointer has fully functioning pointer to the corresponding TreeItem accessible as the Distributor's object variable
- The initialized structure of TreeItems

Description:

Month object firstly receives all direct distributors of the distributor it belongs to. Consequently, it calculate the level of each child distributor. If it is 21, the number of distributors with level 21 has been increased and the totalGw for all of these distributors is stored in memory.

Alternatively, if the children distributor is not on the 21 level, a bonus difference between the distributor and child distributor is calculated and added to provision. The final value of provision is returned.

Pseudo-code:

Method getProvision () return Integer

```
childrenDistributors[] = distributor pointer. treeItem pointer.get DirectChildrenDistributors()
for each child in childrenDistributors
    childMonth = child. Pointer to month manager. Get actual month;
    if childMonth. getLevel() == 21 then
        numberOf21 =numberOf21 +1;
        totalGw21 = totalGw21 + childMonth.getGw()
    else
        provision = provision + childMonth.getGw()*(getLevel ()– childMonth.getLevel())
provision = provision + bonusLevel()*totalGw21;
return provision
```

Post-conditions:

- Calculated provision has been returned

Time-complexity: O(number of children distributors)



## **class TreeItem**

Class TreeItem represents the hierarchical structure of the distributors network in MLM system. It works on the principle of linking various TreeItems through pointers – this has been discussed in the previous B1 section. Apart from general methods, it should provide the following non-trivial methods:

- Get number of all children distributors  
[parameters: none; returns: Integer]
- Get position of current item and all its children

### **Get number of all children distributors**

#### Pre-conditions:

- The initialized structure of TreeItems

#### Description:

The method goes through the array of children and to the counting variable it for each child adds +1 + the recursively called method on the child object.

#### Pseudo-code:

```
Method getNumberOfAllDistributors () return Integer
    for each child in children
        count = count +1 + child.getNumberOfAllDistributors()
    return count
```

#### Post-conditions:

- Calculated number of all distributors has been returned

Time-complexity:  $O(\text{number of children distributors})$

### **Get position of current item and all its children**

#### Pre-conditions:

- The initialized structure of TreeItems

#### Description:

This method makes use of the dynamic data structure Vector. It adds the position in file of the current distributor to the Vector and recursively runs this method on children objects with the same Vector passed in parameter

#### Pseudo-code:

```
Method getPositionOfCurrentItemAndAllItsChildren (Vector) return void
    Vector.pushBack (current position )
    for each child in children
        child.getPositionOfCurrentItemAndAllItsChildren(Vector)
```

Post-conditions:

- Positions of all children have been added to the Vector

Time-complexity: O(number of children distributors)

## **class Static**

Class Static contains constants for the whole application and useful static methods with general purpose.

- Customizing String length to unified length  
[parameters: String orig, Integer length; returns: String]
- Cut String if longer than given length  
[parameters: String orig, Integer length; returns: String]
- Number of occurrences of a given char in a given String  
[parameters: String orig, char c; returns: Integer]
- Check if all of the String characters are numeric  
[parameters: String orig; returns: boolean]
- Remove given char from given String  
[parameters: String orig, char c; returns: String]
- Merge given array of Strings to single String with given separator (implode)  
[parameters: String[] fields, char separator; returns: String]
- Parse merged Strings with a given separator to array of String (explode)  
[parameters: String merged, char separator; returns: String[]]
- Conjunction of more given Integer arrays  
[parameters: int[][] in; returns: int[]]
- Quick sort  
[parameters: int[] in; returns: int[]]
- Char array to String  
[parameters: char[] chards; returns: String]
- E-mail validation  
[parameters: String orig; returns: boolean]

## **Customizing String length to unified length**

Pre-conditions:

- String in parameters is defined
- Length in parameters is defined

Description:

If the length of the given string is lower than the given length, the string is shortened and returned. If the length is higher than the length of string, the string is prolonged with spaces and returned.

Pseudo-code:

Static Method customizeStringLength (String, length) return String

```
If string.length < length then
    For (i=string.length ... i<length) string += " " //add empty space
    Return string
If string.length > length then
    Return (part of the string from 0 to length chars)
```

Post-conditions:

- The customized String has been returned

Time-complexity:  $O(\text{length})$

### **Cut String if longer than given length**

Pre-conditions:

- The String given is defined
- The length given is defined

Description:

If the string is longer than the given length, the string is cut and returned. Alternatively, original String gets returned.

Pseudo-code:

```
Static Method cutStringIfLongerThan (String, length) return String
    If String.length > length then
        Return (part of the String from 0 to length chars)
    Return String;
```

Post-conditions:

- A string with adjusted length has been produced

Time-complexity:  $O(\text{length})$

### **Number of occurrences of a given char in a given String**

Pre-conditions:

- The String given is defined
- The char given is defined

Description:

It goes through the String and compares every char of the String with the char given. If they are equal, the counter is increased.

Pseudo-code:

```
Static Method numberOfOccurrences (String, char) return Integer
    For each char of the given string
        If actual char == given char
            NumberOfOccurrences = NumberOfOccurrences + 1;
```

Return NumberOfOccurrences;

Post-conditions:

- Number of occurrences has been returned.

Time-complexity: O(String length)

**Check if all of the String characters are numeric**

Pre-conditions:

- The String given is defined

Description:

Goes through the given string char by char and checks whether the char is numeric or not. If it is not numeric (e. i. 0,1,2,3,4,5,6,7,8 or 9) the method returns false. If the end of the String is successfully reached, true is returned.

Pseudo-code:

Static Method isNumeric (String) return boolean

For each char in String

    If !(actualChar <= "9" && actualChar >= 0) return false

Return true

Post-conditions:

- The resulting value has been returned

Time-complexity: O(length of the String)

**Remove given char from given String**

Pre-conditions:

- The String given is defined
- The char given is defined

Description:

It goes through the given String char by char. If the actual char is not equal to the char given, it gets copied to the new version of the String. The new version of the String gets returned at the end of the method.

Pseudo-code:

Static Method removeCharFromString (String, char) return String

newString = ""

for each char in String

    if actualChar != char then

        newString += actualChar

return newString

Post-conditions:

- The new String has been produced and returned

Time-complexity: O(String length)

### **Parse merged Strings with a given separator to array of String (explode)**

#### Pre-conditions:

- The given String is defined
- The given separator==char is defined

#### Description:

It goes through the given String char by char and memorizes chars to temporary String. If the separator char is met, the temporary string is inserted to the array and emptied.

#### Pseudo-code:

Static Method explode (String, char) return String[]

String[numberOfOccurrences(char)] //define the new array to the determined length

Index = 0

Temp = ""

For each char in the String

    If actualChar == char then

        String[index] = temp

        Temp = "";

    Else temp += actualChar

Return String[]

#### Post-conditions:

- The new String array has been produced and returned

Time-complexity: O(length of the String)

### **Merge given array of Strings to single String with given separator (implode)**

#### Pre-conditions:

- The given array of String is defined
- The given separator==char is defined

#### Description:

It goes through the given String array and adds each of its items into the temporary string, separating individual items by char given. The temporary string is then returned.

#### Pseudo-code:

Static Method implode (String[], char) return String

For each String in the array

    Temp += currentString+separator;

Return Temp

Post-conditions:

- The new merged String has been produced and returned

Time-complexity:  $O(\text{length of the array})$

### **Conjunction of more given Integer arrays**

Pre-conditions:

- The given array of arrays of Integers is defined

Description:

Each array in array of arrays gets sorted. Then the first array is traversed and first values of all Integer arrays are compared. If the value is in all arrays, it is written in the return array; otherwise the algorithm continues and checks the following Integer in the first array the same way.

Pseudo-code:

```
Static Method conjunction (Integer[][]) return Integer[]
    For each array in the Integer[][]
        Sort array
    For each integer in the first Integer[0] array
        For each arrays of integers
            If the value is equal to value in the first array
                Temp array[] = value
    Return temp array
```

Post-conditions:

- The new merged Integer array has been produced and returned

Time-complexity:  $O(\text{number of arrays} * \text{average array length})$

### **E-mail validation**

Pre-conditions:

- The String given is defined

Description:

Checks the given string for the presence of "@" char and "." char with greater position than "@".

Pseudo-code:

```
Static Method validateEmail (String) return boolean
    If String.indexOf("@") != -1 and String.indexOf("@") < String.lastIndexOf(".") then
        Return true
```

Return false

Post-conditions:

- Boolean method is returned.

Time-complexity:  $O(1)$

## **class List**

List, as a simple dynamic data structure, needs following non-trivial methods for operation:

- Add item to the beginning of the List  
[parameters: String orig, Long position; returns: void]
- Add item to the sorted List  
[parameters: String orig, Long position; returns: void]
- Find position by given key in list  
[parameters: String key; returns: long]
- Find positions of all items with key of which given key is substring in a sorted list  
[parameters: String key; returns: long[]]

### **Add item to the beginning of the List**

Pre-conditions:

- The String given is as key defined
- The position given is defined

Description:

It creates the new Item with the information given and inserts it into the list to the first position.

Pseudo-code:

Static Method addToBeginning (String, position) return void

Item = new Item (String, position);

Item.setNext (first); //set the current first item as Next

First = item //set the new item as the first item in list

Post-conditions:

- The length of the list has been increased by one.
- The First pointer of the List points to newly created Item.

Time-complexity:  $O(1)$

### **Add item to the sorted List**

Pre-conditions:

- The String given is as key defined
- The position given is defined

Description:

It creates the new Item with the information given and inserts it into the list to the right position so that the list remains sorted according to the key.

Pseudo-code:

Static Method addSorted (String, position) return void

    newItem = new Item (String, position);

    newItem.setNext (first); //set the current first item as Next

    If the list is empty

        First = newItem

    Else

        Act = first

        While (act != null)

            If act.key is less than newItem.key then

                Act = act.getNext();

        //the act is now pointing to the point where new item should be inserted

        newItem.setNext(act.getNext())

        act.setNext(newItem)

Post-conditions:

- The length of the list has been increased by one.
- The new Item has been add to the chain of linked Items

Time-complexity: O(number of Items in List)

**Find position by given key in list**

Pre-conditions:

- The String given is as key defined

Description:

It goes through the chain of Items and looks for the given key. When the item with corresponding key is found, its position is returned.

Pseudo-code:

Static Method find Position(String) return Integer

    Act = first

    While (act != null)

        If act.key equals to found key then

            Return act.position

Post-conditions:

- The found position has been returned; the position is in the range from 0 to number of records in the file-1

Time-complexity: O(number of Items in List)



### **Find positions of all items with key of which given key is substring in a list**

#### Pre-conditions:

- The String given is as key defined

#### Description:

It goes through the chain of Items and looks for the given key. When the item with corresponding key is found, its position is stored in the temporary vector. When the end of the chain of linked items is reached, the content of temporary vector is returned.

#### Pseudo-code:

Static Method findMorePositions(String) return Vector<Long>

VectorPositions = temporary vector with long values

Act = first

While (act != null)

    If act.key equals to found key then

        Add act.position to VectorPositions

Return VectorPositions

#### Post-conditions:

- The found positions has been returned; each position is in the range from 0 to number of records in the file-1

Time-complexity: O(number of Items in List)

### **class Vector<O>**

Vector is a complex composite data structure with dynamic allocation of memory. In order to full its purpose, it has to implement wide range of methods for data manipulation:

- Push front [parameters: O object; returns: void]
- Push back [parameters: O object; returns: void]
- Pop front [parameters: none; returns: O]
- Pop back [parameters: none; returns: O]
- Get object in Vector at given position [parameters: Integer position; returns: O]
- Insert given object at given position in the Vector  
[parameters: O object, Integer position; returns: void]
- Find the index of object given in the vector  
[parameters: O object; returns: Integer]
- Insert given object after the occurrence of the other object given  
[parameters: O after, O object; returns: void]
- Insert given object before the occurrence of the other object given  
[parameters: O before, O object; returns: void]

I will now illustrate the way some of these algorithms work:

### **Push front**

#### Pre-conditions:

- The vector has been initialized to store objects of particular type
- The object given is defined and fits the type for which the Vector has been initialized

#### Description:

It adds the given object to the first position in the chain of linked VectorItems.

#### Pseudo-code:

Method pushFront(Type object) return void

Vi = new VectorItem (object)

vi.setNext(first)

first = vi

if isEmpty() then last = vi

count = count + 1

#### Post-conditions:

- Number of objects in the vector has increased
- New vectorItem has been added to chain of linked Vectoritems
- First (and possibly last) pointer of the vector is pointing to newly added vectorItem

Time-complexity: O(1)

### **Pop front**

#### Pre-conditions:

- The vector has been initialized to store objects of particular type
- There is at least one element present in the Vector

#### Description:

The First element in the file gets returned and is removed from the linked vectorItem. Also the vector pointer First gets changed.

#### Pseudo-code:

Method popFront() return Type

If the vector is not empty

Item = first

First = first.getNext()

Count = count -1

Return item

#### Post-conditions:

- Returned item is of the type for which the Vector has been initialized
- The number of elements in the vector has decreased by one
- First (and possibly last) pointer of the vector is pointing to the following item in Vector

Time-complexity: O(1)

### **Insert given object at given position in the Vector**

#### Pre-conditions:

- The vector has been initialized to store objects of particular type
- The object to insert has been initialized
- The position at which the new object is meant to be inserted is within the range 0...count

Description:

The method finds the VectorItem after which the element should be inserted and then creates the new vectorItem and customizes appropriate pointers so that the chain of vectorItems remains consistent.

Pseudo-code:

Method insertAt(Type Object, index) return void

```
If index == 0 then
    pushFront(Object)
else If index == count-1 then
    pushBack(Object)
else
    count = 0
    act = first
    while (count != index)
        act = act.getNext()
        count = count + 1
    //pointers to items in chain whose pointers have to be updated
    before = act;
    after = act.getNext()
    //create new Item
    It = new Item (Object)
    //set correct pointers inside the chain
    It.setNext(after)
    It.setPrevious(before)
    Before.setNext(It)
    After.setPrevious (It)
```

Post-conditions:

- The number of elements in the vector has increased by one
- Pointers within the chain of vectorItems have been updated

Time-complexity: O(number of elements in vector)

**Find the index of object given in the vector**

Pre-conditions:

- The vector has been initialized to store objects of particular type
- There is at least one element present in the Vector

Description:

Elements in the vector get compared with the object given. If an identical object is found, the index of such VectorItem is returned.

Pseudo-code:

Method indexOf(Object) return Integer

```
Act = first
I=0 //index
While act != null //there is another vectorItem linked
    If act.object equals Object
        Return I
    I = I+1
    Act = act.getNext()
```

Post-conditions:

- Returned integer value is in the range 0 ... number of elements in vector-1

Time-complexity: O(number of elements in vector)

### **class Dialog**

Except for trivial methods which offer basic window functionality the class encapsulates the following interesting method:

- Display profile of the particular distributor  
[parameters: O before, O object; returns: void]

#### **Display profile of the particular distributor**

Pre-conditions:

- The customized linked list listRegNum accessible from DistributorManager object is initialized
- Given string is a valid registration number of existing distributor in the system

Description:

It finds the pointer to profileWindow of particular distributor. If its profile window has not been displayed yet, new window is created. Otherwise an existing window is activated again.

Pseudo-code:

```
Static Method displayProfile(regNum) return boolean
    Item = listRegNum.find (regNum)
    If Item does not exist
        Return false
    If Item.window has been initialized before //pointer to JFrame
        Set Window visible and send it to front
    Else
        Item.window = create new JFrame
```

Post-conditions:

- Item.window points to display JFrame displaying distributor's profile window

Time-complexity: O(number of distributors)

## **class DistributorManager**

DistributorManager is a class which operates the logic around managing records of Distributor objects. Since the management of distributors is only one within the whole application, all its methods are meant to be static so that they can be used without initialization of the class. Wide range of methods is required for a successful Distributor management:

- Populate lists for quick searching and TreeItem with current data from files  
[parameters: void; returns: void]
- Add new distributor [parameters: Distributor d; returns: void]
- Get all distributors [parameters: void; returns: Distributor[]]
- Find distributors by combination of their name, surname and registration number  
[parameters: String name, String surname, String regNum; returns: Distributor[]]
- Get distributor object by its registration number  
[parameters: String regNum; returns: Distributor]
- Get distributor object by its sponsor  
[parameters: String sponsor; returns: Distributor]
- Edit distributor information in files and lists  
[parameters: Distributor d; returns: void]
- Delete distributor from file and lists  
[parameters: Distributor d; returns: void]
- Get all children distributors of the particular distributor  
[parameters: Distributor d; returns: Distributor[]]

### **Add new distributor**

#### Pre-conditions:

- Pointer to initialized DirectAccessFile for working with Distributor records

#### Description:

It adds saves the Distributor object given in parameters to the file and adds the information to Linked lists.

#### Pseudo-code:

Static Method addDistributor(distributor) return void

distributor.append(f) //f is a pointer to DAF

addToLists (distributor) //update the Linked lists in the memory

#### Post-conditions:

- The number of items in each of the linked lists has been increased by one
- The number of distributors has been increased by one

Time-complexity:  $O(1)$  /  $O(n)$  for sorted lists

### **Get all distributors**

#### Pre-conditions:

- Pointer to initialized DirectAccessFile for working with Distributor records

Description:

It lists all distributors in the system by reading all records in the corresponding DAF and putting the objects into an array.

Pseudo-code:

```
Static Method getAll() return Distributor[]
    Distributors[(number of records in the file)]
    For i=0 to i=number of records-1
        Distributors[i] = new Distributor()
        Distributors[i].readFromFile(f, i) //from the DAF pointer and i position
    Return Distributors
```

Post-conditions:

- Array of Distributors has the size equal to number of records in the file with Distributor records

Time-complexity:  $O(\text{number of distributors})$

**Find distributors by combination of their name, surname and registration number**

Pre-conditions:

- Pointer to initialized DirectAccessFile for working with Distributor records
- Lists (object variables of the DistributorManager) are initialized and filed with data about existing distributors

Description:

It provides searching for distributor according to tree criteria – name, surname and registration number. Firstly it finds positions of occurrences within all tree lists, of which conjunction is consequently produced. Records from these positions (positions which have been found in all lists) are then fetched.

Pseudo-code:

```
Static Method find(name,surname,regNum) return Distributor[]
    positonsName = listName.findMorePositions (name)
    positionsSurname = listSurname.findMorePositions (surname)
    positionsRegNum = listRegNum.findMorePositions (regNum)

    positionsToRead[] = Static.conjunction({positionsName, positionsSurname, positionsRegNum})
    Distributors[positionsToRead.length]
    For i=0 to i=positionsToRead.length-1
        Distributors[i] = new Distributor()
        Distributors[i].readFromFile(f, positionsToRead[i]) //from the DAF pointer and i
position
    Return Distributors
```

Post-conditions:

- The length of the array returned is in the range from 0 to number of distributors in the system.

Time-complexity:  $O(\text{number of distributors})$

### **class MonthManager**

Similarly to DistributorManager, MonthManager manages the Month records in the application corresponding to particular user. In comparison to DistributorManager, however, MonthManager is not static. Each instance of MonthManager belongs to particular distributor for which it was initialized.

It consequently operates only Month records corresponding to particular distributor.

- Get all months for particular user  
[parameters: void; returns: Month[]]
- Get array of Months for last n months  
[parameters: Integer n; returns: Month[]]
- Get particular Month object by its year and month combination  
[parameters: String key; returns: Month]

#### **Get all months for particular user**

##### Pre-conditions:

- The MonthManager object variable pointer to Distributor to which this MonthManager belongs to is initialized and working.
- Pointer to initialized DirectAccessFile for working with Month records for particular distributor is working

##### Description:

It lists all Months in the system by reading all records in the corresponding DAF and putting the objects into an array.

##### Pseudo-code:

```
Static Method getAll() return Month[]
    Months[(number of records in the file)]
    For i=0 to i=number of records-1
        Months[i] = new Month()
        Months[i].readFromFile(f, i) //from the DAF pointer and i position
    Return Months
```

##### Post-conditions:

- Array of months has the size equal to number of records in the corresponding file with Month records

Time-complexity:  $O(\text{number of months})$

#### **Get particular Month object by its year and month combination**

##### Pre-conditions:

- The MonthManager object variable pointer to Distributor to which this MonthManager belongs to is initialized and working.

- Pointer to initialized DirectAccessFile for working with Month records for particular distributor is working

Description:

The method tries to find month by given String (in format yyyyMM) in the list. If it is found, the whole record is read out of the corresponding DAF and returned.

Pseudo-code:

Static Method getMonth(String) return Month

Item = list.find(String) //find in list by key yyyyMM

If item exists

M = new Month

Return m.readFromFile(f,Item.pos)

Post-conditions:

- Month object is returned

Time-complexity: O(number of months)



### **B3: Modular Organization**

For every project it is important that it remains maintainable after the initial release is produced. This is one of the reasons why a good, clear and intuitive organization of the project is required.

Methods or functions which are dealing with particular issues are organized to classes, where they can cooperate together. Each class is supposed to be a meaningful whole, where all the methods included serve a common purpose.

The stage A – analysis, and previous parts of stage B – design have outlined how certain parts of the application will be organized. The complete organization of all the modules in the application can be seen on the following diagram – Figure 11.

There are 6 types of classes in the program.

- **Static classes**, classes which take care of typical operations; their methods are usually being reused throughout the whole application
- **File classes** are classes which make the file operation user-friendly; they take care of the working with files at the lowest level in the application
- **Data structures**, classes which are designed for handling data the application is working with in memory
- **Logic classes** e. i. classes which use the static, file classes and data structures for providing methods which are directly connected with the application's functionality; in our case managing distributor and month entities
- **Custom GUI controls** are classes which typically extend the official swing and awt classes; their aim is to add advanced features to the applications, which could not be provided otherwise
- **Windows & dialogs** are Graphic User Interface (GUI) classes which use GUI controls and logic classes for providing a user-friendly interface for the user. They typically take the information provided by the logic classes and display it; or take the input from the user, validate it and pass to logic classes

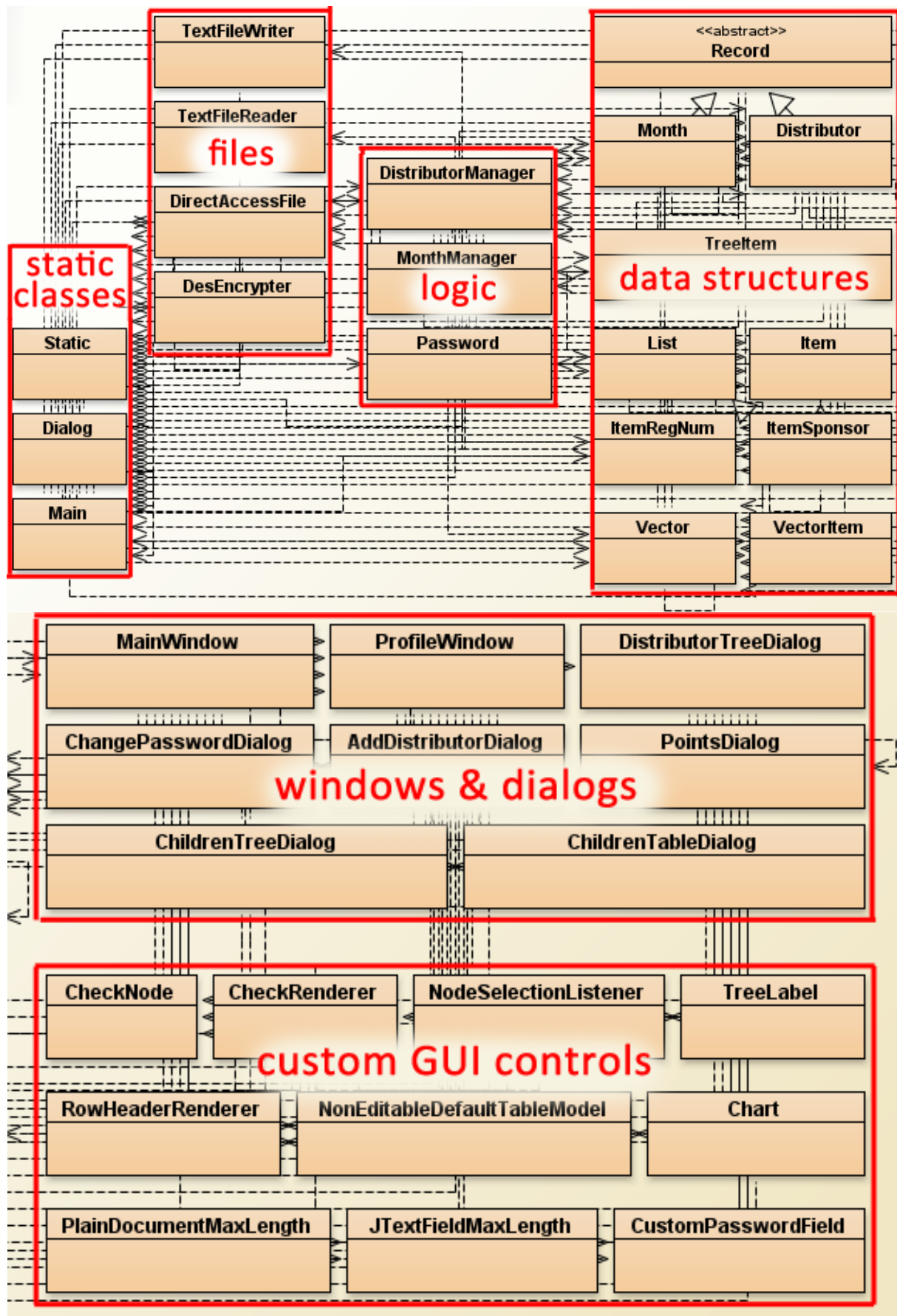


Figure 15: Modular organization of the application

I will now go through all the classes in the application and describe its functions and dependences.

- **Static classes**

- **Static** – provides Static methods for basic actions, such as working with String (counting chars, checking chars, customizing length,...), Arrays (imploding/exploding Strings, conjunction of Integer Arrays), and contains constants for the whole application (file paths, encryption keys,...)
  - **Depends on:** none
  - **It is used in:** basically the whole project, in logic classes as well as in data structures and GUI
- **Dialog** – provides Static methods for working with GUI – executing basic dialogs, etc.
  - **Depends on:** none
  - **It is used in:** throughout the whole application; for displaying error messages in file classes, informative dialogs in window classes, etc.
- **Main** – bootstrapper of the whole application. It checks the initial conditions for successful operation of the program, asks the user for the program and then launches the main window of the application
  - **Depends on:** none
  - **It is used in:** none

- **File classes**

- **TextFileWriter** – encapsulates the Java's classes and provides easy-to-use interface for writing to sequential text files.
  - **Depends on:** none
  - **It is used in:** Password class
- **TextFileReader** – encapsulates the Java's classes and provides easy-to-use interface for reading to sequential text files.
  - **Depends on:** none
  - **It is used in:** Password class
- **DirectAccessFile** – provides interface work writing and reading records to RandomAccessFile.
  - **Depends on:** none
  - **It is used in:** DistributorManager, MonthManager
- **DesEncrypter** – encrypting and decrypting strings by a given passphrase. The encrypted text uses only visible characters because it is encoded in base64 encoding.
  - **Depends on:** none
  - **It is used in:** Static

- **Logic classes**

- **DistributorManager** – manages creating, editing, deleting and finding Distributor records
  - **Depends on:** DirectAccessFile, Vector, List, TreeItem, Distributor
  - **It is used in:** MainWindow, AddDistributorDialog, DistributorTreeDialog, ChildrenTreeDialog, ChildrenTableDialog, Distributor
- **MonthManager** – manages creating, editing and finding Month records for particular Distributor
  - **Depends on:** DirectAccessFile, Distributor, List
  - **It is used in:** Distributor, Month

- **Password** – takes care of password management; checking and setting password access to the application
  - **Depends on:** TextFileWriter, TextFileReader, Dialog
  - **It is used in:** none
- **Data structures**
  - **Record** – the abstract data structure for working with entities in DirectAccessFile
    - **Depends on:** DirectAccessFile, Static
    - **It is used in:** Distributor, Month
  - **Distributor** – a customized Record for managing Distributor record fields. Implements the connection to MonthManager for easy access to Months of particular Distributor
    - **Depends on:** Record, DistributorManager, Static, TreeItem
    - **It is used in:** DistributorManager, profileWindow and basically throughout the whole GUI
  - **Month** – a customized Record for managing Month entity record fields. Implements the connection to the Distributor object to which particular Month belongs to and to the MonthManager for easy editing
    - **Depends on:** Record, Distributor, MonthManager
    - **It is used in:** MonthManager, whole GUI
  - **Vector** – a dynamic complex data structure with use of generics (supports any given variable type) and wide range of actions which can be performed on objects (replaces the stack, queue, double linked list,...)
    - **Depends on:** VectorItem
    - **It is used in:** the whole application, because of its general nature
  - **VectorItem** – class representing a single Item in Vector. Provides the double-linked nature of the Vector.
    - **Depends on:** none
    - **It is used in:** Vector
  - **List** – simple implementation of a dynamic data structure used for indexing records stored in DirectAccessFiles in memory
    - **Depends on:** Item and its children
    - **It is used in:** DistributorManager, MonthManager
  - **Item** – provides the linked nature of the List, linking Items with each other
    - **Depends on:** none
    - **It is used in:** List
  - **ItemRegNum** – customized version of Item, adds special fields and, therefore, stores additional information. The Item's behavior is, however, preserved.
    - **Depends on:** Item
    - **It is used in:** instance of List in DistributorManager
  - **ItemSponsor** – customized version of Item, adds special fields and, therefore, stores additional information. The Item's behavior is, however, preserved.
    - **Depends on:** Item
    - **It is used in:** instance of List in DistributorManager
  - **TreeItem** – represents the linked hierarchical structure of the linked distributors in the MLM network
    - **Depends on:** none

- **It is used in:** Distributor
- **Custom GUI controls**
  - **CheckNode, CheckRenderer, NodeSelectionListener, TreeLabel** are customized components enabling creation of JTree which implements Checkboxes
    - **Depends on:** they have dependences within themselves
    - **It is used in:** DistributorTreeDialog
  - **RowHeaderRenderer** – implements the behavior of a Header to particular row in a JTable
    - **Depends on:** none
    - **It is used in:** profileWindow, PointsDialog
  - **NonEditableDefaultTableModel** – implements the non-editable behavior to cells of JTable
    - **Depends on:** none
    - **It is used in:** mainWindow, profileWindow, pointsDialog
  - **Chart** – takes care of displaying information about Distributor's monthly performance in a graphical way
    - **Depends on:** external library JFreeChart used
    - **It is used in:** profileWindow
  - **PlainDocumentMaxLength** – implements the maximum length parameter in entry field
    - **Depends on:** none
    - **It is used in:** JTextFieldMaxLength
  - **JTextFieldMaxLength** – implements the maximum length property in JTextField
    - **Depends on:** PlainDocumentMaxLength
    - **It is used in:** AddDistributorDialog, profileWindow
  - **CustomPasswordField** – implements executing keyboard shortcut when enter is pressed and focus is on current component. Useful for closing dialogs with this components used – the dialog which asks user for its password.
    - **Depends on:** none
    - **It is used in:** main
- **Windows & dialogs**
  - **MainWindow** – displays the main window of the application
    - **Depends on:** DistributorManager
    - **It is used in:** none
  - **ProfileWindow** – displays the profile window of the distributor
    - **Depends on:** Chart, NonEditableDefaultTableModel, RowHeaderRenderer, JTextFieldMaxLength
    - **It is used in:** none
  - **ChangePasswordDialog** – displays the dialog for changing password
    - **Depends on:** none
    - **It is used in:** none
  - **AddDistributorDialog** – displays the interface for easy addition of distributors to the system. Information input is fully validated.
    - **Depends on:** JTextFieldMaxLength, Static,...
    - **It is used in:** none

- **DistributorTreeDialog** – displays the JTree with checkbox implementation on a single dialog. Provides easy-to-use selection of distributors for the user when inserting points to application.
  - **Depends on:** DistirbutorManager, CheckNode, CheckRenderer, NodeSelectionListener, TreeLabel
  - **It is used in:** none
- **PointsDialog** – displays the dialog with table for entering points for choosed Months
  - **Depends on:** MonthManager, NonEditableDefaultTableModel, RowHeaderRenderer
  - **It is used in:** none
- **ChildrenTreeDialog** – displays the dialog representing the children of particular distributor in a hierarchical way
  - **Depends on:** DistirbutorManager
  - **It is used in:** none
- **ChildrenTableDialog** – displays the dialog showing all children distributors of particular distributor in a table
  - **Depends on:** DistributorManager, NonEditableDefaultTableModel, RowHeaderRenderer
  - **It is used in:** none

## C: The Program

### C1: Source code

#### File readme.txt

1. PROJECT TITLE: Multilevel Marketing Manager
2. PURPOSE OF PROJECT: Application **for** managing human resources in multilevel marketing
3. VERSION or DATE: Created 22/2/2011 - 24/3/2011
4. AUTHOR: Juraj Masar
5. IDE: BlueJ 3.0.4
6. IB SCHOOL: Spojena Skola Novohradska - Gymnazium Jura Hronca, Bratislava, Slovakia

## File Static.java

```
1.  import java.util.Random;
2.
3.  import java.text.DateFormat;
4.  import java.util.Date;
5.  import java.text.SimpleDateFormat;
6.  import java.text.ParseException;
7.
8.  import java.io.File;
9.  /**
10.   * Class full of useful static methods which could be easily reused anywhere
11.   * + definitions of constants for the application
12.   *
13.   * @author Juraj Masar
14.   * @version 0.1
15.   */
16. public class Static
17. {
18.     //definitions of constants in the application
19.     public static final String defaultErrorTitle = "Error";
20.     public static final String defaultWarningTitle = "Warning";
21.     public static final String defaultInfoTitle = "Message";
22.     public static final String defaultPlainTitle = "Message";
23.     public static final String keyIconPath = "resources/key-icon.png";
24.     public static final String iconPath = "resources/user.png";
25.     public static final String profileIconPath = "resources/profile.png";
26.     public static final String profileIconSmallPath = "resources/profileSmall.png";
27.
28.     public static final char defaultSeparator = ';'; //separator used in DAF
29.
```



```

30. public static final String hashSalt = "6eH97dZ90"; //salt for password hash generation
31. public static final String encryptionKey = "5SAcyh8F2vF0"; //key for data encryption
32.
33. public static final String dataDirectory = "data";
34. public static final String filePassword = "data/options.dat";
35. public static final String fileDistributor = "data/distributors.dat";
36. public static final String fileMonthPrefix = "data/dist";
37. public static final String fileMonthSuffix = ".dat";
38.
39. public static final String dateFormat = "dd.MM.yyyy";
40.
41. public static final long monthRecordLength = 50;
42. public static final long distributorRecordLength = 470;
43.
44. /**
45.  * customizes the length of the string given
46.  *
47.  * @param y    a sample parameter for a method
48.  * @return     string customized to the given length
49.  */
50. public static String setStringLength(String s, int numOfChars)
51. {
52.     if (s.length() > numOfChars) //given string is longer
53.     {
54.         s = s.substring(0, numOfChars); //shorten
55.     } else //given string is shorter
56.     {
57.         //prolong
58.         for (int i=s.length(); i<=numOfChars-1;i++) s += " ";
59.     }
60.     return s;

```

```

61.
62.  /**
63.   * Cuts the string given if its length exceeds the maximum length given.
64.   * Otherwise returns original string.
65.   *
66.   * @param s    string to manipulate
67.   * @param max  maximum length of the sting
68.   * @return     the sum of x and y
69.   */
70. public static String cutIfLongerThan(String s, int max)
71. {
72.     if (s.length() > max) //given string is longer
73.     {
74.         return s.substring(0, max); //shorten
75.     } else
76.     {
77.         return s;
78.     }
79. }
80.
81. /**
82.  * counts number of occurences of a given char in a given string
83.  *
84.  * @param s    string to count occurences in
85.  * @param c    char to look for
86.  * @return     number of occurences
87.  */
88. public static int charOccurences(String s, char c)
89. {
90.     if (s == null) return -1;
91.     int count = 0; //count

```

```

92.         for (int i=0;i<=s.length()-1;i++)
93.             if (s.charAt(i) == c) count++;
94.         return count;
95.     }
96.
97.     /**
98.      * Checks whether the string contains only numerical characters.
99.      *
100.     * @param s    string to check
101.     * @return     the sum of x and y
102.     */
103.     public static boolean isNumeric (String s)
104.     {
105.         for (int i=0;i<=s.length()-1;i++)
106.         {
107.             if (s.charAt(i) > 57 || s.charAt(i) < 48) return false;
108.         }
109.         return true;
110.     }
111.
112.
113.     /**
114.     * removes given char from a given string
115.     *
116.     * @param s string to remove char from
117.     * @param c char to remove
118.     * @return  string without given char
119.     */
120.     public static String removeChar(String orig, char c)
121.     {
122.         String s = new String();

```

```

123.         for (int i=0;i<=orig.length()-1;i++)
124.         {
125.             if (orig.charAt(i) != c)
126.             {
127.                 s += orig.charAt(i);
128.             }
129.         }
130.         return s;
131.     }
132.
133.     /**
134.      * creates one string from the array of strings given separated by given separator
135.      * if separator occurs in given data, it is removed.
136.      *
137.      * @param data    array of strings to compile
138.      * @param separator char for separation of strings
139.      * @return        compilation of strings
140.      */
141.     public static String implode(String[] data, char separator)
142.     {
143.         String s = new String();
144.         for (int i=0;i<=data.length-1;i++)
145.         {
146.             s += removeChar(data[i],separator)+separator;
147.         }
148.         return s;
149.     }
150.
151.     /**
152.      * creates one string from the array of strings given separated by default separator
153.      *

```

```

154.      * @param data    array of strings to compile
155.      * @return      compilation of strings
156.      */
157.      public static String implode(String[] data)
158.      {
159.          return implode (data, defaultSeparator);
160.      }
161.
162.
163.      /**
164.       * creates an array of strings from compilation of strings with given separator
165.       *
166.       * @param orig original compilation of strings
167.       * @param separator char for separation of strings
168.       * @return      array of strings
169.       */
170.      /** DOSSIER: SL mastery 8 - user-defined methods */
171.      public static String[] explode(String orig, char separator)
172.      {
173.          if (orig == null) return null;
174.          int count = charOccurrences(orig, separator);
175.          /** DOSSIER: SL mastery 1 - arrays */
176.          String[] data = new String[count];
177.          String current = new String();
178.          int index = 0;
179.          /** DOSSIER: SL mastery 6 - loops */
180.          for (int i=0;i<=orig.length()-1;i++)
181.          {
182.              /** DOSSIER: SL mastery 4 - simple selection */
183.              if (orig.charAt(i) != separator)
184.              {

```

```

185.         current += orig.charAt(i);
186.     } else
187.     {
188.         data[index] = current;
189.         current = new String();
190.         index++;
191.     }
192. }
193. return data;
194. }
195.
196. /**
197.  * creates an array of strings from compilation of strings with default separator
198.  *
199.  * @param orig original compilation of strings
200.  * @return      array of strings
201.  */
202. /** DOSSIER: SL mastery 9 - user-defined methods with parameters */
203. public static String[] explode(String orig)
204. {
205.     return explode (orig, defaultSeparator);
206. }
207.
208. /**
209.  * Removes diacritics from a String given.
210.  *
211.  * Originally from
212.  * http://www.rgagnon.com/javadetails/java-0456.html
213.  *
214.  * @param s      String with diacritics
215.  * @return       String without diacritics

```

```

216.     */
217.     public static String removeDiacritics(String s)
218.     {
219.         String temp = java.text.Normalizer.normalize(s, java.text.Normalizer.Form.NFD);
220.         return temp.replaceAll("[^\\p{ASCII}]", "");
221.     }
222.
223.     /**
224.      * Creates an array of strings from compilation of strings with default separator.
225.      *
226.      * @param arr array of values
227.      * @param left left indx
228.      * @param right right index
229.      * @return integer index
230.      */
231.     private static int quickSortPartition(int arr[], int i, int j)
232.     {
233.         int tmp = 0;
234.         int pivot = arr[(i + j) / 2];
235.
236.         /** DOSSIER: SL mastery 7 - nested loops */
237.         while (i <= j)
238.         {
239.             while (arr[i] < pivot) i++;
240.             while (arr[j] > pivot) j--;
241.             if (i <= j)
242.             {
243.                 tmp = arr[i];
244.                 arr[i] = arr[j];
245.                 arr[j] = tmp;
246.                 i++;

```

```

247.         j--;
248.     }
249. }
250.
251.     return i;
252. }
253.
254. /**
255.  * Sorts given array within indexes given.
256.  *
257.  * @param arr array of values
258.  * @param left left indx
259.  * @param right right index
260.  */
261. private static void quickSort(int arr[], int left, int right)
262. {
263. /** DOSSIER: HL mastery 4 - recursion */
264.     int index = quickSortPartition(arr, left, right);
265.     if (left < index - 1) quickSort(arr, left, index - 1);
266.     if (index < right) quickSort(arr, index, right);
267. }
268.
269. /**
270.  * Sorts given array.
271.  *
272.  * @param arr array of values
273.  */
274. public static void sort(int arr[])
275. {
276.     if (arr != null && arr.length>0)quickSort(arr, 0, arr.length-1);
277. }

```



```

278.
279.  /**
280.   * Produces an array of ints - with numbers which are
281.   * included in all given arrays.
282.   *
283.   * @param in array of arrays of int
284.   * @return int[] array
285.   */
286. /** DOSSIER: HL mastery 19 - arrays of two or more dimensions */
287. public static int[] conjunction(int[][] in)
288. {
289.     if (in.length == 0) return new int[0];
290.     for (int a=0;a<=in.length-1;a++)
291.     {
292.         sort(in[a]);
293.     }
294.
295. /** DOSSIER: HL mastery 5 - merging sorted data structures */
296.
297.     Vector<Integer> v = new Vector<Integer>();
298.
299.     int[] indices = new int[in.length];
300.     for (int i=0;i<=in.length-1;i++) indices[i] = 0;
301.
302.     boolean fail;
303.
304.     for (indices[0]=0;indices[0]<=in[0].length-1;indices[0]++)
305.     {
306.         fail = false;
307.         for (int b=1;b<=in.length-1;b++)
308.         {

```

```

309.         while (indices[b] < in[b].length
310.             && in[b][indices[b]] < in[0][indices[0]])
311.             indices[b]++;
312.         if (indices[b] >= in[b].length ||
313.             in[b][indices[b]] != in[0][indices[0]])
314.         {
315.             fail = true;
316.             break;
317.         }
318.     }
319.     if (!fail) v.pushBack(in[0][indices[0]]);
320. }
321.
322.     return v.toIntArray();
323. }
324.
325. /**
326.  * Generates random number in a given range.
327.  *
328.  * @param from    the beginning of the range
329.  * @param to      the end of the range
330.  * @return        integer number from the range
331.  */
332. public static int random(int from, int to)
333. {
334.     if (from > to) return -1;
335.
336.     return new Random().nextInt(to-from+1) + from;
337. }
338.
339. /**

```

```

340.      * Parses given String into Date
341.      *
342.      * @param s      string to parse
343.      * @return      Date object
344.      */
345.  public static Date stringToDate(String s)
346.  {
347.      try
348.      {
349.          DateFormat formatter = new SimpleDateFormat(dateFormat);
350.          return (Date)formatter.parse(s);
351.      } catch (ParseException e)
352.      {
353.          return null;
354.      }
355.  }
356.
357.
358.  /**
359.   * Parses formats given Date into String
360.   *
361.   * @param d      Date to format
362.   * @return      String
363.   */
364.  /** DOSSIER: SL mastery 10 - user-defined methods with return values */
365.  public static String dateToString(Date d)
366.  {
367.      DateFormat formatter = new SimpleDateFormat(dateFormat);
368.      return formatter.format(d);
369.  }
370.

```

```

371.  /**
372.   * Deletes file by a given path
373.   *
374.   * @param fileName fileName of the file to delete
375.   */
376.  public static void deleteFile (String fileName)
377.  {
378.      try
379.      {
380.          File f = new File(fileName);
381.
382.          if (!f.exists())
383.          {
384.              Dialog.error("Error while deleting file '"+fileName+"':\n"
385.                  +"file does not exist.");
386.              return;
387.          }
388.
389.          if (!f.delete())
390.              Dialog.error("Error while deleting file '"+fileName+"':\n"
391.                  +"deleting failed.");
392.      } catch (SecurityException e)
393.      {
394.          Dialog.error("Error while deleting file '"+fileName+"':\n"
395.              +"exception: "+e.getMessage());
396.      }
397.  }
398.
399.
400.  /**
401.   * Converts given char Array to a single String

```

```

402.      *
403.      * @param ca    char array
404.      * @return     String object
405.      */
406.  public static String charArrayToString(char[] ca)
407.  {
408.      String ret = "";
409.      for (int i=0;i<ca.length;i++) ret += ca[i];
410.      return ret;
411.  }
412.
413.  /**
414.   * Validates the given email address.
415.   *
416.   * @param s      String email address
417.   * @return       true if valid
418.   */
419.  public static boolean validateEmail(String s)
420.  {
421.      if (s == null || s.length() == 0)
422.          return false;
423.      if ((s.indexOf('@')+1 < s.lastIndexOf('.')) &&
424.          (s.indexOf('@')!=0) && (s.lastIndexOf('.')!=s.length()-1))
425.          return true;
426.      return false;
427.  }
428.
429.  /**
430.   * An example of a method - replace this comment with your own
431.   *
432.   * @param y      a sample parameter for a method

```

```

433.      * @return      the sum of x and y
434.      */
435. public static boolean validateLetters(String s)
436. {
437.     for (int i=0;i<s.length();i++)
438.         if (s.charAt(i) < 57 && s.charAt(i) > 48) return false; //is number
439.     return true;
440. }
441.
442.
443. /**
444.  * Validates given String according to given parameters.
445.  * If it is not valid, returns error message.
446.  *
447.  * @param s      input string
448.  * @param type 0 - numeric, 1 - letters and spaces, 2 - email
449.  * @param compulsory if string can't be null
450.  * @param name    to use in error message
451.  * @return      error message
452.  */
453. public static String validateString(String s, int type, boolean compulsory, String name)
454. {
455.     if (compulsory && (s == null || s.length() == 0))
456.         return name+" is a compulsory field!";
457.     if (type == 0 && !Static.isNumeric(s)) //numeric
458.         return name+" has to be numeric!";
459.     if (type == 1 && !Static.validateLetters(s)) //numeric
460.         return name+" can't contain numbers!";
461.     if (type == 2 && !Static.validateEmail(s) && (s != null && s.length() > 0)) //numeric
462.         return name+" has to be a valid email address!";
463.

```

```

464.         return null;
465.     }
466.
467.     /**
468.      * Returns a word representing Month given in number.
469.      *
470.      * @param String    month in number
471.      * @return         String month in world
472.      */
473.     public static String monthByWord(String in)
474.     {
475.         if (Integer.parseInt(in) == 1) return "January";
476.         if (Integer.parseInt(in) == 2) return "February";
477.         if (Integer.parseInt(in) == 3) return "March";
478.         if (Integer.parseInt(in) == 4) return "April";
479.         if (Integer.parseInt(in) == 5) return "May";
480.         if (Integer.parseInt(in) == 6) return "June";
481.         if (Integer.parseInt(in) == 7) return "July";
482.         if (Integer.parseInt(in) == 8) return "August";
483.         if (Integer.parseInt(in) == 9) return "September";
484.         if (Integer.parseInt(in) == 10) return "October";
485.         if (Integer.parseInt(in) == 11) return "November";
486.         if (Integer.parseInt(in) == 12) return "December";
487.         return "";
488.     }
489.
490.     /**
491.      * Encrypts given String according to given key.
492.      *
493.      * @param in    String to encrypt
494.      * @param key   key of the encryption

```

```

495.      * @return      encrypted String
496.      */
497.  public static String encrypt(String in)
498.  {
499.      DesEncrypter encrypter = new DesEncrypter(Static.encryptionKey);
500.      return encrypter.encrypt(in);
501.  }
502.
503.  /**
504.   * Decrypts given String according to given key.
505.   *
506.   * @param in      String to decrypt
507.   * @return      encrypted String
508.   */
509.  public static String decrypt(String in)
510.  {
511.      DesEncrypter encrypter = new DesEncrypter(Static.encryptionKey);
512.      return encrypter.decrypt(in);
513.  }
514.
515. }
516.

```

88



## File Dialog.java

```
1. import javax.swing.JOptionPane;
2. import javax.swing.ImageIcon;
3. import javax.swing.SwingUtilities;
4. import javax.swing.text.*;
5. import javax.swing.*;
6.
7. import java.text.DateFormat;
8. import java.util.Date;
9. import java.text.SimpleDateFormat;
10. import java.text.ParseException;
11. /**
12.  * Contains methods for executing window dialogs.
13.  *
14.  * @author Juraj Masar
15.  * @version 0.1
16.  */
17. public class Dialog
18. {
19.     public static MainWindow mainWindow;
20.     /**
21.      * Displays error message with a given string and title.
22.      *
23.      * @param msg    message to display
24.      * @param title   title of the window
25.      */
26.     public static void error(String msg, String title)
27.     {
28.         JOptionPane.showMessageDialog(null, msg, title, JOptionPane.ERROR_MESSAGE);
29.     }
```

```

30.
31.  /**
32.   * Displays error message with a given string.
33.   *
34.   * @param msg    message to display
35.   */
36. public static void error(String msg)
37. {
38.     error (msg, Static.defaultErrorTitle);
39. }
40.
41. /**
42.   * Displays warning message with a given string and title.
43.   *
44.   * @param msg    message to display
45.   * @param title  title of the window
46.   */
47. public static void warning(String msg, String title)
48. {
49.     JOptionPane.showMessageDialog(null, msg, title, JOptionPane.WARNING_MESSAGE);
50. }
51.
52. /**
53.   * Displays warning message with a given string.
54.   *
55.   * @param msg    message to display
56.   */
57. public static void warning(String msg)
58. {
59.     warning (msg, Static.defaultWarningTitle);
60. }

```

```
61.
62.  /**
63.   * Displays info message with a given string and title.
64.   *
65.   * @param msg    message to display
66.   * @param title  title of the window
67.   */
68. public static void info(String msg, String title)
69. {
70.     JOptionPane.showMessageDialog(null, msg, title, JOptionPane.INFORMATION_MESSAGE);
71. }
72.
73. /**
74.   * Displays info message with a given string.
75.   *
76.   * @param msg    message to display
77.   */
78. public static void info(String msg)
79. {
80.     info (msg, Static.defaultInfoTitle);
81. }
82.
83.
84. /**
85.   * Displays plain message - without icon - with a given string and title.
86.   *
87.   * @param msg    message to display
88.   * @param title  title of the window
89.   */
90. public static void plain(String msg, String title)
91. {
```

```

92.         JOptionPane.showMessageDialog(null, msg, title, JOptionPane.PLAIN_MESSAGE);
93.     }
94.
95.     /**
96.      * Displays info message with a given string.
97.      *
98.      * @param msg    message to display
99.      */
100.    public static void plain(String msg)
101.    {
102.        plain (msg, Static.defaultPlainTitle);
103.    }
104.
105.    /**
106.     * Creates ImageIcon object out of given path.
107.     *
108.     * @param path path to file
109.     * @param description description of the icon
110.     * @return ImageIcon object
111.     */
112.    public static ImageIcon createImageIcon(String path, String description)
113.    {
114.        java.net.URL imgURL = Dialog.class.getResource(path);
115.        if (imgURL != null) {
116.            return new ImageIcon(imgURL, description);
117.        } else {
118.            Dialog.error("Static error: \n couldn't find icon: " + path);
119.            return null;
120.        }
121.    }
122.

```

```

123.      /**
124.       * Displays profile window for given distributor
125.       *
126.       * @param regNum    registration number of the distributor
127.       * @return true if successful
128.       */
129. public static boolean displayProfile(final String regNum)
130. {
131.     DistributorManager.init();
132.     ItemRegNum item = ((ItemRegNum)DistributorManager.listRegNum.find(regNum));
133.     if (item == null) return false;
134.     if (item.window != null && item.window.isDisplayable() && item.window.isShowing())
135.     {
136.         item.window.toFront();
137.         return true;
138.     } else
139.     {
140.         final Distributor d = DistributorManager.getDistributorByRegNum(regNum);
141.         item.window = new ProfileWindow(d);
142.         return true;
143.     }
144. }
145.
146. /**
147.  * Using modal window it asks user to insert month in
148.  * MM/yyyy format. If not valid, query is repeated
149.  * after error message. The inserted String is returned.
150.  *
151.  * @param y    a sample parameter for a method
152.  * @return     the sum of x and y
153.  */

```

```

154. public static String askForMonth()
155. {
156.     //show the month query screen
157.     final String monthFormat = "MM/yyyy";
158.     JFormattedTextField monthField = new JFormattedTextField();
159.     //submit the dialog when enter is pressed
160.     monthField.addKeyListener(new java.awt.event.KeyAdapter()
161.     {
162.         public void keyPressed(java.awt.event.KeyEvent kEvt)
163.         {
164.             if (kEvt.getKeyCode() == java.awt.event.KeyEvent.VK_ENTER)
165.             {
166.                 kEvt.consume();
167.                 // auto generate TAB + Enter keypress events
168.                 try
169.                 {
170.                     java.awt.Robot robot = new java.awt.Robot();
171.                     robot.setAutoDelay(100);
172.                     robot.keyPress(java.awt.event.KeyEvent.VK_TAB);
173.                     robot.keyPress(java.awt.event.KeyEvent.VK_SPACE);
174.                     robot.keyRelease(java.awt.event.KeyEvent.VK_SPACE);
175.                 }
176.                 catch (java.awt.AWTException awtEx) {
177.                     awtEx.printStackTrace();
178.                 }
179.             }
180.         }
181.     });
182.
183.     monthField.setFormatterFactory(
184.         new DefaultFormatterFactory(new DateFormatter(

```

```

185.         new java.text.SimpleDateFormat(monthFormat)))));
186. monthField.setText("01/2011");
187. Object[] obj = {"Please, insert month to which you want to add points:\n",
188.                 monthField};
189. Object stringArray[] = {"Continue...", "Cancel"};
190.
191. if (JOptionPane.showOptionDialog(
192.         null,
193.         obj,
194.         "Insert month",
195.         JOptionPane.YES_NO_OPTION,
196.         JOptionPane.QUESTION_MESSAGE,
197.         null,
198.         //icon from
199.         //http://www.iconarchive.com/show/security-icons-by-aha-soft/key-icon.html
200.         stringArray,
201.         null) == JOptionPane.YES_OPTION)
202. {
203.     String s = monthField.getText();
204.
205.     //check format
206.     try
207.     {
208.         DateFormat formatter = new SimpleDateFormat(monthFormat);
209.         if ((Date)formatter.parse(s) != null) //if the format is valid
210.         {
211.             return s;
212.         }
213.     } catch (ParseException e)
214.     {
215.         error("The inserted format is not valid.");

```

```
216.         return askForMonth();
217.     }
218. }
219. return null;
220. }
221. }
222.
```



## File Main.java

```
1. import javax.swing.JOptionPane;
2. import javax.swing.JPasswordField;
3. import javax.swing.UIManager;
4. import javax.swing.JOptionPane;
5. import java.io.*;
6.
7. /**
8.  * Main class of the program - it controls all other classes
9.  *
10.  * @author Juraj Masar
11.  * @version 0.1
12.  */
13. /** DOSSIER: SL mastery 2 - user-defined objects */
14. public class Main
15. {
16.     /**
17.      * Bootstrapper of the application.
18.      * Loads the rest of the application.
19.      */
20.     public static void main()
21.     {
22.         //set the windows look
23.         try
24.         {
25.             UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
26.         } catch (Exception e) {
27.             e.printStackTrace();
28.         }
29.
```

```

30.      //check if dictionary Data exists
31.
32.      File dir = new File(Static.dataDirectory);
33.
34.      if (!dir.exists() && !dir.mkdir())
35.      {
36.          Dialog.error ("Error: Data directory \""+Static.dataDirectory+"\" does not exist\n"
37.              +"and cannot be created."
38.          );
39.          return;
40.      }
41.
42.
43.      //check the data consistency
44.      Distributor d = DistributorManager.getDistributorBySponsor("0");
45.      if (d == null)
46.      {
47.          //data is inconcistent
48.          int n = JOptionPane.showConfirmDialog(
49.              null,
50.              "Data of the application is inconsistent.\n"
51.              +"The root distributor cannot be found.\n"
52.              +"Create initial data?",
53.              "Create initial data?",
54.              JOptionPane.YES_NO_OPTION);
55.
56.          if (n == 0) //yes
57.          {
58.              String s = (String)JOptionPane.showInputDialog(
59.                  null,
60.                  "Please, insert the registration number of the \n"

```

```

61.         + "root distributor (the user of the application): ",
62.         "Installation",
63.         JOptionPane.PLAIN_MESSAGE,
64.         null,
65.         null,
66.         "10000");
67.         if (s == null) return;
68.         if (s != null && s.length() > 0 && Static.isNumeric(s))
69.         {
70.             createInitData(s);
71.             Dialog.info ("Initial data has been successfully created.\n"
72.                 + "Password has been set to \"" + s + "\"");
73.         } else
74.         {
75.             Dialog.error ("Registration number has to be numeric.");
76.             return;
77.         }
78.     } else
79.     {
80.         return;
81.     }
82.     //show the password screen
83.     CustomPasswordField passwordField = new CustomPasswordField();
84.
85.     Object[] obj = {"Welcome to Multilevel Marketing Manager.\n"
86.         + "Please, enter password to continue: ", passwordField};
87.     Object stringArray[] = {"Login", "Cancel"};
88.
89.     if (JOptionPane.showOptionDialog(
90.         null,
91.         obj,

```

```

92.         "Multilevel Marketing Manager - Login",
93.         JOptionPane.YES_NO_OPTION,
94.         JOptionPane.QUESTION_MESSAGE,
95.         Dialog.createImageIcon(Static.keyIconPath, "Login"),
96.         //icon from
97.         //http://www.iconarchive.com/show/security-icons-by-aha-soft/key-icon.html
98.         stringArray,
99.         null) == JOptionPane.YES_OPTION)
100.    {
101.        String password = Static.charArrayToString(passwordField.getPassword());
102.
103.        if (Password.checkPassword(password))
104.        {
105.            Dialog.mainWindow = new MainWindow();
106.        }
107.        else
108.        {
109.            Dialog.warning ("The password you have inserted is invalid.");
110.            main();
111.        }
112.    }
113. }
114.
115.
116. /**
117.  * Creates initial data required for the program.
118.  * Sets the password empty - "";
119.  * Adds the root distributor with sponsor 0;
120.  *
121.  * @param regNum registration number of the root distributor
122.  */

```

```

123. public static void createInitData(String regNum)
124. {
125.     //delete all existing data
126.     DirectAccessFile f = new DirectAccessFile (Static.fileDistributor,
127.         Static.distributorRecordLength);
128.     while (f.getNumberOfRecords() != 0)
129.     {
130.         f.delete((long) Static.random(0, (int)f.getNumberOfRecords()-1));
131.     }
132.
133.     //set the password
134.     Password.setPassword("");
135.
136.     //add the first distributor
137.     DistributorManager.addDistributor(regNum,
138.         "0",
139.         "SK",
140.         "1.1.2010",
141.         "",
142.         "Name",
143.         "Surname",
144.         "1.1.1950",
145.         "valid@email.com",
146.         "00 421 907 123 456",
147.         "Please, replace this data with your details."
148.         );
149.     }
150. }

```

## File TextFileWriter.java

```
1. import java.io.*;
2.
3. /**
4.  * Used for writing text content into sequential files
5.  *
6.  * @author Juraĵ Masar
7.  * @version 0.1
8.  */
9. public class TextFileWriter
10. {
11.     private boolean ready; //is file ready for writing?
12.     private PrintWriter pw; //Java's class for sequential files manipulation
13.
14.     /**
15.      * constructor - opens the file and get it ready for writing or throws an I/O exception
16.      *
17.      * @param fileName name of file for writing
18.      */
19.     public TextFileWriter(String fileName)
20.     {
21.         try
22.         {
23.             pw = new PrintWriter (new FileOutputStream (fileName));
24.             ready = true;
25.         } catch (IOException e)
26.         {
27.             ready = false;
28.             Dialog.error ("Opening file "+fileName+" was unsuccessful.");
29.         }
```

```
30.     }
31.
32.     /**
33.      * checks whether the writing to file is ready
34.      *
35.      * @return true if the file is ready
36.      */
37.     public boolean isReady()
38.     {
39.         return ready;
40.     }
41.
42.     /**
43.      * writes text into the current position of sequential file
44.      *
45.      * @param line text to be written
46.      * @return void
47.      */
48.     public void write(String line)
49.     {
50.         if (ready) pw.print (line);
51.     }
52.
53.     /**
54.      * writes line ended by \n - newline character into the current position in sequential file
55.      *
56.      * @param line string to be written into file
57.      */
58.     public void writeln(String line)
59.     {
60.         if (ready) pw.println (line);
```

```
61.     }
62.
63.     /**
64.      * closes the writing stream to the file
65.      *
66.      */
67.     public void close()
68.     {
69.         if (ready)
70.         {
71.             pw.flush();
72.             pw.close();
73.             ready = false;
74.         }
75.     }
76.
77. }
78.
```



## File TextFileReader.java

```
1. import java.io.*;
2. /**
3.  * Used for reading text from a sequential file.
4.  *
5.  * @author Juraj Masar
6.  * @version 0.1
7.  */
8. public class TextFileReader
9. {
10.     private BufferedReader br; //java's object for reading sequential files
11.     private String line;
12.     private boolean ready; //states whether the file is ready
13.     /**
14.      * constructor for objects of class TextFileReader.
15.      * initializes the access to the file or throws an exception.
16.      *
17.      * @params fileName name of the file to read
18.      */
19.     public TextFileReader(String fileName)
20.     {
21.         line = null;
22.         ready = false;
23.         try
24.         {
25.             br = new BufferedReader(new FileReader(fileName));
26.             ready = true;
27.         }
28.         catch (IOException e)
29.         {
```

```

30.         ready = false;
31.         Dialog.error("Opening file "+fileName+" was unsuccessful.");
32.     }
33. }
34.
35. /**
36.  * checks whether file is ready for reading
37.  *
38.  * @return      true if the file is ready for reading
39.  */
40. public boolean isReady()
41. {
42.     return ready;
43. }
44.
45. /**
46.  * reads and returns current line from the sequential file or throws an I/O exception
47.  *
48.  * @return      current line in the sequential file
49.  */
50. public String readLine()
51. {
52.     String line = "";
53.     try
54.     {
55.         if (ready) line = br.readLine();
56.         if (line == null) ready = false;
57.         return line;
58.     }
59.     catch (IOException e)
60.     {

```

```
61.         ready = false;
62.         Dialog.error("Reading from file was unsuccessful.");
63.         return null;
64.     }
65. }
66.
67. }
68.
```

## File DirectAccessFile.java

```
1.  import java.io.*;
2.  /**
3.   * Provides manipulation of random access files
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8.  public class DirectAccessFile
9.  {
10.     private RandomAccessFile f; //pointer to random access file
11.     private long recordLength; //defines number of bytes of one record in a file
12.     private boolean ready; //state of the file
13.
14.     /**
15.      * constructor for objects of class DirectAccessFile
16.      * opens file for reading & writing or throws an I/O exception
17.      *
18.      * @param fileName name of file to work with
19.      * @param          recordLength number of bytes of one record in a file
20.      */
21.     public DirectAccessFile(String fileName, long recordLength)
22.     {
23.         ready = false;
24.         try
25.         {
26.             f = new RandomAccessFile (fileName, "rw");
27.             this.recordLength = recordLength;
28.             ready = true;
29.         }
```

```

30.         catch (IOException e)
31.         {
32.             ready = false;
33.             Dialog.error ("Opening file "+fileName+" was unsuccessful. \n Error: #1 "
34.                             +e.getMessage());
35.         }
36.     }
37.
38.     /**
39.      * checks if the file is ready for manipulation
40.      *
41.      * @return      true if the file is ready
42.      */
43.     public boolean isReady()
44.     {
45.         return ready;
46.     }
47.
48.     /**
49.      * returns length of the file in bytes or throws an I/O exception
50.      *
51.      * @return      length of the file in bytes
52.      */
53.     public long getLength()
54.     {
55.         if (ready)
56.             try
57.             {
58.                 return f.length();
59.             }
60.             catch (IOException e)

```

```

61.         {
62.             Dialog.error  ("The following error occured while working with file: #2 \n "
63.                             +e.getMessage());
64.             return -1;
65.         }
66.     else
67.     {
68.         return -1;
69.     }
70. }
71.
72.
73. /**
74.  * returns length of one record
75.  *
76.  * @return      length of one record
77.  */
78. public long getRecordLength()
79. {
80.     if (ready)
81.         return recordLength;
82.     else return -1;
83. }
84.
85. /**
86.  * according to file length and length of one record it returns num. of records in a file.
87.  *
88.  * @return      number of records in a file
89.  */
90. public long getNumberOfRecords()
91. {

```

```

92.         if (ready)
93.             return getLength()/recordLength;
94.         else return -1;
95.     }
96.
97.     /**
98.      * moves the file pointer to a different record or throws an exception
99.      *
100.     * @param i    position of the record
101.     */
102. public void goToRecord (long i)
103. {
104.     try
105.     {
106.         if (ready && i >= 0)
107.             f.seek (i*recordLength); //uses RandomAccessFile method seek
108.     }
109.     catch (IOException e)
110.     {
111.         Dialog.error ("The following error occurred while working with file: \n #3 "
112.                        +e.getMessage());
113.     }
114. }
115.
116. /**
117.  * writes string to the current record or throws an exception
118.  *
119.  * @param s    string to write into file
120.  */
121. public void write(String s)
122. {

```

```

123.     try
124.     {
125.         //method editString firstly customizes the length of the string
126.         //to the length recordLength+2
127.         //-2 is there for the metadata of UTF8
128.         if (ready) f.writeUTF(Static.setStringLength(s,(int)recordLength-2));
129.     }
130.     catch (IOException e)
131.     {
132.         Dialog.error ("The following error occurred while working with file: \n #4 "
133.             +e.getMessage());
134.     }
135. }
136.
137. /**
138.  * writes string to the record given or throws an exception
139.  *
140.  * @param s    string to write into file
141.  * @param record    position to which string should be written
142.  */
143. public void write(String s, long record)
144. {
145.     /** DOSSIER: HL mastery 1 - adding data to RAF by seek method */
146.     if (ready)
147.     {
148.         goToRecord(record);
149.         write(s);
150.     }
151. }
152.
153. /**

```



```

154.      * reads and returns string - content of current record or throws an exception
155.      *
156.      * @return      string - content of the current record
157.      */
158.  public String read ()
159.  {
160.      if (ready)
161.      {
162.          try
163.          {
164.              //readUTF returns all the characters until the null character \00
165.              //trim - function of a String - removes all spaces at the beginning and end of the string
166.              return f.readUTF().trim();
167.          }
168.          catch (IOException e)
169.          {
170.              Dialog.error ("The following error occured while working with file: \n #5 "
171.                  +e.getMessage());
172.              return null;
173.          }
174.      } else
175.      {
176.          return null;
177.      }
178.  }
179.
180.  /**
181.      * reads and returns string - content of record given or throws an exception
182.      *
183.      * @return      string - content of the record given
184.      */

```

```

185.     public String read (long record)
186.     {
187.         if (ready)
188.         {
189.             goToRecord (record);
190.             return read();
191.         } else
192.         {
193.             return null;
194.         }
195.     }
196.
197.     /**
198.      * closes the pointer to the random access file or throws an exception
199.      *
200.      */
201.     public void close ()
202.     {
203.         if (ready)
204.         {
205.             ready = false;
206.             try
207.             {
208.                 f.close(); //sends close signal to the RandomAccessFile object
209.             }
210.             catch (IOException e)
211.             {
212.                 Dialog.error ("The following error occurred while working with file: #6 \n "
213.                               +e.getMessage());
214.             }
215.         }

```

```

216.     }
217.
218.
219.     /**
220.      * shortens file to the current number of records - the rest of the file is cut off
221.      * or throws an exception
222.      *
223.      */
224.     public void shortenFile ()
225.     {
226.         if (ready)
227.         {
228.             try
229.             {
230.                 //uses the method of RandomAccessFile setLength to shorten the file
231.                 f.setLength((getNumberOfRecords()-1)*recordLength);
232.             }
233.             catch (IOException e)
234.             {
235.                 Dialog.error ("The following error occurred while working with file: #7 \n "
236.                               +e.getMessage());
237.             }
238.         }
239.     }
240.
241.     /**
242.      * appends string to the end of the file
243.      *
244.      * @param s      string to append
245.      */
246.     public void append(String s)

```

```

247.     {
248.         if (ready) write (s, getNumberOfRecords());
249.     }
250.
251.     /**
252.      * Deletes the current record.
253.      *
254.      * @param position  number of record
255.      * @return true if successful
256.      */
257.     public boolean delete(long position)
258.     {
259.         /** DOSSIER: HL mastery 2 - deleting data from RAF by seek method */
260.         if (ready && position < getNumberOfRecords())
261.         {
262.             write (read(getNumberOfRecords()-1), position);
263.             shortenFile();
264.             return true;
265.         }
266.         else return false;
267.     }
268.
269. }
270.

```

## File DesEncrypter.java

```
1. import javax.crypto.Cipher;
2. import javax.crypto.SecretKey;
3. import javax.crypto.IllegalBlockSizeException;
4. import java.io.UnsupportedEncodingException;
5. import java.security.spec.*;
6. import javax.crypto.spec.PBEKeySpec;
7. import javax.crypto.SecretKeyFactory;
8. import javax.crypto.spec.PBEParameterSpec;
9. import org.apache.commons.codec.binary.Base64;
10. /**
11.  * Encrypting and decrypting interface for Strings.
12.  * All Strings remain in Base64!
13.  *
14.  * Originally from:
15.  * http://www.exampledepot.com/egs/javax.crypto/PassKey.html
16.  *
17.  * @author Juraj Masar
18.  * @version 0.1
19.  */
20. public class DesEncrypter
21. {
22.     Cipher ecipher;
23.     Cipher dcipher;
24.
25.     // 8-byte Salt
26.     byte[] salt = {
27.         (byte)0xA9, (byte)0x9B, (byte)0xC8, (byte)0x32,
28.         (byte)0x56, (byte)0x35, (byte)0xE3, (byte)0x03
29.     };
```

```

30.
31. // Iteration count
32. int iterationCount = 19;
33. /**
34.  * Constructor - initializes the object for given
35.  * passPhrase.
36.  *
37.  * @param passPhrase passPhase for encryption
38.  */
39. DesEncrypter(String passPhrase)
40. {
41.     try
42.     {
43.         // Create the key
44.         KeySpec keySpec = new PBEKeySpec(passPhrase.toCharArray(), salt, iterationCount);
45.         SecretKey key = SecretKeyFactory.getInstance(
46.             "PBKDF2WithHmacSHA1").generateSecret(keySpec);
47.         ecipher = Cipher.getInstance(key.getAlgorithm());
48.         dcipher = Cipher.getInstance(key.getAlgorithm());
49.
50.         // Prepare the parameter to the ciphers
51.         AlgorithmParameterSpec paramSpec = new PBKDF2ParameterSpec(salt, iterationCount);
52.
53.         // Create the ciphers
54.         ecipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
55.         dcipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
56.     }
57.     catch (java.security.InvalidAlgorithmParameterException e) {}
58.     catch (java.security.spec.InvalidKeySpecException e) {}
59.     catch (javax.crypto.NoSuchPaddingException e) {}
60.     catch (java.security.NoSuchAlgorithmException e) {}

```

```

61.         catch (java.security.InvalidKeyException e)
62.         {
63.             Dialog.error(e.getMessage());
64.         }
65.
66.     }
67.
68.     /**
69.      * Encrypts given String
70.      *
71.      * @param str String to encrypt
72.      */
73.     public String encrypt(String str)
74.     {
75.         if (str == null) return null;
76.         try
77.         {
78.             // Encode the string into bytes using utf-8
79.             byte[] utf8 = str.getBytes("UTF8");
80.
81.             // Encrypt
82.             byte[] enc = ecipher.doFinal(utf8);
83.
84.             // Encode bytes to base64 to get a string
85.             return new String(Base64.encodeBase64(enc), "UTF8");
86.         } catch (javax.crypto.BadPaddingException e) {}
87.         catch (IllegalBlockSizeException e) {}
88.         catch (UnsupportedEncodingException e) {}
89.         catch (java.io.IOException e) {}
90.
91.     return null;

```

```

92.     }
93.
94.     /**
95.      * Decrypts given String
96.      *
97.      * @param str String to decrypt
98.      */
99.     public String decrypt(String str)
100.    {
101.        if (str == null) return null;
102.        try
103.        {
104.            // Decode base64 to get bytes
105.            byte[] dec = Base64.decodeBase64(str);
106.
107.            // Decrypt
108.            byte[] utf8 = dcipher.doFinal(dec);
109.
110.            // Decode using utf-8
111.            return new String(utf8, "UTF8");
112.        } catch (javax.crypto.BadPaddingException e) {}
113.        catch (IllegalBlockSizeException e) {}
114.        catch (UnsupportedEncodingException e) {}
115.        catch (java.io.IOException e) {}
116.
117.        return null;
118.    }
119. }

```



## File DistributorManager.java

```
1.
2.  /**
3.   * Manipulates Distributor entities and their info in file.
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8.  public class DistributorManager
9.  {
10.     private static DirectAccessFile f; //file with records of distributors
11.     private static TreeItem root; //pointer to first TreeItem in the structure
12.     public static List listRegNum, listName, listSurname, listSponsor;
13.
14.     /**
15.      * Populates Lists with information from records
16.      *
17.      */
18.     private static void populateLists()
19.     {
20.         init();
21.
22.         //initialize queue for populating tree of distributors
23.         Vector<TreeItem> queue = new Vector<TreeItem> ();
24.
25.         //create lists
26.         listRegNum      = new List(true);
27.         listName         = new List(true);
28.         listSurname      = new List(true);
29.         listSponsor      = new List(true);
```

```

30.
31.    //populate lists
32.    for (int i=0;i<=f.getNumberOfRecords()-1;i++)
33.    {
34.        Distributor d = new Distributor();
35.        d.readFromFile(f,i);
36.
37.        //set the root for tree of distributors
38.        if (d.getSponsor().trim().equals("0"))
39.        {
40.            root = new TreeItem (d.getRegNum());
41.            root.position = i;
42.            root.name = d.getName();
43.            root.surname = d.getSurname();
44.            root.registrationDate = Static.stringToDate(d.getRegistrationDate());
45.            queue.pushBack(root);
46.        }
47.        //populating
48.        addToLists(d, i);
49.    }
50.
51.    //fix listRegNum -> tree for item with sponsor 0 bug
52.    ItemSponsor itemSponsor = (ItemSponsor)listSponsor.find("0");
53.    if (itemSponsor != null) itemSponsor.itemRegNum.treeItem = root;
54.    //populateTree
55.    while (!queue.isEmpty())
56.    {
57.        TreeItem act = queue.popFront();
58.        Item[] distributors = listSponsor.findMore(act.regNum);
59.        if (distributors != null)
60.        {

```

```

61.         TreeItem[] children = new TreeItem[distributors.length];
62.         for (int i=0;i<distributors.length;i++)
63.         {
64.             TreeItem child = new TreeItem (((ItemSponsor) distributors[i]).regNum);
65.
66.             //set properties for new child
67.             child.position = distributors[i].getPos();
68.             child.parent = act;
69.             child.name = ((ItemSponsor) distributors[i]).name;
70.             child.surname = ((ItemSponsor) distributors[i]).surname;
71.             child.registrationDate = ((ItemSponsor) distributors[i]).registrationDate;
72.             children[i] = (TreeItem) child.clone();
73.
74.             //set new pointer from listRegNum to tree
75.             ((ItemSponsor) distributors[i]).itemRegNum.treeItem = children[i];
76.
77.             queue.pushBack(children[i]);
78.         }
79.
80.         act.children = children.clone();
81.     } else
82.         act.children = null;
83. }
84.
85. }
86.
87. /**
88.  * Prints the tree structure of distributors in the system.
89.  * For debugging purposes.
90.  *
91.  */

```

```

92.     public static void printTree ()
93.     {
94.         init();
95.         if (root != null) root.print(0);
96.     }
97.
98.
99.     /**
100.      * Initializes static variables of the object.
101.      *
102.      */
103.     public static void init()
104.     {
105.         if (!(f instanceof DirectAccessFile))
106.         {
107.             //initialize pointer to DAF
108.             f = new DirectAccessFile (Static.fileDistributor, Static.distributorRecordLength);
109.
110.             //create lists
111.             listRegNum      = new List(true);
112.             listName        = new List(true);
113.             listSurname     = new List(true);
114.             listSponsor     = new List(true);
115.
116.             if (getCount() > 0) populateLists();
117.         }
118.     }
119.
120.
121.     /**
122.      * Returns the number of distributors in the system.

```

```

123.      *
124.      * @return      the number of distributors in the system
125.      */
126.  public static int getCount()
127.  {
128.      init();
129.      return (int) f.getNumberOfRecords();
130.  }
131.
132.  /**
133.   * Adds new distributor to the system.
134.   * It creates new record in DAF and also adds new
135.   * item to particular distributor lists.
136.   *
137.   * @param regNum      registration number
138.   * @param sponsor     sponsor's registration number
139.   * @param country     distributor's country
140.   * @param registrationDate registration date
141.   * @param status      distributor's status
142.   * @param name        distributor's name
143.   * @param surname     distributor's surname
144.   * @param birthDate   distributor's birthDate
145.   * @param email       distributor's email
146.   * @param telephone   distributor's telephone
147.   * @param note        note about distributor
148.   */
149.  public static boolean addDistributor(
150.      String regNum,
151.      String sponsor,
152.      String country,
153.      String registrationDate,

```

```

154.     String status,
155.     String name,
156.     String surname,
157.     String birthDate,
158.     String email,
159.     String telephone,
160.     String note
161. )
162. {
163.     init();
164.     if (!Static.isNumeric(regNum)) return false;
165.     Distributor d = new Distributor (regNum);
166.
167.     //check if there is no such regNum in the system
168.     if (listRegNum.findPosition(regNum) == -1)
169.     {
170.         d.setSponsor(sponsor);
171.         d.setCountry(country);
172.         d.setRegistrationDate(registrationDate);
173.         d.setStatus(status);
174.         d.setName(name);
175.         d.setSurname(surname);
176.         d.setBirthDate(birthDate);
177.         d.setEmail(email);
178.         d.setTelephone(telephone);
179.         d.setNote(note);
180.
181.         d.appendToFile(f);
182.
183.         //add to lists
184.         //addToLists(d,f.getNumberOfRecords()-1);

```

```

185.         populateLists();//TreeItem uses fixed array for children
186.
187.         return true;
188.     } else
189.     {
190.         //there is a distributor with such regNum
191.         return false;
192.     }
193. }
194.
195.
196.
197. /**
198.  * Adds distributor given in parameters to lists
199.  *
200.  * @param regNum          registration number
201.  * @param sponsor         sponsor's registration number
202.  * @param country         distributor's country
203.  * @param registrationDate registration date
204.  * @param status          distributor's status
205.  * @param name            distributor's name
206.  * @param surname         distributor's surname
207.  * @param birthDate       distributor's birthDate
208.  * @param email           distributor's email
209.  * @param telephone       distributor's telephone
210.  * @param note            note about distributor
211.  * @param position        position of record in DAF
212.  */
213. private static void addToLists(
214.     String regNum,
215.     String sponsor,

```

```

216.     String country,
217.     String registrationDate,
218.     String status,
219.     String name,
220.     String surname,
221.     String birthDate,
222.     String email,
223.     String telephone,
224.     String note,
225.     long position //position of this record in file
226. )
227. {
228.     init();
229.     Distributor d = new Distributor (regNum);
230.
231.     d.setSponsor(sponsor);
232.     d.setCountry(country);
233.     d.setRegistrationDate(registrationDate);
234.     d.setStatus(status);
235.     d.setName(name);
236.     d.setSurname(surname);
237.     d.setBirthDate(birthDate);
238.     d.setEmail(email);
239.     d.setTelephone(telephone);
240.     d.setNote(note);
241.
242.     addToLists(d, position);
243. }
244.
245. /**
246.  * Adds distributor given in parameters to lists

```



```

247.      *
248.      * @param d          object Distributor
249.      * @param position    position of record in DAF
250.      */
251.  private static void addToLists(Distributor d, long position)
252.  {
253.      init();
254.      ItemRegNum clone,itemRegNum = new ItemRegNum(Static.removeDiacritics(d.getRegNum()).toLowerCase(),
255.          position,null);
256.      clone = (ItemRegNum) itemRegNum.clone();
257.      listRegNum.add(clone);
258.      listName.add(Static.removeDiacritics(d.getName()).toLowerCase(), position);
259.      listSurname.add(Static.removeDiacritics(d.getSurname()).toLowerCase(), position);
260.      listSponsor.add(new ItemSponsor(Static.removeDiacritics(d.getSponsor()).toLowerCase(),
261.          position,null,d.getRegNum(),d.getRegistrationDate(),clone,d.getName(),
262.          d.getSurname()));
263.  }
264.
265.
266.  /**
267.   * Reads from file and returns array of objects
268.   * of all distributors in the system.
269.   *
270.   * @return    array of distributor objects
271.   *
272.   */
273.  public static Distributor[] getAllDistributors ()
274.  {
275.      init();
276.      Distributor[] distributors = new Distributor[(int) f.getNumberOfRecords()];
277.

```

```

278.         for (int i=0;i<=f.getNumberOfRecords()-1;i++)
279.         {
280.             distributors[i] = new Distributor();
281.             distributors[i].readFromFile(f,(long) i);
282.         }
283.
284.         return distributors;
285.     }
286.
287.     /**
288.      * Returns array of Distributors from records from given positions.
289.      *
290.      * @param v    vector of positions (long)
291.      * @return     array of distributors
292.      */
293.     public static Distributor[] getDistributorsFromPositions(Vector<Long> v)
294.     {
295.         init();
296.         Distributor[] distributors = new Distributor[v.size()];
297.
298.         Distributor d = new Distributor();
299.         int i=0;
300.         while (!v.isEmpty())
301.         {
302.             d.readFromFile(f, v.popFront());
303.             distributors[i] = (Distributor) d.clone();
304.             i++;
305.         }
306.
307.         return distributors;
308.     }

```

```

309.
310.  /**
311.   * Returns array of Distributors from records from given positions.
312.   *
313.   * @param v    vector of positions (long)
314.   * @return     array of distributors
315.   */
316. public static Distributor[] getDistributorsFromPositions(int[] positions)
317. {
318.     init();
319.     Distributor[] distributors = new Distributor[positions.length];
320.
321.     Distributor d = new Distributor();
322.     int index = 0;
323.     for (int i=0;i<=positions.length-1;i++)
324.     {
325.         d.readFromFile(f,(long) positions[i]);
326.         distributors[index] = (Distributor) d.clone();
327.         index++;
328.     }
329.
330.     return distributors;
331. }
332.
333. /**
334.   * Return array of distributors by search in a list
335.   *
336.   * @param list list of distributors
337.   * @param needle string to find
338.   * @return     array of distributors
339.   */

```

```

340. public static Distributor[] findInList(List list, String needle)
341. {
342.     init();
343.     return getDistributorsFromPositions(list.findPositions(needle).toIntArray());
344. }
345.
346. /**
347.  * Find distributors according to given parameters in String.
348.  *
349.  * @param regNum    registration number in string
350.  * @param firstName first name of distributor
351.  * @param lastName  last name of distributor
352.  * @return          array of distributors
353.  */
354. public static Distributor[] find(String regNum, String name, String surname)
355. {
356.     init();
357.     int c = 0;
358.     if (regNum != null && !regNum.isEmpty()) c++;
359.     if (name != null && !name.isEmpty()) c++;
360.     if (surname != null && !surname.isEmpty()) c++;
361.
362.     int[][] a = new int[c][];
363.     int i=0;
364.
365.     if (regNum != null && !regNum.isEmpty())
366.     {
367.         a[i] = listRegNum.findPositionsSubstring(Static.removeDiacritics(regNum).toLowerCase())
368.             .toIntArray();
369.         i++;
370.     }

```

```

371.         if (name != null && !name.isEmpty())
372.         {
373.             a[i] = listName.findPositionsSubstring(Static.removeDiacritics(name).toLowerCase())
374.                 .toIntArray();
375.             i++;
376.         }
377.         if (surname != null && !surname.isEmpty())
378.         {
379.             a[i] = listSurname.findPositionsSubstring(Static.removeDiacritics(surname).toLowerCase())
380.                 .toIntArray();
381.             i++;
382.         }
383.         int[] arr = Static.conjunction(a);
384.         return getDistributorsFromPositions(arr);
385.     }
386.
387.     /**
388.      * Retrieves Distributor object by given regNum.
389.      *
390.      * @param regNum    registration number of the distributor
391.      * @return          distributor object
392.      */
393.     public static Distributor getDistributorByRegNum(String regNum)
394.     {
395.         /** DOSSIER: HL mastery 3 - Searching for a specified data in a file */
396.         init();
397.         long position = listRegNum.findPosition(regNum);
398.         if (position == -1) return null;
399.         else
400.         {
401.             Distributor d = new Distributor ();

```

```

402.         d.readFromFile(f, position);
403.         return d;
404.     }
405. }
406.
407.
408. /**
409.  * Edits distributor's record in the database
410.  * according to given information.
411.  *
412.  * @param d    distributor object with new information
413.  * @param position position in the file
414.  * @return     true if the editing was successful
415.  */
416. public static boolean editDistributor(Distributor d, long position)
417. {
418.     init();
419.     if (f.getNumberOfRecords() > position)
420.     {
421.         d.writeToFile(f, position);
422.         populateLists();
423.         return true;
424.     } else
425.     {
426.         return false;
427.     }
428. }
429.
430. /**
431.  * Edits distributor's record in the database
432.  * according to given information.

```

```

433.      *
434.      * @param regNum      registration number
435.      * @param sponsor      sponsor's registration number
436.      * @param country      distributor's country
437.      * @param registrationDate registration date
438.      * @param status      distributor's status
439.      * @param name      distributor's name
440.      * @param surname      distributor's surname
441.      * @param birthDate      distributor's birthDate
442.      * @param email      distributor's email
443.      * @param telephone      distributor's telephone
444.      * @param note      note about distributor
445.      * @param position      record number in the file
446.      * @return true if editing was successful
447.      */
448.      public static boolean editDistributor(
449.          String regNum,
450.          String sponsor,
451.          String country,
452.          String registrationDate,
453.          String status,
454.          String name,
455.          String surname,
456.          String birthDate,
457.          String email,
458.          String telephone,
459.          String note,
460.          long position
461.      )
462.      {
463.          init();

```

```

464.     Distributor d = new Distributor (regNum);
465.
466.     d.setSponsor(sponsor);
467.     d.setCountry(country);
468.     d.setRegistrationDate(registrationDate);
469.     d.setStatus(status);
470.     d.setName(name);
471.     d.setSurname(surname);
472.     d.setBirthDate(birthDate);
473.     d.setEmail(email);
474.     d.setTelephone(telephone);
475.     d.setNote(note);
476.
477.     return editDistributor (d, position);
478. }
479.
480. /**
481.  * Edits distributor's record in the database
482.  * according to given information.
483.  *
484.  * @param d    distributor object with new information
485.  * @param regNum registration number
486.  * @return     true if editing was successful
487.  */
488. public static boolean editDistributor(Distributor d)
489. {
490.     init();
491.     return editDistributor (d,listRegNum.findPosition(d.getRegNum()));
492. }
493.
494.

```



```

495.  /**
496.   * Edits distributor's record in the database
497.   * according to given information.
498.   *
499.   * @param regNum      registration number
500.   * @param sponsor     sponsor's registration number
501.   * @param country     distributor's country
502.   * @param registrationDate registration date
503.   * @param status      distributor's status
504.   * @param name         distributor's name
505.   * @param surname     distributor's surname
506.   * @param birthDate   distributor's birthDate
507.   * @param email       distributor's email
508.   * @param telephone   distributor's telephone
509.   * @param note        note about distributor
510.   * @return true if editing was successful
511.  */
512.  public static boolean editDistributor(
513.      String regNum,
514.      String sponsor,
515.      String country,
516.      String registrationDate,
517.      String status,
518.      String name,
519.      String surname,
520.      String birthDate,
521.      String email,
522.      String telephone,
523.      String note
524.  )
525.  {

```

```

526.         init();
527.         Distributor d = new Distributor (regNum);
528.
529.         d.setSponsor(sponsor);
530.         d.setCountry(country);
531.         d.setRegistrationDate(registrationDate);
532.         d.setStatus(status);
533.         d.setName(name);
534.         d.setSurname(surname);
535.         d.setBirthDate(birthDate);
536.         d.setEmail(email);
537.         d.setTelephone(telephone);
538.         d.setNote(note);
539.
540.         return editDistributor (d);
541.     }
542.
543.     /**
544.      * Delete distributor by given position.
545.      *
546.      * @param position position in the file
547.      * @return true if successful
548.      */
549.     public static boolean deleteDistributor(long position)
550.     {
551.         init();
552.         return deleteDistributor(listRegNum.findByPosition(position).getKey());
553.     }
554.
555.
556.     /**

```

```

557.      * Delete distributor by given regNum.
558.      * and trigger to repopulate lists.
559.      *
560.      * @param regNum    registration number
561.      * @param repopulate boolean
562.      * @return      true if successful
563.      */
564.  public static boolean deleteDistributor(String regNum)
565.  {
566.      init();
567.
568.      //delete all children
569.      Item[] items = listSponsor.findMore(regNum);
570.      if (items != null)
571.      {
572.          String childrenRegNums[] = new String[items.length];
573.
574.          for (int i=0;i<items.length;i++)
575.              childrenRegNums[i] = ((ItemSponsor) items[i]).itemRegNum.getKey();
576.
577.          for (int i=0;i<childrenRegNums.length;i++) deleteDistributor (childrenRegNums[i]);
578.      }
579.      //delete current Distributor
580.      if (f.delete(listRegNum.findPosition(regNum)))
581.      {
582.          populateLists();
583.          return true;
584.      }
585.      else return false;
586.  }
587.

```

```

588.  /**
589.   * Delete distributor by given object.
590.   *
591.   * @param d    distributor object
592.   * @return     true if successful
593.   */
594.  public static boolean deleteDistributor(Distributor d)
595.  {
596.      init();
597.      return deleteDistributor(d.getRegNum());
598.  }
599.
600.  /**
601.   * Retrieves distributor object according to its sponsor.
602.   * Returns the first occurrence in list.
603.   *
604.   * @param sponsor string
605.   * @return     distributor object
606.   */
607.  public static Distributor getDistributorBySponsor(String sponsor)
608.  {
609.      init();
610.      ItemSponsor item = ((ItemSponsor)listSponsor.find(sponsor));
611.      if (item == null) return null;
612.      return getDistributorByRegNum(item.regNum);
613.  }
614.
615.  /**
616.   * From a given array of regNums this method returns array of distributors
617.   * of all distributors given in vector together with all their children.
618.   *

```

```

619.      * @param regNums  vector filled with regNums
620.      * @return      array of distributors
621.      */
622.      public static Distributor[] getDistributorsWithDirectChildrenByRegNums(Vector<String> regNums)
623.      {
624.          init();
625.      /** DOSSIER: HL mastery 10 - hierarchical composite data structure */
626.          Vector<Distributor> ret = new Vector<Distributor>(); //array for returning
627.          String[] regNumsArray = regNums.toStringArray();
628.          for (int i=0;i<regNumsArray.length;i++)
629.          {
630.              Distributor d = getDistributorByRegNum(regNumsArray[i]);
631.              ret.pushBack (d);
632.
633.              //all their children
634.              Distributor[] children = d.getDirectChildrenDistributors();
635.
636.              if (children != null)
637.              for (int b=0;b<children.length;b++)
638.              {
639.                  if (regNums.indexOf(children[b].getRegNum()) == -1)
640.                  {
641.                      regNums.pushBack(children[b].getRegNum());
642.
643.                      ret.pushBack(children[b]);
644.                  }
645.              }
646.          }
647.          return ret.toDistributorArray();
648.      }
649.  }

```

## File MonthManager.java

```
1.
2.  /**
3.   * Manipulates Month entities and their info in files.
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8. public class MonthManager
9. {
10.     private DirectAccessFile f; //file with records of months for particular distributor
11.     private Distributor d; //distributor to which this MonthManager belongs to
12.     private List list;
13.
14.     /**
15.      * Constructor - creates the pointer to file and initialize list
16.      *
17.      * @param regNum    registration number of distributor
18.      */
19.     public MonthManager(Distributor d)
20.     {
21.         //initialize pointer to DAF
22.         f = new DirectAccessFile (Static.fileMonthPrefix+d.getRegNum()+Static.fileMonthSuffix,
23.             Static.monthRecordLength);
24.         populateLists();
25.         this.d = d;
26.     }
27.
28.     /**
29.      * Populates lists with information from records.
```

```

30.      *
31.      */
32.  public void populateLists()
33.  {
34.      //initialize lists
35.      list = new List(true);
36.
37.      //populate lists
38.      Month m;
39.
40.      for (int i=0;i<=f.getNumberOfRecords()-1;i++)
41.      {
42.          m = new Month(d);
43.          m.readFromFile(f,i);
44.
45.          //populating
46.          addToLists(m, i);
47.      }
48.
49.  }
50.
51.  /**
52.   * Adds Month given to lists
53.   *
54.   * @param m      Month object
55.   * @param position position in file
56.   */
57.  public void addToLists(Month m, long position)
58.  {
59.      list.add(Static.removeDiacritics(m.getYear()+m.getMonth()).toLowerCase(), position);
60.  }

```

```
61.
62.  /**
63.   * Returns the number of Months in the file.
64.   *
65.   * @return      the number of Months in the file
66.   */
67. public int getCount()
68. {
69.     return (int) f.getNumberOfRecords();
70. }
71.
72. /**
73.   * Adds new Month to the database.
74.   * Creates new record in particular DAF and creates list items.
75.   *
76.   * @param year
77.   * @param month
78.   * @param pw
79.   * @param groupPw
80.   * @param gw
81.   * @param groupGw
82.   * @return      true if successful
83.   */
84. public boolean addMonth(
85.     String year,
86.     String month,
87.     String pw,
88.     String groupPw,
89.     String gw,
90.     String groupGw
91. )
```



```

92.     {
93.         Month m = new Month(d);
94.         //check if there is no such month in the system
95.         if (list.findPosition(year+month) == -1)
96.         {
97.             m.setYear(year);
98.             if (Integer.parseInt(month) < 10)
99.                 month = "0"+Integer.parseInt(month);
100.            m.setMonth(month);
101.            m.setPw(pw);
102.            m.setGroupPw(groupPw);
103.            m.setGw(gw);
104.            m.setGroupGw(gw);
105.
106.            m.appendToFile(f);
107.
108.            //add to lists
109.            addToLists(m,f.getNumberOfRecords()-1);
110.
111.            return true;
112.        } else
113.        {
114.            //there is a such month
115.            return false;
116.        }
117.    }
118.
119.    /**
120.     * Edits Month record in the database
121.     * according to given information.
122.     *

```

```

123.      * @param m      Month object with new information
124.      * @param position position in the file
125.      * @return      true if the editing was successful
126.      */
127.  public boolean editMonth(Month m, long position)
128.  {
129.      if (f.getNumberOfRecords() > position)
130.      {
131.          m.writeToFile(f,position);
132.          populateLists();
133.          return true;
134.      } else
135.      {
136.          return false;
137.      }
138.  }
139.
140.  /**
141.   * Edits Month record in the database
142.   * according to given information.
143.   *
144.   * @param m      Month object with new information
145.   * @return      true if the editing was successful
146.   */
147.  public boolean editMonth(Month m)
148.  {
149.      return editMonth (m,list.findPosition(m.getYear()+m.getMonth()));
150.  }
151.
152.  /**
153.   * Delete Month by given position.

```

```

154.      *
155.      * @param position position in the file
156.      * @return true if successful
157.      */
158. public boolean deleteMonth(long position)
159. {
160.     if (f.delete(position))
161.     {
162.         populateLists();
163.         return true;
164.     } else return false;
165. }
166.
167. /**
168.  * Delete Month by given object.
169.  *
170.  * @param m Month object
171.  * @return true if successful
172.  */
173. public boolean deleteMonth(Month m)
174. {
175.     return f.delete(list.findPosition(m.getYear()+m.getMonth()));
176. }
177.
178. /**
179.  * Retrieves Month object from given position.
180.  *
181.  * @param position record in file
182.  * @return Month object
183.  */
184. public Month getMonth(long position)

```

```

185.     {
186.         Month m = new Month(d);
187.         m.readFromFile(f,position);
188.         return m;
189.     }
190.
191.     /**
192.      * Retrieves Month object by given key.
193.      *
194.      * @param key    key of the month
195.      * @return      Month object
196.      */
197.     public Month getMonth(String key)
198.     {
199.         long position = list.findPosition(key);
200.         if (position != -1)
201.         {
202.             Month m = new Month(d);
203.             m.readFromFile(f,position);
204.             return m;
205.         } else
206.             return null;
207.     }
208.
209.
210.     /**
211.      * Returns all Months sorted by YYYYMM
212.      *
213.      * @return      the sum of x and y
214.      */
215.     public Month[] getAll()

```

```

216.     {
217.         Month[] months = new Month[getCount()];
218.
219.         Item act = list.getFirst();
220.         int i = 0;
221.         while (act != null)
222.         {
223.             months[i] = getMonth(act.getPos());
224.             act = act.getNext();
225.             i++;
226.         }
227.         return months;
228.     }
229.
230.     /**
231.      * Returns last N months in records. If n is greater
232.      * than number of actual Months available, the maximum
233.      * number of Months is returned.
234.      *
235.      * @param n    number of months
236.      * @return     array of Months
237.      */
238.     public Month[] getLastMonths(int n)
239.     {
240.         Month[] months = getAll();
241.         if (n > months.length) n = months.length;
242.         Month[] ret = new Month[n];
243.         for (int i=0;i<n;i++)
244.         {
245.             ret[i] = months[months.length-i-1];
246.         }

```

```
247.  
248.     return ret;  
249. }  
250.  
251. }  
252.
```

## File Password.java

```
1.  import java.io.UnsupportedEncodingException;
2.  import java.security.MessageDigest;
3.  import java.security.NoSuchAlgorithmException;
4.
5.  /**
6.   * Contains methods for application's password manipulation.
7.   *
8.   * @author Juraj Masar
9.   * @version 0.1
10.  */
11. public class Password
12. {
13.     /**
14.      * Sets password to the application.
15.      * It generates salted hash of the password and consequently saves it.
16.      *
17.      * @param pass    password to save
18.      */
19.     public static void setPassword(String pass)
20.     {
21.         TextFileWriter t = new TextFileWriter (Static.filePassword);
22.         try
23.         {
24.             /** DOSSIER: HL mastery 17 - inserting data into ordered sequential file */
25.             t.writeln(SHA1(pass+Static.hashSalt));
26.         } catch (NoSuchAlgorithmException ae)
27.         {
28.             Dialog.error ("Setting new password was unsuccessful. \nError:"+ae.getMessage());
29.         } catch (UnsupportedEncodingException ae)
```

```

30.     {
31.         Dialog.error ("Setting new password was unsuccessful. \nError:"+ae.getMessage());
32.     }
33.     t.close();
34. }
35.
36. /**
37.  * Checks if given password is equal to password set in application.
38.  * It computes salted SHA1 hash which is consequently compared to hash
39.  * stored by the application.
40.  *
41.  * @param pass      password to check
42.  * @return          true if the password is valid
43.  */
44. public static boolean checkPassword(String pass)
45. {
46.     String hash = new String();
47.     try
48.     {
49.         hash = SHA1(pass+Static.hashSalt);
50.     } catch (NoSuchAlgorithmException ae)
51.     {
52.         Dialog.error ("Password checking was unsuccessful. \nError:"+ae.getMessage());
53.     } catch (UnsupportedEncodingException ae)
54.     {
55.         Dialog.error ("Password checking was unsuccessful. \nError:"+ae.getMessage());
56.     }
57.     TextFileReader t = new TextFileReader (Static.filePassword);
58.     /** DOSSIER: HL mastery 9 - parsing text file */
59.     return t.readLine().equals(hash);
60. }

```



```

61.
62.  /**
63.   * Converts text into hex-string.
64.   *
65.   * Originally from
66.   * http://www.anyexample.com/programming/java/java\_simple\_class\_to\_compute\_sha\_1\_hash.xml
67.   * (accessed 23. February 2010).
68.   *
69.   * @param data    array of bytes
70.   * @return        hex-string
71.   */
72. private static String convertToHex(byte[] data)
73. {
74.     StringBuffer buf = new StringBuffer();
75.     for (int i = 0; i < data.length; i++) {
76.         int halfbyte = (data[i] >>> 4) & 0x0F;
77.         int two_halfs = 0;
78.         do {
79.             if ((0 <= halfbyte) && (halfbyte <= 9))
80.                 buf.append((char) ('0' + halfbyte));
81.             else
82.                 buf.append((char) ('a' + (halfbyte - 10)));
83.             halfbyte = data[i] & 0x0F;
84.         } while(two_halfs++ < 1);
85.     }
86.     return buf.toString();
87. }
88.
89. /**
90.  * Computes SHA1 hash from a given string.
91.  *

```

```

92.      * Originally from
93.      * http://www.anyexample.com/programming/java/java\_simple\_class\_to\_compute\_sha\_1\_hash.xml
94.      * (accessed 23. February 2010).
95.      *
96.      * @param text original string to compute hash from
97.      * @return      SHA1 hash
98.      */
99.  private static String SHA1(String text)
100.      throws NoSuchAlgorithmException, UnsupportedEncodingException
101.  {
102.      MessageDigest md;
103.      md = MessageDigest.getInstance("SHA-1");
104.      byte[] shalhash = new byte[40];
105.      md.update(text.getBytes("utf-8"), 0, text.length());
106.      shalhash = md.digest();
107.      return convertToHex(shalhash);
108.  }
109. }
110.

```

## File Record.java

```
1.
2.  /**
3.   * Abstract class Record - predefines the behaviour of an entity which is to be saved in DAF
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8.  public abstract class Record implements Cloneable
9.  {
10.     protected String[] data;
11.
12.     /**
13.      * returns key of the record - what sorting of records is based on
14.      *
15.      * @return      array of strings
16.      */
17.     public abstract String getKey();
18.
19.     /**
20.      * Returns all data in an array
21.      *
22.      * @return      data array
23.      */
24.     public String[] toArray()
25.     {
26.         return data;
27.     }
28.
29.     /**
```

```

30.      * Returns all data in a string
31.      *
32.      * @return      data string
33.      */
34.  public String toString()
35.  {
36.      return Static.implode(data);
37.  }
38.
39.  /**
40.      * Returns a clone of current object
41.      *
42.      * @return      clone of current Object
43.      */
44.
45.  public Object clone()
46.  {
47.      try
48.      {
49.          return super.clone();
50.      }
51.      catch (CloneNotSupportedException e) {
52.          return null;
53.      }
54.  }
55.
56.  /**
57.      * writes the content of the record to the given DirectAccessFile
58.      *
59.      * @param  f      file to which the record should be written
60.      */

```

```
61. public void writeToFile(DirectAccessFile f)
62. {
63.     f.write(Static.encrypt(Static.implode(data)));
64. }
65.
66. /**
67.  * writes the content of the record to the given DirectAccessFile
68.  * at given position
69.  *
70.  * @param f    file to which the record should be written
71.  * @param pos  position in the file
72.  */
73. public void writeToFile(DirectAccessFile f, long pos)
74. {
75.     f.write(Static.encrypt(Static.implode(data)), pos);
76. }
77.
78. /**
79.  * reads and populates the record with info from file
80.  *
81.  * @param f    DirectAccessFile to read from
82.  */
83. public void readFromFile (DirectAccessFile f)
84. {
85.     data = Static.explode(Static.decrypt(f.read()));
86. }
87.
88. /**
89.  * reads and populates the record with info from file
90.  * from given position
91.  *
```

```

92.      * @param f    DirectAccessFile to read from
93.      * @param pos  position in the file to read from
94.      */
95.      public void readFromFile (DirectAccessFile f, long pos)
96.      {
97.      /** DOSSIER: HL mastery 9 - parsing data stream */
98.          data = Static.explode(Static.decrypt(f.read(pos)));
99.      }
100.
101.      /**
102.       * appends the content of the record to the end of given file
103.       *
104.       * @param f    DirectAccessFile to which record should be appended
105.       */
106.      public void appendToFile(DirectAccessFile f)
107.      {
108.          f.append(Static.encrypt(Static.implode(data)));
109.      }
110.
111.      /**
112.       * reads and populates the record with info from given string
113.       *
114.       * @param s    string with information
115.       */
116.      public void readFromString (String s)
117.      {
118.          data = Static.explode(Static.decrypt(s));
119.      }
120.
121.      /**
122.       * returns the content of the record

```

```
123.      *
124.      * @return imploded string with all information
125.      */
126.  public String writeTostring()
127.  {
128.      return Static.encrypt(Static.implode(data));
129.  }
130.
131.
132. }
133.
```

## File Month.java

```
1.  import java.util.Date;
2.
3.  /**
4.   * defines the way the entity month (of a distributor) is stored in file
5.   *
6.   * @author Juraj Masar
7.   * @version 0.1
8.   */
9.  public class Month extends Record
10. {
11.     Distributor d; //distributor to which this Month belongs to
12.     /**
13.      * constructor for objects of class Month
14.      * initializes the data variable
15.      */
16.     public Month(Distributor d)
17.     {
18.         data = new String[6]; //allocates memory for all required information about Month
19.         //the order of information in data array is as:
20.         //data[0] - year      - 4  - getYear / setYear
21.         //data[1] - month     - 2  - getMonth / setMonth
22.         //data[2] - pw        - 10 - getPw / setPw
23.         //data[3] - grouppw   - 10 - getGroupPw / setGroupPw
24.         //data[4] - gw        - 10 - getGw / setGw
25.         //data[5] - groupgw   - 10 - getGroupGw / setGroupGw
26.
27.         //initialize values of all properties to empty string
28.         for (int i=0;i<=data.length-1;i++) data[i] = new String();
29.
```



```

30.         //set pointer do Distributor
31.         this.d = d;
32.     }
33.
34.     /**
35.      * Checks whether Month has been filled with data or not
36.      *
37.      * @return      true if empty
38.      */
39.     public boolean isEmpty()
40.     {
41.         return !(getPwInt() != 0 || getGroupPwInt() != 0 || getGwInt() != 0 || getGroupGwInt() != 0);
42.     }
43.
44.     /**
45.      * returns the key according to which it is logical to sort distributors
46.      *
47.      * @return      combination of year and month
48.      */
49.     public String getKey()
50.     {
51.         return getYear()+getMonth();
52.     }
53.
54.     /**
55.      * returns year of the Month object
56.      *
57.      * @return      year
58.      */
59.     public String getYear()
60.     {

```

```

61.         return data[0];
62.     }
63.
64.     /**
65.      * sets year of the Month object to the string given
66.      *
67.      * @param s    year
68.      */
69.     public void setYear (String s)
70.     {
71.         data[0] = Static.cutIfLongerThan(s,4);
72.     }
73.
74.     /**
75.      * returns month of the Month object
76.      *
77.      * @return     month
78.      */
79.     public String getMonth()
80.     {
81.         return data[1];
82.     }
83.
84.     /**
85.      * sets month of the Month object to the string given
86.      *
87.      * @param s    month
88.      */
89.     public void setMonth (String s)
90.     {
91.         data[1] = Static.cutIfLongerThan(s,2);

```

```

92.     }
93.
94.     /**
95.      * returns Pw of the Month object
96.      *
97.      * @return      Pw
98.      */
99.     public String getPw()
100.    {
101.        return data[2];
102.    }
103.
104.    /**
105.     * returns Pw of the Month object
106.     *
107.     * @return      Pw
108.     */
109.     public int getPwInt()
110.    {
111.        try
112.        {
113.            return Integer.parseInt(data[2]);
114.        } catch (java.lang.NumberFormatException e)
115.        {
116.            return 0;
117.        }
118.    }
119.
120.    /**
121.     * sets Pw of the Month object to the string given
122.     *

```

```

123.     * @param s    Pw
124.     */
125. public void setPw (String s)
126. {
127.     data[2] = Static.cutIfLongerThan(s,10);
128. }
129.
130. /**
131.  * returns GroupPw of the Month object
132.  *
133.  * @return      GroupPw
134.  */
135. public String getGroupPw()
136. {
137.     return data[3];
138. }
139.
140. /**
141.  * returns GroupPw of the Month object
142.  *
143.  * @return      GroupPw
144.  */
145. public int getGroupPwInt()
146. {
147.     try
148.     {
149.         return Integer.parseInt(data[3]);
150.     } catch (java.lang.NumberFormatException e)
151.     {
152.         return 0;
153.     }

```

```

154.     }
155.
156.     /**
157.      * sets GroupPw of the Month object to the string given
158.      *
159.      * @param s      GroupPw
160.      */
161.     public void setGroupPw (String s)
162.     {
163.         data[3] = Static.cutIfLongerThan(s,10);
164.     }
165.
166.     /**
167.      * returns Gw of the Month object
168.      *
169.      * @return      Gw
170.      */
171.     public String getGw()
172.     {
173.         return data[4];
174.     }
175.
176.     /**
177.      * returns Gw of the Month object
178.      *
179.      * @return      Gw
180.      */
181.     public int getGwInt()
182.     {
183.         try
184.         {

```

```

185.         return Integer.parseInt(data[4]);
186.     } catch (java.lang.NumberFormatException e)
187.     {
188.         return 0;
189.     }
190. }
191.
192. /**
193.  * sets Gw of the Month object to the string given
194.  *
195.  * @param s    Gw
196.  */
197. public void setGw (String s)
198. {
199.     data[4] = Static.cutIfLongerThan(s,10);
200. }
201.
202. /**
203.  * returns GroupGw of the Month object
204.  *
205.  * @return     GroupGw
206.  */
207. public String getGroupGw()
208. {
209.     return data[5];
210. }
211.
212. /**
213.  * returns GroupGw of the Month object
214.  *
215.  * @return     GroupGw

```

```

216.     */
217. public int getGroupGwInt()
218. {
219.     try
220.     {
221.         return Integer.parseInt(data[5]);
222.     } catch (java.lang.NumberFormatException e)
223.     {
224.         return 0;
225.     }
226. }
227.
228. /**
229.  * sets GroupGw of the Month object to the string given
230.  *
231.  * @param s    GroupGw
232.  */
233. public void setGroupGw (String s)
234. {
235.     data[5] = Static.cutIfLongerThan(s,10);
236. }
237.
238. /**
239.  * Saves the record to the database.
240.  *
241.  * @return     true if successful
242.  */
243. public boolean save()
244. {
245.     return d.months.editMonth(this);
246. }

```

```

247.
248.  /**
249.   * Deletes the record of particular Month in database.
250.   *
251.   * @return      true if successful
252.   */
253. public boolean delete()
254. {
255.     return d.months.deleteMonth(this);
256. }
257.
258. /**
259.   * Calculates total pw
260.   *
261.   * @return      the sum of pw and groupPw
262.   */
263. public int getTotalPw()
264. {
265.     return getPwInt()+getGroupPwInt();
266. }
267.
268.
269. /**
270.   * Calculates total Gw
271.   *
272.   * @return      the sum of pw and groupPw
273.   */
274. public int getTotalGw()
275. {
276.     return getGwInt()+getGroupGwInt();
277. }

```



```

278.
279.
280.  /**
281.   * Determines the level for current Month
282.   *
283.   * @return      level in MLM system
284.   */
285. public int getLevel()
286. {
287.     if (getTotalPw() < 2) return 0;
288.     else if (getTotalPw() >= 2 && getTotalPw() < 500) return 3;
289.     else if (getTotalPw() >= 500 && getTotalPw() < 1000) return 6;
290.     else if (getTotalPw() >= 1000 && getTotalPw() < 2000) return 9;
291.     else if (getTotalPw() >= 2000 && getTotalPw() < 4000) return 12;
292.     else if (getTotalPw() >= 4000 && getTotalPw() < 8000) return 15;
293.     else if (getTotalPw() >= 8000 && getTotalPw() < 12000) return 18;
294.     else return 21;
295. }
296.
297. /**
298.   * Determines the bonus level according to given number
299.   * of distributors with level 21;
300.   *
301.   * @param count21 distributors with level 21
302.   * @return      bonus level
303.   */
304. public static double getBonusLevel(int count21)
305. {
306.     if (count21 < 1) return 0;
307.     if (count21 == 1) return 7;
308.     if (count21 < 4) return 7.5;

```

```

309.         if (count21 < 6) return 8;
310.         if (count21 < 8) return 8.5;
311.         if (count21 < 10) return 9;
312.         if (count21 < 12) return 9.5;
313.         else return 10;
314.     }
315.
316.     /**
317.      * Calculates provision for particular month
318.      *
319.      * @return    provision
320.      */
321.     public int getProvision()
322.     {
323.         double provision = 0;
324.
325.         //init provision from differences + provision from 21 levels
326.         int count21 = 0;
327.         double totalGw21 = 0;
328.
329.         Distributor[] children = d.getDirectChildrenDistributors();
330.         for (int i=0;i<children.length;i++)
331.         {
332.             Month m = children[0].months.getMonth(getKey());
333.             if (m != null && !m.isEmpty())
334.             {
335.                 if (m.getLevel() == 21)
336.                 {
337.                     count21++;
338.                     totalGw21 += m.getTotalGw();
339.                 } else

```

```

340.         provision += (double)(getLevel()-m.getLevel())*m.getTotalGw();
341.     }
342. }
343.     int prov = (int) (provision + (double)(getBonusLevel(count21)*totalGw21)
344.         +(double)getLevel()*getGwInt());
345.
346.     if (prov < 0) return 0; //check for incorrect data
347.     else return prov;
348. }
349.
350. /**
351.  * Calculates the number of children distributors of particular distributor
352.  * who were registered until the end of this month.
353.  *
354.  * @return number of direct children
355.  */
356. public int getNumberOfDirectChildren ()
357. {
358.     if (isEmpty()) return 0;
359.
360.     //get border date
361.     int bmonth = Integer.parseInt(getMonth());
362.     int byear = Integer.parseInt(getYear());
363.
364.     //increase month
365.     if (bmonth == 12)
366.     {
367.         bmonth = 1;
368.         byear ++;
369.     } else
370.     {

```

```

371.         bmonth ++;
372.     }
373.
374.     return d.getTree().getNumberOfDirectChildrenBefore("1."+bmonth+"."+byear);
375. }
376.
377. /**
378.  * Calculates the number of children distributors of particular distributor
379.  * who were registered until the end of this month.
380.  *
381.  * @return number of direct children
382.  */
383. public int getNumberOfAllChildren ()
384. {
385.     if (isEmpty()) return 0;
386.
387.     //get border date
388.     int bmonth = Integer.parseInt(getMonth());
389.     int byear = Integer.parseInt(getYear());
390.
391.     //increase month
392.     if (bmonth == 12)
393.     {
394.         bmonth = 1;
395.         byear ++;
396.     } else
397.     {
398.         bmonth ++;
399.     }
400.
401.     return d.getTree().getNumberOfAllChildrenBefore("1."+bmonth+"."+byear);

```

402.        }  
403.    }  
404.

## File Distributor.java

```
1.
2.  /**
3.   * defines the way the entity distributor is stored in file
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8.  /** DOSSIER: SL mastery 3 - objects as data records */
9.  /** DOSSIER: HL mastery 7 - inheritance */
10. public class Distributor extends Record
11. {
12.     public MonthManager months;
13.     private TreeItem tree;
14.     /**
15.      * constructor for objects of class Distributor
16.      * initializes the data variable
17.      */
18.     public Distributor()
19.     {
20.         //initialization of object variables
21.         months = null;
22.         tree = null;
23.
24.         data = new String[11]; //allocates memory for all required information about Distributor
25.         //the order of information in data array is as (with max chars possible):
26.         //data[0] - registration number      - 10 - getRegNum / setRegNum
27.         //data[1] - sponsor number           - 10 - getSponsor / setSponsor
28.         //data[2] - country                  - 3  - getCountry / setCountry
29.         //data[3] - date of registration     - 15 - getRegistrationDate / setRegistrationDate
```

```

30.         //data[4] - status                - 5    - getStatus / setStatus
31.         //data[5] - name                  - 20    - getName / setName
32.         //data[6] - surname               - 20    - getSurname / setSurname
33.         //data[7] - date of birth         - 15    - getBirthDate / setBirthDate
34.         //data[8] - email                 - 50    - getEmail / setEmail
35.         //data[9] - telephone            - 20    - getTelephone / setTelephone
36.         //data[10] - note                 - 300   - getNote / setNote
37.
38.         //initialize values of all properties to empty string
39.         for (int i=0;i<=data.length-1;i++) data[i] = new String();
40.
41.     }
42.
43.     /**
44.      * constructor for objects of class Distributor
45.      * initializes the data variable and sets regNum
46.      */
47.     public Distributor(String regNum)
48.     {
49.         //initialization of object variables
50.         months = null;
51.         tree = null;
52.
53.         data = new String[11]; //allocates memory for all required information about Distributor
54.         //initialize values of all properties to empty string
55.         for (int i=0;i<=data.length-1;i++) data[i] = new String();
56.         if (!setRegNum(regNum)) Dialog.error ("Error while creating new distributor:\n"+
57.             "the registration number has to be numeric.");
58.     }
59.
60.     /**

```

```

61.      * returns the key according to which it is logical to sort distributors
62.      *
63.      * @return      registration number of a distributor
64.      */
65.  public String getKey()
66.  {
67.      return getRegNum();
68.  }
69.
70.  /**
71.      * returns registration number of a distributor
72.      *
73.      * @return      registration number
74.      */
75.  public String getRegNum()
76.  {
77.      return data[0];
78.  }
79.
80.  /**
81.      * Checks if distributor is the root distributor in the system.
82.      *
83.      * @return      true if it is
84.      */
85.  public boolean isRoot()
86.  {
87.      if (getSponsor().equals("0")) return true;
88.      else return false;
89.  }
90.
91.  /**

```



```

92.      * Returns the generation of distributors to which
93.      * this distributor belongs to.
94.      *
95.      * @return      generation (0 is root)
96.      */
97.  public int getGeneration()
98.  {
99.      if (checkTree())
100.      {
101.          TreeItem t = tree;
102.          int generation = 0;
103.          while (t.parent != null)
104.          {
105.              t = t.parent;
106.              generation++;
107.          }
108.          return generation;
109.      } else
110.      {
111.          return -1;
112.      }
113.  }
114.
115.
116.  /**
117.   * sets registratration number of a distributor to the string given
118.   *
119.   * @param s      registration number
120.   * @return true if successful
121.   */
122.  private boolean setRegNum (String s)

```

```

123.     {
124.         if (Static.isNumeric(s))
125.         {
126.             data[0] = Static.cutIfLongerThan(s,10);
127.
128.             //initialize MonthManager for this Distributor
129.             months = new MonthManager(this);
130.
131.             return true;
132.         } else
133.             return false;
134.     }
135.
136.     /**
137.      * returns sponsor number of a distributor
138.      *
139.      * @return      sponsor number
140.      */
141.     public String getSponsor()
142.     {
143.         return data[1];
144.     }
145.
146.     /**
147.      * sets sponsor of a distributor to the string given
148.      *
149.      * @param s      sponsor number
150.      */
151.     public boolean setSponsor (String s)
152.     {
153.         if (Static.isNumeric(s))

```

```

154.         {
155.             data[1] = Static.cutIfLongerThan(s,10);
156.             return true;
157.         } else
158.             return false;
159.     }
160.
161.     /**
162.      * returns country of a distributor
163.      *
164.      * @return      country
165.      */
166.     public String getCountry()
167.     {
168.         return data[2];
169.     }
170.
171.     /**
172.      * sets country of a distributor to the string given
173.      *
174.      * @param s      country
175.      */
176.     public void setCountry (String s)
177.     {
178.         data[2] = Static.cutIfLongerThan(s,3);
179.     }
180.
181.     /**
182.      * returns date of registration of a distributor
183.      *
184.      * @return      date of registration

```

```

185.     */
186. public String getRegistrationDate()
187. {
188.     return data[3];
189. }
190.
191. /**
192.  * sets registration date of a distributor to the string given
193.  *
194.  * @param s    registration date
195.  */
196. public void setRegistrationDate (String s)
197. {
198.     data[3] = Static.cutIfLongerThan(s,15);
199. }
200.
201. /**
202.  * returns status of a distributor
203.  *
204.  * @return     status
205.  */
206. public String getStatus()
207. {
208.     return data[4];
209. }
210.
211. /**
212.  * sets status of a distributor to the string given
213.  *
214.  * @param s    status
215.  */

```

```

216.     public void setStatus (String s)
217.     {
218.         data[4] = Static.cutIfLongerThan(s,5);
219.     }
220.
221.     /**
222.      * returns name of a distributor
223.      *
224.      * @return      name
225.      */
226.     public String getName()
227.     {
228.         return data[5];
229.     }
230.
231.     /**
232.      * sets name of a distributor to the string given
233.      *
234.      * @param s      name
235.      */
236.     public void setName (String s)
237.     {
238.         data[5] = Static.cutIfLongerThan(s,20);
239.     }
240.
241.     /**
242.      * returns surname of a distributor
243.      *
244.      * @return      surname
245.      */
246.     public String getSurname()

```

```

247.     {
248.         return data[6];
249.     }
250.
251.     /**
252.      * Returns the full name of the distributor
253.      * a combination of their first name and surname.
254.      *
255.      * @return      fullname of distributor
256.      */
257.     public String getFullname()
258.     {
259.         return getName()+" "+getSurname();
260.     }
261.
262.     /**
263.      * sets surname of a distributor to the string given
264.      *
265.      * @param s      surname
266.      */
267.     public void setSurname (String s)
268.     {
269.         data[6] = Static.cutIfLongerThan(s,20);
270.     }
271.
272.     /**
273.      * returns date of birth of a distributor
274.      *
275.      * @return      date of birth
276.      */
277.     public String getBirthDate()

```

```

278.     {
279.         return data[7];
280.     }
281.
282.     /**
283.      * sets birth date of a distributor to the string given
284.      *
285.      * @param s    birth date
286.      */
287.     public void setBirthDate (String s)
288.     {
289.         data[7] = Static.cutIfLongerThan(s,15);
290.     }
291.
292.     /**
293.      * returns email of a distributor
294.      *
295.      * @return    email
296.      */
297.     public String getEmail()
298.     {
299.         return data[8];
300.     }
301.
302.     /**
303.      * sets email of a distributor to the string given
304.      *
305.      * @param s    email
306.      */
307.     public void setEmail (String s)
308.     {

```

```

309.         data[8] = Static.cutIfLongerThan(s,50);
310.     }
311.
312.     /**
313.      * returns telephone of a distributor
314.      *
315.      * @return     telephone
316.      */
317.     public String getTelephone()
318.     {
319.         return data[9];
320.     }
321.
322.     /**
323.      * sets telephone of a distributor to the string given
324.      *
325.      * @param s     telephone
326.      */
327.     public void setTelephone (String s)
328.     {
329.         data[9] = Static.cutIfLongerThan(s,20);
330.     }
331.
332.     /**
333.      * returns note of a distributor
334.      *
335.      * @return     note
336.      */
337.     public String getNote()
338.     {
339.         return data[10];

```



```

340.     }
341.
342.     /**
343.      * sets note of a distributor to the string given
344.      *
345.      * @param s    note
346.      */
347.     public void setNote (String s)
348.     {
349.         data[10] = Static.cutIfLongerThan(s,300);
350.     }
351.
352.     /**
353.      * Saves the record to the database.
354.      *
355.      * @return      true if successful
356.      */
357.     public boolean save()
358.     {
359.         return DistributorManager.editDistributor(this);
360.     }
361.
362.     /**
363.      * Deletes the record of particular distributor in database.
364.      *
365.      * @return      true if successful
366.      */
367.     public boolean delete()
368.     {
369.         return DistributorManager.deleteDistributor(this);
370.     }

```

```

371.
372.  /**
373.   * reads and populates the record with info from given string
374.   *
375.   * @param s    string with information
376.   */
377. public void readFromString (String s)
378. {
379.     super.readFromString(s);
380.
381.     //initialize MonthManager for this Distributor
382.     months = new MonthManager(this);
383. }
384.
385. /**
386.   * reads and populates the record with info from file
387.   *
388.   * @param f    DirectAccessFile to read from
389.   */
390. /** DOSSIER: HL mastery 6 - polymorphism */
391. public void readFromFile (DirectAccessFile f)
392. {
393.     super.readFromFile (f);
394.
395.     //initialize MonthManager for this Distributor
396.     months = new MonthManager(this);
397. }
398.
399. /**
400.   * reads and populates the record with info from file
401.   * from given position

```

```

402.      *
403.      * @param f    DirectAccessFile to read from
404.      * @param pos  position in the file to read from
405.      */
406.  public void readFromFile (DirectAccessFile f, long pos)
407.  {
408.      super.readFromFile (f, pos);
409.
410.      //initialize MonthManager for this Distributor
411.      months = new MonthManager(this);
412.
413.  }
414.
415.  /**
416.   * Checks whether the pointer to TreeItem is ready.
417.   * If not it makes it ready.
418.   *
419.   * @return true if successful
420.   */
421.  public boolean checkTree ()
422.  {
423.      tree = ((ItemRegNum)DistributorManager.listRegNum.find(getRegNum())).treeItem;
424.      if (tree != null) return true;
425.      else return false;
426.  }
427.
428.  /**
429.   * Checks if the tree pointer is correct and returns it
430.   *
431.   * @return    pointer to TreeItem
432.   */

```

```

433. public TreeItem getTree()
434. {
435.     if (checkTree()) return tree;
436.     else return null;
437. }
438.
439. /**
440.  * Returns the number of direct children of this distributor
441.  *
442.  * @return      number of direct children
443.  */
444. public int getNumberOfDirectChildren()
445. {
446.     if (checkTree())
447.         return tree.getNumberOfDirectChildren();
448.     else return -1;
449. }
450.
451. /**
452.  * Returns the number of all children of this distributor
453.  *
454.  * @return      number of all children
455.  */
456. public int getNumberOfAllChildren()
457. {
458.     if (checkTree())
459.         return tree.getNumberOfAllChildren();
460.     else return -1;
461. }
462.
463. /**

```

```

464.      * Retrieves direct children Distributors
465.      *
466.      * @return      array of Distributor objects
467.      */
468.  public Distributor[] getDirectChildrenDistributors()
469.  {
470.      if (checkTree())
471.      {
472.          Vector<Long> positions = new Vector<Long>();
473.          if (tree.children != null)
474.              for (int i=0;i<tree.children.length;i++) positions.pushBack (tree.children[i].position);
475.          return DistributorManager.getDistributorsFromPositions(positions);
476.      } else return null;
477.  }
478.
479.  /**
480.      * Retrieves all children Distributors
481.      *
482.      * @return      array of Distributor objects
483.      */
484.  public Distributor[] getAllChildrenDistributors()
485.  {
486.      if (checkTree())
487.      {
488.          Vector<Long> positions = new Vector<Long>();
489.          tree.getPositions(positions, false);
490.          return DistributorManager.getDistributorsFromPositions(positions);
491.      } else return null;
492.  }
493.  }

```

## File Vector.java

```
1.
2.  /**
3.   * Vector class - data structure with dynamic allocation of memory.
4.   * 'Enhanced array'
5.   *
6.   * @author Juraj Masar
7.   * @version 0.1
8.   */
9.  /** DOSSIER: HL mastery 12-15 - ADT */
10. public class Vector<O>
11. {
12.     private VectorItem<O> first; //pointer to first VectorItem in the vector
13.     private VectorItem<O> last; //pointer to last VectorItem in the vector
14.     private int count; //number of elements in Vector
15.     /**
16.      * Constructor for objects of class Vector
17.      */
18.     public Vector()
19.     {
20.         // initialise instance variables
21.         first = null;
22.         last = null;
23.         count = 0;
24.     }
25.
26.     /**
27.      * Constructor for objects of class Vector
28.      * which populates the Vector with data given.
29.      * @param c number of elements to insert
```

```

30.      * @param o default element
31.      */
32.  public Vector(int c, O o)
33.  {
34.      // initialise instance variables
35.      first = null;
36.      last = null;
37.      count = 0;
38.      for (int i=0;i<=c-1;i++) pushBack(o);
39.  }
40.
41.
42.  /**
43.   * Returns the number of elements in the Vector
44.   *
45.   * @return      count of the elements in Vector
46.   */
47.  public int size()
48.  {
49.      return count;
50.  }
51.
52.  /**
53.   * Checks whether the vector is empty.
54.   *
55.   * @return      true if the vector is empty
56.   */
57.  public boolean isEmpty()
58.  {
59.      if (count > 0) return false;
60.      else return true;

```

```
61.     }
62.
63.
64.     /**
65.      * Removes all elements from the Vector
66.      *
67.      */
68.     public void clear()
69.     {
70.         first = null;
71.         last = null;
72.         count = 0;
73.     }
74.
75.     /**
76.      * Returns a clone of this vector.
77.      *
78.      * @return      vector
79.      */
80.     public Vector clone()
81.     {
82.         Vector<O> v = new Vector<O>();
83.
84.         VectorItem<O> vi = first;
85.         while (vi != null)
86.         {
87.             v.pushBack (vi.getObject());
88.             vi = vi.getNext();
89.         }
90.
91.         return v;
```



```
92.     }
93.
94.     /**
95.      * Returns the first element
96.      *
97.      * @return      object
98.      */
99.     public O first()
100.    {
101.        if (isEmpty()) return null;
102.        return first.getObject();
103.    }
104.
105.    /**
106.     * Returns the first element
107.     *
108.     * @return      VectorItem
109.     */
110.    public VectorItem<O> firstVectorItem()
111.    {
112.        if (isEmpty()) return null;
113.        return first;
114.    }
115.
116.
117.    /**
118.     * Returns the last element
119.     *
120.     * @return      object
121.     */
122.    public O last()
```

```

123.     {
124.         if (isEmpty()) return null;
125.         return last.getObject();
126.     }
127.
128.     /**
129.      * Adds element to the end of Vector
130.      *
131.      * @param o    object to be inserted
132.      */
133.     public void pushBack (O o)
134.     {
135.         //create new VectorItem
136.         VectorItem<O> vi = new VectorItem<O> (o,last, null);
137.         if (isEmpty())
138.         {
139.             last = vi;
140.             first = vi;
141.         }
142.         else
143.         {
144.             last.setNext(vi);
145.             last = vi;
146.         }
147.         count++;
148.     }
149.
150.     /**
151.      * Adds element to the beginning of the vector
152.      *
153.      * @param o    object to add

```

```

154.     */
155. public void pushFront(O o)
156. {
157.     //create new VectorItem
158.     VectorItem<O> vi = new VectorItem<O> (o,null, first);
159.
160.     first = vi;
161.     if (isEmpty()) last = vi;
162.     count++;
163. }
164.
165. /**
166.  * Removes and returns the last element of the vector.
167.  *
168.  * @return    object
169.  */
170. public O popBack()
171. {
172.     if (!isEmpty())
173.     {
174.         VectorItem<O> l = last;
175.         last = last.getPrevious();
176.
177.         count--;
178.         if (isEmpty()) first = null;
179.         else last.setNext(null);
180.         return l.getObject();
181.     } else
182.         return null;
183.
184. }

```

```

185.
186.  /**
187.   * Removes and returns the first element of the vector.
188.   *
189.   * @return    object
190.   */
191. public O popFront()
192. {
193.     if (!isEmpty())
194.     {
195.         VectorItem<O> l = first;
196.         first = first.getNext();
197.
198.         count--;
199.         if (isEmpty()) last = null;
200.         else first.setPrevious(null);
201.
202.         return l.getObject();
203.     } else
204.         return null;
205. }
206.
207. /**
208.   * Returns object at index given.
209.   *
210.   * @param index    index of the element
211.   * @return    object
212.   */
213. public O get(int index)
214. {
215.     if (index >= count || index < 0) return null;

```

```
216.
217.     if (index > count/2)
218.     {
219.         //search from the end
220.         VectorItem<O> vi = last;
221.         for (int i=count-1;i>index;i--)
222.         {
223.             vi = vi.getPrevious();
224.         }
225.         return vi.getObject();
226.
227.     } else
228.     {
229.         //search from the beginning
230.         VectorItem<O> vi = first;
231.
232.         for (int i=0;i<index;i++)
233.         {
234.             vi = vi.getNext();
235.         }
236.         return vi.getObject();
237.     }
238. }
239.
240. /**
241.  * Returns the VectorItem at index given.
242.  *
243.  * @param index    index of the element
244.  * @return         Object
245.  */
246. public VectorItem<O> getVectorItem(int index)
```

```

247.     {
248.         if (index >= count || index < 0) return null;
249.
250.         if (index > count/2)
251.         {
252.             //search from the end
253.             VectorItem<O> vi = last;
254.             for (int i=count-1;i>index;i--)
255.             {
256.                 vi = vi.getPrevious();
257.             }
258.             return vi;
259.
260.         } else
261.         {
262.             //search from the beginning
263.             VectorItem<O> vi = first;
264.
265.             for (int i=0;i<index;i++)
266.             {
267.                 vi = vi.getNext();
268.             }
269.             return vi;
270.         }
271.     }
272.
273. /**
274.  * Sets object to the vector at position given.
275.  *
276.  * @param o    object
277.  * @param index position

```

```

278.      * @return true if operation was successful
279.      */
280.  public boolean set(O o, int index)
281.  {
282.      if (index >= count || index < 0) return false;
283.
284.      VectorItem<O> vi = first;
285.
286.      for (int i=0;i<=index-1;i++)
287.      {
288.          vi = vi.getNext();
289.      }
290.
291.      vi.setObject(o);
292.      return true;
293.
294.  }
295.
296.  /**
297.   * Returns the index of first occurrence of given object.
298.   *
299.   * @param o    object to search for
300.   * @return     index of the found object or -1
301.   */
302.  public int indexOf(O o)
303.  {
304.      VectorItem<O> vi = first;
305.
306.      for (int i=0;i<=count-1;i++)
307.      {
308.          if (vi.getObject().equals(o)) return i;

```

```

309.         vi = vi.getNext();
310.     }
311.     return -1;
312. }
313.
314. /**
315.  * Inserts object given to the position given in Vector.
316.  *
317.  * @param index    index where to insert
318.  * @param O        object to insert
319.  * @return         true if successful
320.  */
321. public boolean insertAt(int index, O o)
322. {
323.     if (index < 0 || index > size()) return false; //out of range
324.     if (index == size()) //last
325.     {
326.         pushBack (o);
327.         return true;
328.     } else if (index == 0)
329.     {
330.         pushFront (o);
331.         return true;
332.     } else
333.     {
334.         VectorItem<O> before, after;
335.         before = getVectorItem(index-1);
336.         after = getVectorItem(index);
337.
338.         VectorItem<O> vi = new VectorItem<O> (o,before,after);
339.         before.setNext(vi);

```



```

340.         after.setPrevious(vi);
341.         return true;
342.     }
343. }
344.
345. /**
346.  * Inserts the object given after the other given object.
347.  *
348.  * @param where    object after which new object should be inserted
349.  * @param what      object to insert
350.  * @return         true if successful
351.  */
352. public boolean insertAfter(O where, O what)
353. {
354.     return insertAt(indexOf(where)+1,what);
355. }
356.
357. /**
358.  * Inserts the object given before the other given object.
359.  *
360.  * @param where    object before which new object should be inserted
361.  * @param what      object to insert
362.  * @return         true if successful
363.  */
364. public boolean insertBefore(O where, O what)
365. {
366.     return insertAt(indexOf(where)-1,what);
367. }
368.
369. /**
370.  * Returns object which is situated in the Vector just

```

```

371.      * before the object given or null
372.      *
373.      * @param O    object to find
374.      * @return     object before the object given
375.      */
376.  public O before(O a)
377.  {
378.      int index = indexOf(a);
379.      if (index != -1 && index != 0) //is defined and not first
380.          return get(index-1);
381.      else return null;
382.  }
383.
384.  /**
385.   * Returns object which is situated in the Vector just
386.   * after the object given or null
387.   *
388.   * @param O    object to find
389.   * @return     object before the object given
390.   */
391.  public O after(O a)
392.  {
393.      int index = indexOf(a);
394.      if (index != -1 && index != size()-1) //is defined and not last
395.          return get(index+1);
396.      else return null;
397.  }
398.
399.  /**
400.   * Returns the content of the vector in an array
401.   *

```

```

402.      * @return      array of objects
403.      */
404.  public O[] toArray()
405.  {
406.
407.      @SuppressWarnings("unchecked")
408.      O[] a = (O[]) new Object[count];
409.
410.      VectorItem<O> vi = first;
411.
412.      for (int i=0;i<=count-1;i++)
413.      {
414.          a[i] = vi.getObject();
415.          vi = vi.getNext();
416.      }
417.
418.      return a;
419.  }
420.
421.  /**
422.   * Returns the content of the vector in an array of Integers
423.   *
424.   * @return      array of int
425.   */
426.  public int[] toIntArray()
427.  {
428.      int[] a = new int[count];
429.
430.      VectorItem<O> vi = first;
431.
432.      for (int i=0;i<=count-1;i++)

```

```

433.     {
434.         a[i] = (Integer) vi.getObject();
435.         vi = vi.getNext();
436.     }
437.
438.     return a;
439. }
440.
441. /**
442.  * Returns the content of the vector in an array of Integers
443.  *
444.  * @return      array of int
445.  */
446. public long[] toLongArray()
447. {
448.     long[] a = new long[count];
449.
450.     VectorItem<O> vi = first;
451.
452.     for (int i=0;i<=count-1;i++)
453.     {
454.         a[i] = (Long) vi.getObject();
455.         vi = vi.getNext();
456.     }
457.
458.     return a;
459. }
460.
461. /**
462.  * Returns the content of the vector in an array of Items
463.  *

```

```

464.      * @return      array of Item
465.      */
466.  public Item[] toItemArray()
467.  {
468.      Item[] a = new Item[count];
469.
470.      VectorItem<O> vi = first;
471.
472.      for (int i=0;i<=count-1;i++)
473.      {
474.          a[i] = (Item) vi.getObject();
475.          vi = vi.getNext();
476.      }
477.
478.      return a;
479.  }
480.
481.  /**
482.   * Returns the content of the vector in an array of Distributors
483.   *
484.   * @return      array of Distributor
485.   */
486.  public Distributor[] toDistributorArray()
487.  {
488.      Distributor[] a = new Distributor[count];
489.
490.      VectorItem<O> vi = first;
491.
492.      for (int i=0;i<=count-1;i++)
493.      {
494.          a[i] = (Distributor) vi.getObject();

```

```

495.         vi = vi.getNext();
496.     }
497.
498.     return a;
499. }
500.
501.
502. /**
503.  * Returns the content of the vector in an array of Strings
504.  *
505.  * @return      array of Strings
506.  */
507. public String[] toStringArray()
508. {
509.     String[] a = new String[count];
510.
511.     VectorItem<O> vi = first;
512.
513.     for (int i=0;i<=count-1;i++)
514.     {
515.         a[i] = (String) vi.getObject();
516.         vi = vi.getNext();
517.     }
518.
519.     return a;
520. }
521.
522. /**
523.  * Returns the content of the vector in a String
524.  *
525.  * @return      string

```

```

526.      */
527.  public String toString()
528.  {
529.      String s = new String();
530.
531.      VectorItem<O> vi = first;
532.
533.      for (int i=0;i<=count-1;i++)
534.      {
535.          s += vi.getObject().toString()+Static.defaultSeparator;
536.          vi = vi.getNext();
537.      }
538.
539.      return s;
540.  }
541.
542. }
543.

```

## File VectorItem.java

```
1.
2.  /**
3.   * VectorItem object to be used in Vector data structure
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8.  public class VectorItem<O>
9.  {
10.     private O o;
11.     private VectorItem<O> next; //pointer to next VectorItem
12.     private VectorItem<O> previous; //pointer to next VectorItem
13.
14.     /**
15.      * Constructor - initializes object variables to default values
16.      *
17.      */
18.     public VectorItem()
19.     {
20.         o = null;
21.         next = null;
22.         previous = null;
23.     }
24.
25.     /**
26.      * Constructor - initializes object variables to given values
27.      *
28.      */
29.     public VectorItem(O o, VectorItem<O> previous, VectorItem<O> next)
```



```
30.     {
31.         this.o = o;
32.         this.next = next;
33.         this.previous = previous;
34.     }
35.
36.     /**
37.      * Returns the object encapsulated in VectorItem
38.      *
39.      * @return Object
40.      */
41.     public O getObject()
42.     {
43.         return o;
44.     }
45.
46.     /**
47.      * Returns pointer to next VectorItem in Vector.
48.      *
49.      * @return VectorItem
50.      */
51.     public VectorItem<O> getNext()
52.     {
53.         return next;
54.     }
55.
56.     /**
57.      * Returns pointer to next VectorItem in Vector.
58.      *
59.      * @return VectorItem
60.      */
```

```
61. public VectorItem<O> getPrevious()  
62. {  
63.     return previous;  
64. }  
65.  
66.  
67. /**  
68.  * Sets the object encapsulated in VectorItem  
69.  * to given object.  
70.  *  
71.  * @param p position  
72.  */  
73. public void setObject(O o)  
74. {  
75.     this.o = o;  
76. }  
77.  
78. /**  
79.  * Sets pointer to next Item in Vector.  
80.  *  
81.  * @param next Item  
82.  */  
83. public void setNext(VectorItem<O> next)  
84. {  
85.     this.next = next;  
86. }  
87.  
88. /**  
89.  * Sets pointer to previous Item in Vector.  
90.  *  
91.  * @param next Item
```

```
92.      */
93.  public void setPrevious(VectorItem<O> previous)
94.  {
95.      this.previous = previous;
96.  }
97. }
98.
```

## File List.java

```
1.  /**
2.   * Class List - data structure with dynamic allocation of memory.
3.   *
4.   * @author Juraj Masar
5.   * @version 0.1
6.   */
7.  public class List
8.  {
9.      private Item first; //pointer to first item of the list
10.     private boolean sorted; //is the list sorted?
11.
12.     /**
13.      * Constructor - sets object variables to default values
14.      *
15.      */
16.     public List()
17.     {
18.         first = null;
19.         sorted = false;
20.     }
21.
22.     /**
23.      * Constructor - sets object variables to default values
24.      * and flags the list as sorted if given.
25.      *
26.      * @param sorted if the array is supposed to be sorted or not
27.      */
28.     public List (boolean sorted)
29.     {
```

```

30.         first = null;
31.         this.sorted = sorted;
32.     }
33.
34.     /**
35.      * Constructor - sets object variables to given values.
36.      *
37.      * @param first pointer to first item in the array
38.      * @param sorted if the array is supposed to be sorted or not
39.      */
40.     public List(Item first, boolean sorted)
41.     {
42.         this.first = first;
43.         this.sorted = sorted;
44.     }
45.
46.     /**
47.      * Checks if the list is sorted.
48.      *
49.      * @return boolean true if sorted
50.      */
51.     public boolean isSorted ()
52.     {
53.         return sorted;
54.     }
55.
56.     /**
57.      * Checks if the list is empty.
58.      *
59.      * @return boolean true if list is empty
60.      */

```

```
61. public boolean isEmpty()  
62. {  
63.     return (first == null);  
64. }  
65.  
66. /**  
67.  * Returns pointer to first item in the list.  
68.  *  
69.  * @return pointer to first item in the list  
70.  */  
71. public Item getFirst ()  
72. {  
73.     return first;  
74. }  
75.  
76. /**  
77.  * Adds given item to the list according to information  
78.  * whether the list is sorted or not.  
79.  *  
80.  * @param item item to add  
81.  */  
82. public void add(Item newItem)  
83. {  
84.     if (!sorted)  
85.     {  
86.         addToBeginning(newItem);  
87.     }else  
88.     {  
89.         addSorted(newItem);  
90.     }  
91. }
```

```

92.
93.  /**
94.   * Adds item created upon given information to the list
95.   * according to fact whether the list is sorted or not.
96.   *
97.   * @param key    key of the item
98.   * @param pos    position in DAF to which this item corresponds
99.   */
100. public void add(String key, long pos)
101. {
102.     Item newItem = new Item(key,pos,null);
103.     add(newItem);
104. }
105.
106. /**
107.  * Displays the content of the list on output.
108.  * Useful for debugging purposes.
109.  */
110. public void display()
111. {
112.     Item act = first;
113.     System.out.println("List:");
114.     while(act != null)
115.     {
116.         act.print();
117.         act = act.getNext();
118.     }
119. }
120.
121. /**
122.  * Finds and returns Item according to given key.

```

```

123.      *
124.      * @param key key to find
125.      * @return pointer to found item
126.      */
127.  public Item find(String k)
128.  {
129.      if (sorted)
130.      {
131.          return findSorted(k);
132.      } else
133.      {
134.          return findUnsorted(k);
135.      }
136.  }
137.
138.  /**
139.   * Finds and returns position according to given key.
140.   *
141.   * @param key key to find
142.   * @return position of the record
143.   */
144.  public long findPosition(String k)
145.  {
146.      Item i = find(k);
147.      if (i != null)
148.          return i.getPos();
149.      else return -1;
150.  }
151.
152.  /**
153.   * Finds and returns vector of positions according to given key.

```



```

154.      *
155.      * @param key key to find
156.      * @return vector of positions of the records
157.      */
158.  public Vector<Integer> findPositions(String k)
159.  {
160.      Item[] items = findMore(k);
161.      Vector<Integer> positions = new Vector<Integer>();
162.
163.      for (int i=0;i<=items.length-1;i++) positions.pushBack((int) items[i].getPos());
164.
165.      return positions;
166.  }
167.
168.  /**
169.   * Finds and returns vector of positions according to given key.
170.   * The key could be only substring of elements' keys.
171.   *
172.   * @param key key to find
173.   * @return vector of positions of the records
174.   */
175.  public Vector<Integer> findPositionsSubstring(String k)
176.  {
177.      Item[] items = findMoreSubstring(k);
178.      Vector<Integer> positions = new Vector<Integer>();
179.
180.      for (int i=0;i<=items.length-1;i++) positions.pushBack((int) items[i].getPos());
181.
182.      return positions;
183.  }
184.

```

```

185.  /**
186.   * Finds and returns array of Item according to given key.
187.   * The key could be only substring of elements' keys.
188.   *
189.   * @param key key to find
190.   * @return array of pointers to found Item
191.   */
192.  public Item[] findMoreSubstring(String k)
193.  {
194.      if (!isEmpty())
195.      {
196.          Vector<Item> items = new Vector<Item>();
197.          Item act = first;
198.          while (act != null)
199.          {
200.              if (act.getKey().indexOf(k) != -1)
201.                  items.pushBack((Item) act.clone());
202.              act = act.getNext();
203.          }
204.          return items.toItemArray();
205.      }else
206.      {
207.          //the list is empty
208.          return null;
209.      }
210.  }
211.
212.  /**
213.   * Finds and returns array of Item according to given key.
214.   *
215.   * @param key key to find

```

```

216.      * @return array of pointers to found Item
217.      */
218.      public Item[] findMore(String k)
219.      {
220.          if (sorted)
221.          {
222.              return findSortedMore(k);
223.          } else
224.          {
225.              return findUnsortedMore(k);
226.          }
227.      }
228.
229.
230.      /**
231.       * Finds and returns Item according to given position.
232.       *
233.       * @param position position of item to find
234.       * @return pointer to found item
235.       */
236.      public Item findByPosition(long pos)
237.      {
238.          if (isEmpty() == false)
239.          {
240.              Item act = first;
241.              while (act.getPos() != pos)
242.              {
243.                  if (act.getNext() != null)
244.                  {
245.                      act = act.getNext();
246.                  } else

```

```

247.         {
248.             //there is nothing to find
249.             return null;
250.         }
251.     }
252.     return act;
253. }else
254. {
255.     //the list is empty
256.     return null;
257. }
258. }
259.
260.
261. /**
262.  * Adds given item to the beginning of the list.
263.  *
264.  * @param item item to add
265.  */
266. private void addToBeginning(Item newItem)
267. {
268.     if (newItem != null)
269.     {
270.         if (!isEmpty())
271.         {
272.             newItem.setNext(first);
273.         }
274.         first = newItem;
275.     }
276. }
277.

```

```

278.  /**
279.   * Adds given item to the list on the right place assuming the list is sorted.
280.   *
281.   * @param item item to add
282.   */
283.  private void addSorted(Item newItem)
284.  {
285.      if (newItem != null)
286.      {
287.          if (!isEmpty())
288.          {
289.              Item act = first;
290.
291.              if (first.getKey().compareTo(newItem.getKey()) > 0)
292.              {
293.                  addToBeginning(newItem);
294.              } else if (act.getNext() != null)
295.              {
296.                  while (act.getNext().getKey().compareTo(newItem.getKey()) < 0)
297.                  {
298.                      act = act.getNext();
299.                      if (act.getNext() == null) break;
300.                  }
301.                  newItem.setNext(act.getNext());
302.                  act.setNext(newItem);
303.              } else
304.              {
305.                  act.setNext(newItem);
306.              }
307.          } else
308.          {

```

```

309.         first = newItem;
310.     }
311. }
312. }
313.
314. /**
315.  * Finds and returns Item according to given key
316.  * assuming the list is sorted.
317.  *
318.  * @param key key to find
319.  * @return pointer to found item
320.  */
321. private Item findSorted(String k)
322. {
323.     if (!isEmpty())
324.     {
325.         Item act = first;
326.         while (act != null && act.getKey().compareTo(k) < 0)
327.         {
328.             act = act.getNext();
329.         }
330.         if (act != null && act.getKey().equals(k))
331.         {
332.             return act;
333.         } else
334.         {
335.             //there is nothing to find
336.             return null;
337.         }
338.     } else
339.     {

```

```

340.         //the list is empty
341.         return null;
342.     }
343. }
344.
345. /**
346.  * Finds and returns array of Item according to given key
347.  * assuming the list is sorted.
348.  *
349.  * @param key key to find
350.  * @return array of pointers to found Item
351.  */
352. private Item[] findSortedMore(String k)
353. {
354.     if (!isEmpty())
355.     {
356.         Vector<Item> items = new Vector<Item>();
357.         Item act = first;
358.         while (act != null && act.getKey().compareTo(k) < 0)
359.         {
360.             act = act.getNext();
361.         }
362.         if(act != null && act.getKey().equals(k))
363.         {
364.             while (act != null && act.getKey().equals(k))
365.             {
366.                 items.pushBack((Item)act.clone());
367.                 act = act.getNext();
368.             }
369.             return items.toItemArray();
370.         }else
  
```

```

371.         {
372.             //there is nothing to find
373.             return null;
374.         }
375.     }else
376.     {
377.         //the list is empty
378.         return null;
379.     }
380. }
381.
382. /**
383.  * Finds and returns array of Item according to given key
384.  * assuming the list is unsorted.
385.  *
386.  * @param key key to find
387.  * @return array of pointers to found Item
388.  */
389. private Item[] findUnsortedMore(String k)
390. {
391.     if (!isEmpty())
392.     {
393.         Vector<Item> items = new Vector<Item>();
394.         Item act = first;
395.         while (act != null)
396.         {
397.             if (act.getKey().equals(k)) items.pushBack((Item)act.clone());
398.             act = act.getNext();
399.         }
400.         return items.toItemArray();
401.     }else

```



```

402.         {
403.             //the list is empty
404.             return null;
405.         }
406.     }
407.
408.     /**
409.      * Finds and returns Item according to given key
410.      * assuming the list is unsorted.
411.      *
412.      * @param key key to find
413.      * @return pointer to found item
414.      */
415.     private Item findUnsorted(String k)
416.     {
417.         if (isEmpty() == false)
418.         {
419.             Item act = first;
420.             while (act.getKey().compareTo(k) != 0)
421.             {
422.                 if (act.getNext() != null)
423.                 {
424.                     act = act.getNext();
425.                 } else
426.                 {
427.                     //there is nothing to find
428.                     return null;
429.                 }
430.             }
431.             return act;
432.         } else

```

```
433.     {  
434.         //the list is empty  
435.         return null;  
436.     }  
437. }  
438.  
439. }  
440.
```

## File Item.java

```
1.
2.  /**
3.   * Item object to be used in List data structure
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8.  public class Item implements Cloneable
9.  {
10.     protected String key; //according to which items are sort
11.     protected long pos; //position in DirectAccessFile
12.     protected Item next; //pointer to next item in list
13.
14.     /** DOSSIER: HL mastery 8 - encapsulation */
15.
16.     /**
17.      * Constructor - initializes object variables to default values
18.      *
19.      */
20.     public Item()
21.     {
22.         key = "";
23.         pos = -1;
24.         next = null;
25.     }
26.
27.     /**
28.      * Constructor - initializes object variables to given values
29.      *
```

```

30.     */
31.     public Item(String key, long pos, Item next)
32.     {
33.         this.key = key;
34.         this.pos = pos;
35.         this.next = next;
36.     }
37.
38.     /**
39.      * Produces a clone of current object
40.      *
41.      * @return      Object instance
42.      */
43.     public Object clone()
44.     {
45.         try
46.         {
47.             return super.clone();
48.         }
49.         catch (CloneNotSupportedException e) {
50.             return null;
51.         }
52.     }
53.
54.
55.     /**
56.      * Returns key - string according to which items can be sorted.
57.      *
58.      * @return key
59.      */
60.     public String getKey()

```

```

61.     {
62.         return key;
63.     }
64.
65.     /**
66.      * Returns position of record with key set in this item in DAF.
67.      *
68.      * @return position
69.      */
70.     public long getPos()
71.     {
72.         return pos;
73.     }
74.
75.     /**
76.      * Returns pointer to next Item in List.
77.      *
78.      * @return Item
79.      */
80.     public Item getNext()
81.     {
82.         return next;
83.     }
84.
85.     /**
86.      * Sets key - string according to which list can be sorted.
87.      *
88.      * @param k key
89.      */
90.     public void setKey(String k)
91.     {

```

```

92.         key = k;
93.     }
94.
95.     /**
96.      * Sets position of record with key set in this item in DAF.
97.      *
98.      * @param p position
99.      */
100.    public void setPos(long p)
101.    {
102.        pos = p;
103.    }
104.    /**
105.     * Sets pointer to next Item in List.
106.     *
107.     * @param next Item
108.     */
109.    public void setNext(Item n)
110.    {
111.        next = n;
112.    }
113.    /**
114.     * Prints the content of the Item.
115.     * Only for debugging purposes.
116.     *
117.     */
118.    public void print()
119.    {
120.        System.out.println(getKey()+" "+getPos());
121.    }
122. }

```

## File ItemRegNum.java

```
1.
2.  /**
3.   * Item object to be used in listRegNum data structure
4.   *
5.   * @author Juraj Masar
6.   * @version 0.1
7.   */
8.  public class ItemRegNum extends Item
9.  {
10.     public TreeItem treeItem;
11.     public ProfileWindow window;
12.     /**
13.      * Constructor - initializes object variables to given values
14.      *
15.      */
16.     public ItemRegNum(String key, long pos, Item next)
17.     {
18.         this.key = key;
19.         this.pos = pos;
20.         this.next = next;
21.         treeItem = null;
22.         window = null;
23.     }
24.
25.     /**
26.      * Prints the content of the Item.
27.      * Only for debugging purposes.
28.      *
29.      */
```

```
30.     public void print()  
31.     {  
32.         System.out.println(getKey()+" "+getPos()+" "  
33.             +((treeItem == null) ? "null" : treeItem.regNum));  
34.     }  
35. }  
36.
```



## File ItemSponsor.java

```
1. import java.util.Date;
2.
3. /**
4.  * Item object to be used in listRegNum data structure
5.  *
6.  * @author JuraĹ Masar
7.  * @version 0.1
8.  */
9. public class ItemSponsor extends Item
10. {
11.     public String regNum; //registration number of this distributor
12.     public Date registrationDate; //registration date of this distributor
13.     public String name;
14.     public String surname;
15.     public ItemRegNum itemRegNum; //pointer to item of listRegNum
16.     /**
17.      * Constructor - initializes object variables to given values
18.      *
19.      * @param key key of item
20.      * @param position position in file
21.      * @param next position to next Item
22.      * @param regNum registration number
23.      * @param registrationDate registrationDate
24.      * @param itemRegNum pointer to ItemRegNum
25.      */
26.     public ItemSponsor(
27.         String key,
28.         long pos,
29.         Item next,
```

```
30.     String regNum,  
31.     String registrationDate,  
32.     ItemRegNum itemRegNum,  
33.     String name,  
34.     String surname  
35. )  
36. {  
37.     this.key = key;  
38.     this.pos = pos;  
39.     this.next = next;  
40.     this.regNum = regNum;  
41.     this.registrationDate = Static.stringToDate(registrationDate);  
42.     this.itemRegNum = itemRegNum;  
43.     this.name = name;  
44.     this.surname = surname;  
45. }  
46. }  
47.
```

## File TreeItem.java

```
1.  import java.util.Date;
2.  import javax.swing.tree.DefaultMutableTreeNode;
3.  /**
4.   * Item of Tree object - describing the structure in MLM system.
5.   *
6.   * @author Juraj Masar
7.   * @version 0.1
8.   */
9.  public class TreeItem implements Cloneable
10. {
11.     public String regNum; //registration number of a distributor
12.     public TreeItem[] children; //array of TreeItems for children of this distributor
13.     public TreeItem parent; //TreeItem of the sponsor of this distributor
14.     public String name;
15.     public String surname;
16.     public Date registrationDate; //registration date of this distributor
17.     public long position; //position in file of this distributor
18.
19.     /**
20.      * Constructor of TreeItem - initialize the regNum and variables
21.      */
22.     public TreeItem (String regNum)
23.     {
24.         this.regNum = regNum;
25.         parent = null;
26.         registrationDate = null;
27.         position = -1;
28.         children = null;
29.     }
```

```

30.
31.  /**
32.   * Returns a clone of current object
33.   *
34.   * @return      clone of current Object
35.   */
36.  public Object clone()
37.  {
38.      try
39.      {
40.          return super.clone();
41.      }
42.      catch (CloneNotSupportedException e) {
43.          return null;
44.      }
45.  }
46.
47.  /**
48.   * Returns the number of direct children of this distributor
49.   *
50.   * @return      number of direct children
51.   */
52.  public int getNumberOfDirectChildren()
53.  {
54.      if (children == null) return 0;
55.
56.      return children.length;
57.  }
58.
59.  /**
60.   * Returns total number of children

```

```

61.      *
62.      * @return      number of children
63.      */
64.  public int getNumberOfAllChildren()
65.  {
66.      if (children == null) return 0;
67.
68.      int count = 0;
69.
70.      for (int i=0;i<children.length;i++)
71.      {
72.          count += 1 + children[i].getNumberOfAllChildren();
73.      }
74.
75.      return count;
76.
77.  }
78.
79.  /**
80.   * Prints this TreeItem and all its children.
81.   * For debugging purposes.
82.   *
83.   * @param level      height in which the item is
84.   */
85.  public void print(int level)
86.  {
87.      for (int i=0;i<level;i++) System.out.print("-");
88.      System.out.println(regNum+" "+Static.dateToString(registrationDate));
89.      if (children != null )for (int i=0;i<children.length;i++) children[i].print(level+1);
90.  }
91.

```

```

92.      /**
93.       * Checks whether the distributor was registered before given date.
94.       *
95.       * @param s date given in string
96.       * @return true if was
97.       */
98.      public boolean wasRegisteredBefore(String s)
99.      {
100.         return (Static.stringToDate(s).compareTo(registrationDate) > 0);
101.      }
102.
103.      /**
104.       * Counts direct children distributors registered before
105.       * given date.
106.       *
107.       * @param s date given in string
108.       * @return number of distributors before given date
109.       */
110.      public int getNumberOfDirectChildrenBefore(String s)
111.      {
112.         if (!wasRegisteredBefore(s)) return 0;
113.
114.         int count = 0;
115.
116.         if (children != null)
117.             for (int i=0;i<children.length;i++)
118.                 if (children[i].registrationDate.compareTo(Static.stringToDate(s)) < 0) count++;
119.
120.         return count;
121.      }
122.

```

```

123.  /**
124.   * Counts all children distributors registered before
125.   * given date.
126.   *
127.   * @param s date given in string
128.   * @return number of distributors before given date
129.   */
130. public int getNumberOfAllChildrenBefore(String s)
131. {
132.     if (!wasRegisteredBefore(s)) return 0;
133.
134.     int count = 0;
135.
136.     if (children != null)
137.         for (int i=0;i<children.length;i++)
138.             {
139.                 if (children[i].registrationDate.compareTo(Static.stringToDate(s)) < 0)
140.                 {
141.                     count += 1 + children[i].getNumberOfAllChildrenBefore(s);
142.                 }
143.             }
144.
145.     return count;
146. }
147.
148. /**
149.   * Returns key - a String identifier of this
150.   * distributor.
151.   *
152.   * @return String identifier
153.   */

```

```

154. public String getKey()
155. {
156.     Distributor d = DistributorManager.getDistributorByRegNum(regNum);
157.     return regNum+" (" +d.getFullname()+") ";
158. }
159.
160. /**
161.  * Appends to node to JTree fulfilled with data about distributor and
162.  * all its children.
163.  *
164.  */
165. public void setJTreeNode(DefaultMutableTreeNode top, boolean first)
166. {
167.     if (top != null)
168.     {
169.         if (first)
170.         {
171.             top.setUserObject(getKey());
172.         } else {
173.             DefaultMutableTreeNode item = new DefaultMutableTreeNode(getKey());
174.             top.add (item);
175.             top = item;
176.         }
177.         if (children != null)
178.         for (int i=0;i<children.length;i++)
179.         {
180.             children[i].setJTreeNode(top, false);
181.         }
182.     }
183. }
184.

```



```

185.  /**
186.   * Appends to node to JTree fulfilled with data about distributor and
187.   * all its children.
188.   *
189.   */
190.  public void setJTreeNode(CheckNode top, boolean first)
191.  {
192.      if (top != null)
193.      {
194.          if (first)
195.          {
196.              top.setUserObject(getKey());
197.          } else {
198.              CheckNode item = new CheckNode(getKey());
199.              top.add (item);
200.              top = item;
201.          }
202.          if (children != null)
203.              for (int i=0;i<children.length;i++)
204.              {
205.                  children[i].setJTreeNode(top, false);
206.              }
207.      }
208.  }
209.
210.  /**
211.   * Set the position in file of current distributor
212.   * to Vector<Long> given in parameters and
213.   * recursively sends this command to ist children
214.   * distributors.
215.   *

```

```

216.      * @param positions    Vector of positions
217.      */
218.  public void getPositions(Vector<Long> positions, boolean includeCurrent)
219.  {
220.      if (includeCurrent) positions.pushBack(position);
221.
222.      if (children != null)
223.      for (int i=0;i<children.length;i++)
224.      {
225.          children[i].getPositions(positions, true);
226.      }
227.  }
228. }

```

## File MainWindow.java

```
1.  import javax.swing.*;
2.  import java.awt.event.ActionEvent;
3.  import java.awt.event.ActionListener;
4.  import java.awt.event.*;
5.
6.
7.  /**
8.   * MainWindows of the application.
9.   * Customized JFrame.
10.  *
11.  * @author Juraj Masar
12.  * @version 0.1
13.  */
14. public class MainWindow extends JFrame implements ActionListener
15. {
16.     final String[] searchResultsTableHeader;
17.
18.     //definitions of controls
19.     private JButton closeSearchBtn;
20.     private JButton aboutBtn;
21.     private JButton addDistributorBtn;
22.     private JButton changePasswordBtn;
23.     private JButton exitBtn;
24.     private JButton managePointsBtn;
25.     private JButton myProfileBtn;
26.     private TextField nameField;
27.     private JLabel nameLabel;
28.     private TextField regNumField;
29.     private JLabel regNumLabel;
```

```
30. private JSeparator resultsSeparator;
31. private JScrollPane scrollPane;
32. public JButton searchBtn;
33. private JPanel searchPanel;
34. private JLabel searchResultsLabel;
35. private JTable searchResultsTable;
36. private JTextField surnameField;
37. private JLabel surnameLabel;
38. private JLabel welcomeLabel;
39. private JLabel countLabel;
40.
41. /**
42.  * Designs and creates the window.
43.  */
44. public MainWindow()
45. {
46.
47.     //gui
48.     welcomeLabel = new JLabel();
49.     countLabel = new JLabel();
50.     myProfileBtn = new JButton();
51.     addDistributorBtn = new JButton();
52.     managePointsBtn = new JButton();
53.     exitBtn = new JButton();
54.     changePasswordBtn = new JButton();
55.     searchPanel = new JPanel();
56.     nameField = new JTextField();
57.     nameLabel = new JLabel();
58.     surnameField = new JTextField();
59.     surnameLabel = new JLabel();
60.     regNumField = new JTextField();
```

```
61.         regNumLabel = new JLabel();
62.         searchBtn = new JButton();
63.         resultsSeparator = new JSeparator();
64.         searchResultsLabel = new JLabel();
65.         scrollPane = new JScrollPane();
66.         searchResultsTable = new JTable();
67.         aboutBtn = new JButton();
68.         closeSearchBtn = new JButton();
69.
70.         setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
71.         getContentPane().setLayout(null);
72.
73.         welcomeLabel.setText("Welcome to Multilevel Marketing Manager v1.0");
74.         getContentPane().add(welcomeLabel);
75.         welcomeLabel.setBounds(10, 11, 223, 14);
76.
77.         updateCounter();
78.         getContentPane().add(countLabel);
79.         countLabel.setBounds(10, 120, 250, 14);
80.
81.         myProfileBtn.setText("My Profile");
82.         getContentPane().add(myProfileBtn);
83.         myProfileBtn.setBounds(10, 31, 117, 23);
84.         myProfileBtn.addActionListener(this);
85.
86.         addDistributorBtn.setText("Add distributor");
87.         getContentPane().add(addDistributorBtn);
88.         addDistributorBtn.setBounds(10, 60, 117, 23);
89.         addDistributorBtn.addActionListener(this);
90.
91.         managePointsBtn.setText("Manage points");
```

```
92.         getContentPane().add(managePointsBtn);
93.         managePointsBtn.setBounds(10, 89, 117, 23);
94.         managePointsBtn.addActionListener(this);
95.
96.         exitBtn.setText("Exit");
97.         getContentPane().add(exitBtn);
98.         exitBtn.setBounds(133, 90, 120, 23);
99.         exitBtn.addActionListener(this);
100.
101.         changePasswordBtn.setText("Change password");
102.         getContentPane().add(changePasswordBtn);
103.         changePasswordBtn.setBounds(133, 31, 120, 23);
104.         changePasswordBtn.addActionListener(this);
105.
106.         searchPanel.setBorder(BorderFactory.createTitledBorder("Search for distributor"));
107.         searchPanel.setLayout(null);
108.         searchPanel.add(nameField);
109.
110.         nameField.setBounds(79, 20, 143, 20);
111.         nameLabel.setText("Name:");
112.         searchPanel.add(nameLabel);
113.         nameField.addActionListener(this);
114.
115.         nameLabel.setBounds(16, 23, 31, 14);
116.         searchPanel.add(surnameField);
117.         surnameField.setBounds(79, 46, 143, 20);
118.         surnameField.addActionListener(this);
119.
120.         surnameLabel.setText("Surname:");
121.         searchPanel.add(surnameLabel);
122.         surnameLabel.setBounds(16, 49, 46, 14);
```

```
123.      searchPanel.add(regNumField);
124.      regNumField.setBounds(79, 72, 143, 20);
125.      regNumField.addActionListener(this);
126.
127.      regNumLabel.setText("Reg. num:");
128.      searchPanel.add(regNumLabel);
129.      regNumLabel.setBounds(16, 75, 50, 14);
130.
131.      searchBtn.setText("Search");
132.      searchPanel.add(searchBtn);
133.      searchBtn.setBounds(16, 98, 206, 23);
134.      searchBtn.addActionListener(this);
135.
136.      getContentPane().add(searchPanel);
137.      searchPanel.setBounds(278, 11, 238, 130);
138.      getContentPane().add(resultsSeparator);
139.      resultsSeparator.setBounds(10, 150, 500, 2);
140.
141.      searchResultsLabel.setText("Search Results");
142.      getContentPane().add(searchResultsLabel);
143.      searchResultsLabel.setBounds(10, 160, 450, 14);
144.
145.      closeSearchBtn.setText("close");
146.      getContentPane().add(closeSearchBtn);
147.      closeSearchBtn.setBounds(455, 155, 57, 23);
148.      closeSearchBtn.addActionListener(this);
149.
150.      searchResultsTableHeader = new String [] {
151.          "Reg. num.", "Name", "Surname", "E-mail", "Country"
152.      };
153.
```

```

154.     searchResultsTable.setModel(new NonEditableDefaultTableModel(
155.         new Object [][ ] {
156.             null
157.         }, searchResultsTableHeader
158.     ));
159.     scrollPane.setViewportViewView(searchResultsTable);
160.
161.     getContentPane().add(scrollPane);
162.     scrollPane.setBounds(10, 180, 500, 150);
163.
164.     aboutBtn.setText("About");
165.     getContentPane().add(aboutBtn);
166.     aboutBtn.setBounds(133, 60, 120, 23);
167.     aboutBtn.addActionListener(this);
168.
169.
170.     setIconImage(Dialog.createImageIcon(Static.iconPath, "Manager").getImage());
171.     setResizable(false);
172.     hideSearch(); //setSize
173.     setLocationRelativeTo(null); //center
174.     setTitle("Multilevel Marketing Manager");
175.
176.     //display
177.     setVisible(true);
178. }
179.
180. /**
181.  * Shows search results - changes the size of the window
182.  *
183.  */
184. public void showSearch ()

```



```

185.     {
186.         setSize(530, 370);
187.     }
188.
189.     /**
190.      * Hides search results - changes the size of the window
191.      *
192.      */
193.     public void hideSearch ()
194.     {
195.         setSize(530, 175);
196.     }
197.
198.     /**
199.      * Updates the number of distributors displayed in the window.
200.      *
201.      */
202.     public void updateCounter()
203.     {
204.         if (DistributorManager.getCount() == 0)
205.             countLabel.setText("There is currently no distributor in the system.");
206.         else if (DistributorManager.getCount() == 1)
207.             countLabel.setText("There is currently one distributor in the system.");
208.         else
209.             countLabel.setText("There are currently "+DistributorManager.getCount()
210.                                 +" distributors in the system.");
211.     }
212.
213.     /**
214.      * Defines action when ActionEvent on controls
215.      * at this frame fires

```

```

216.      *
217.      * @param y    a sample parameter for a method
218.      * @return    the sum of x and y
219.      */
220.      public void actionPerformed(ActionEvent e)
221.      {
222.          if (e.getSource () == searchBtn ||
223.              e.getSource () == nameField ||
224.              e.getSource () == surnameField ||
225.              e.getSource () == regNumField
226.          )
227.          {
228.              if (regNumField.getText().equals("") &&
229.                  nameField.getText().equals("") &&
230.                  surnameField.getText().equals(""))
231.              {
232.                  Dialog.warning ("Please insert what to search for.");
233.                  return;
234.              }
235.              Distributor[] distributors = DistributorManager.find(regNumField.getText(),
236.                                                                      nameField.getText(), surnameField.getText());
237.              final Object[][] objects = new Object[distributors.length][5];
238.
239.              for (int i=0;i<distributors.length;i++)
240.              {
241.                  objects[i][0] = distributors[i].getRegNum();
242.                  objects[i][1] = distributors[i].getName();
243.                  objects[i][2] = distributors[i].getSurname();
244.                  objects[i][3] = distributors[i].getEmail();
245.                  objects[i][4] = distributors[i].getCountry();
246.

```

```

247.     }
248.
249.     //insert new data
250.     searchResultsTable.setModel(
251.         new NonEditableDefaultTableModel(objects, searchResultsTableHeader)
252.     );
253.
254.     //disable dragndrop
255.     searchResultsTable.setDragEnabled(false);
256.     searchResultsTable.getTableHeader().setReorderingAllowed(false);
257.
258.     //set email column width
259.     searchResultsTable.getColumnModel().getColumn(0).setPreferredWidth(25);
260.     //set email column width
261.     searchResultsTable.getColumnModel().getColumn(3).setPreferredWidth(100);
262.     //set country column width
263.     searchResultsTable.getColumnModel().getColumn(4).setPreferredWidth(25);
264.
265.     searchResultsLabel.setText("Search Results for "
266.                                +nameField.getText()+"/"
267.                                +surnameField.getText()+"/"
268.                                +regNumField.getText()
269.     );
270.
271.     searchResultsTable.addMouseListener(new MouseAdapter()
272.     {
273.         public void mouseClicked(MouseEvent e)
274.         {
275.             if (e.getClickCount() == 2)
276.             {
277.                 JTable target = (JTable)e.getSource();

```

```

278.         int row = target.getSelectedRow();
279.
280.         String regNum = (String)searchResultsTable.getValueAt(row, 0);
281.
282.         if (!Dialog.displayProfile(regNum))
283.             Dialog.error ("Such distributor does not exist anymore!");
284.
285.     }
286. }
287. });
288.
289.
290.     //enlarge form
291.     showSearch();
292. } else if (e.getSource () == closeSearchBtn)
293. {
294.     hideSearch();
295. } else if (e.getSource () == aboutBtn)
296. {
297.     Dialog.info ("Multilevel Marketing Manager 1.0"
298.         + " is an application for managing human resources.\nIt has been originally "
299.         + "created for the MLM system Network World Alliance."
300.         + " \n\n Author: Juraj Masar (2011) mail@jurajmasar.com",
301.         "Multilevel Marketing Manager - About");
302. } else if (e.getSource () == exitBtn)
303. {
304.     dispose();
305. } else if (e.getSource () == changePasswordBtn)
306. {
307.     new ChangePasswordDialog (this, "Change Password", true);
308. } else if (e.getSource () == addDistributorBtn)

```

```

309.      {
310.          new AddDistributorDialog (this, "Add Distributor", true);
311.      } else if (e.getSource() == myProfileBtn)
312.      {
313.          myProfileBtn.setEnabled(false);
314.          Distributor d = DistributorManager.getDistributorBySponsor("0");
315.          if (d != null)
316.              Dialog.displayProfile(d.getRegNum());
317.          else
318.              //data is inconcistent
319.              Dialog.error("Error: Data of the application is inconsistent.\n"
320.                  + "The root distributor cannot be found.\n"
321.                  + "Please, reinstall the application."
322.              );
323.          myProfileBtn.setEnabled(true);
324.      } else if (e.getSource() == managePointsBtn)
325.      {
326.          String m = Dialog.askForMonth();
327.          if (m == null) return;
328.
329.          new DistributorTreeDialog (m);
330.
331.      }
332.  }
333.  }
334.  }
  
```

## File ProfileWindow.java

```

1.  import javax.swing.*;
2.  import javax.swing.table.*;
3.  import javax.swing.table.DefaultTableModel;
4.  import javax.swing.event.*;
5.  import java.awt.event.ActionEvent;
6.  import java.awt.event.ActionListener;
7.  import javax.swing.text.*;
8.  import javax.swing.border.*;
9.  import java.awt.Color;
10. import org.jfree.chart.ChartPanel;
11. import java.awt.Dimension;
12.
13. /**
14.  * Profile window of a distributor - customized JFrame.
15.  * Displays info about particular distributor.
16.  *
17.  * @author Juraj Masar
18.  * @version 0.1
19.  */
20. public class ProfileWindow extends JFrame implements ActionListener
21. {
22.     private Border defaultBorder;
23.     private Distributor d;
24.     private Month[] months;
25.
26.     //definitions of controls
27.     private JFormattedTextField birthDateField;
28.     private JLabel birthDateLabel;
29.     private JLabel editableLabel;

```

```

30.     private JButton childTableBtn;
31.     private JButton childTreeBtn;
32.     private JCheckBox consumerCheckbox;
33.     private JTextFieldMaxLength countryField;
34.     private JLabel countryLabel;
35.     private JButton deleteBtn;
36.     private JTextFieldMaxLength emailField;
37.     private JLabel emailLabel;
38.     private JScrollPane jScrollPane1;
39.     private JCheckBox managerCheckbox;
40.     private JTextFieldMaxLength nameField;
41.     private JLabel nameLabel;
42.     private JTextPane noteField;
43.     private JLabel noteLabel;
44.     private JFormattedTextField registrationDateField;
45.     private JLabel registrationDateLabel;
46.     private JCheckBox salesmanCheckbox;
47.     private JButton saveBtn;
48.     private JScrollPane scrollPane;
49.     private JSeparator separator;
50.     private JComboBox sponsorBox;
51.     private JLabel sponsorLabel;
52.     private JLabel statisticsLabel;
53.     private JTable statsTable;
54.     private JLabel statusLabel;
55.     private JTextFieldMaxLength surnameField;
56.     private JLabel surnameLabel;
57.     private JTextFieldMaxLength telephoneField;
58.     private JLabel telephoneLabel;
59.     private JLabel titleLabel;
60.     private ChartPanel chartPanel;

```

```

61.
62.  /**
63.   * Designs and creates the window.
64.   *
65.   * @param d Distributor object to display
66.   */
67. public ProfileWindow(Distributor d)
68. {
69.     this.d = d;
70.
71.     editableLabel = new JLabel();
72.     statsTable = new JTable();
73.     titleLabel = new JLabel();
74.     nameLabel = new JLabel();
75.     surnameLabel = new JLabel();
76.     sponsorLabel = new JLabel();
77.     countryLabel = new JLabel();
78.     registrationDateLabel = new JLabel();
79.     birthDateLabel = new JLabel();
80.     emailLabel = new JLabel();
81.     telephoneLabel = new JLabel();
82.     noteLabel = new JLabel();
83.     nameField = new JTextFieldMaxLength(20);
84.     surnameField = new JTextFieldMaxLength(20);
85.     emailField = new JTextFieldMaxLength(50);
86.     telephoneField = new JTextFieldMaxLength(20);
87.     countryField = new JTextFieldMaxLength(3);
88.     registrationDateField = new JFormattedTextField();
89.     sponsorBox = new JComboBox();
90.     jScrollPane1 = new JScrollPane();
91.     noteField = new JTextPane();
  
```



```
92.         saveBtn = new JButton();
93.         deleteBtn = new JButton();
94.         separator = new JSeparator();
95.         statisticsLabel = new JLabel();
96.         scrollPane = new JScrollPane();
97.         birthDateField = new JFormattedTextField();
98.         childTreeBtn = new JButton();
99.         childTableBtn = new JButton();
100.        statusLabel = new JLabel();
101.        consumerCheckbox = new JCheckBox();
102.        salesmanCheckbox = new JCheckBox();
103.        managerCheckbox = new JCheckBox();
104.
105.        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
106.        getContentPane().setLayout(null);
107.
108.        titleLabel.setFont(new java.awt.Font("Tahoma", 0, 18));
109.        getContentPane().add(titleLabel);
110.        titleLabel.setBounds(10, 10, 650, 22);
111.
112.        nameLabel.setText("* Name:");
113.        getContentPane().add(nameLabel);
114.        nameLabel.setBounds(20, 47, 45, 14);
115.
116.        surnameLabel.setText("* Surname:");
117.        getContentPane().add(surnameLabel);
118.        surnameLabel.setBounds(20, 73, 55, 14);
119.
120.        countryLabel.setText("* Country:");
121.        getContentPane().add(countryLabel);
122.        countryLabel.setBounds(326, 47, 55, 14);
```

```
123.
124.     registrationDateLabel.setText("* Date of registration:");
125.     getContentPane().add(registrationDateLabel);
126.     registrationDateLabel.setBounds(326, 73, 110, 14);
127.
128.     birthDateLabel.setText("* Date of birth:");
129.     getContentPane().add(birthDateLabel);
130.     birthDateLabel.setBounds(20, 99, 75, 14);
131.
132.     emailLabel.setText("E-mail:");
133.     getContentPane().add(emailLabel);
134.     emailLabel.setBounds(20, 125, 32, 14);
135.
136.     telephoneLabel.setText("Telephone:");
137.     getContentPane().add(telephoneLabel);
138.     telephoneLabel.setBounds(20, 151, 54, 14);
139.
140.     noteLabel.setText("Note:");
141.     getContentPane().add(noteLabel);
142.     noteLabel.setBounds(20, 174, 27, 14);
143.
144.     nameField.setText(d.getName());
145.     getContentPane().add(nameField);
146.     nameField.setBounds(107, 44, 190, 20);
147.
148.     surnameField.setText(d.getSurname());
149.     getContentPane().add(surnameField);
150.     surnameField.setBounds(107, 70, 190, 20);
151.
152.     emailField.setText(d.getEmail());
153.     getContentPane().add(emailField);
```

```

154.     emailField.setBounds(107, 122, 190, 20);
155.
156.     telephoneField.setText(d.getTelephone());
157.     getContentPane().add(telephoneField);
158.     telephoneField.setBounds(107, 148, 190, 20);
159.
160.     countryField.setText(d.getCountry());
161.     getContentPane().add(countryField);
162.     countryField.setBounds(437, 44, 190, 20);
163.
164.     registrationDateField.setText(d.getRegistrationDate());
165.     getContentPane().add(registrationDateField);
166.     registrationDateField.setBounds(437, 70, 190, 20);
167.     registrationDateField.setFormatterFactory(
168.         new DefaultFormatterFactory(new DateFormatter(
169.             new java.text.SimpleDateFormat("dd.MM.yyyy"))));
170.     registrationDateField.setText(d.getRegistrationDate());
171.
172.     if (!d.isRoot())
173.     {
174.         sponsorLabel.setText("* Sponsor:");
175.         getContentPane().add(sponsorLabel);
176.         sponsorLabel.setBounds(326, 125, 55, 14);
177.
178.         String[] sponsors = new String[DistributorManager.getCount()-1];
179.         ItemRegNum item = (ItemRegNum) DistributorManager.listRegNum.getFirst();
180.         int selected = 0, i = 0;
181.         while (item != null)
182.         {
183.             if (!item.getKey().equals(d.getRegNum()))
184.             {

```

```

185.         sponsors[i] = item.getKey()+" (" +item.treeItem.name+" "+item.treeItem.surname+" )";
186.         if (item.getKey().equals(d.getSponsor())) selected = i;
187.         i++;
188.     }
189.     item = (ItemRegNum) item.getNext();
190. }
191. sponsorBox.setModel(new DefaultComboBoxModel(sponsors));
192. sponsorBox.setSelectedIndex(selected);
193.
194. getContentPane().add(sponsorBox);
195. sponsorBox.setBounds(437, 122, 190, 20);
196. }
197.
198. noteField.setText(d.getNote());
199. jScrollPane1.setViewportViewView(noteField);
200. getContentPane().add(jScrollPane1);
201. jScrollPane1.setBounds(107, 175, 190, 51);
202.
203. saveBtn.setText("Save changes");
204. getContentPane().add(saveBtn);
205. saveBtn.setBounds(320, 180, 115, 23);
206. saveBtn.addActionListener(this);
207.
208. if (!d.isRoot())
209. {
210.     deleteBtn.setText("Delete distributor");
211.     getContentPane().add(deleteBtn);
212.     deleteBtn.addActionListener(this);
213.     deleteBtn.setBounds(320, 210, 115, 23);
214. }
215.
  
```

```

216.     getContentPane().add(separator);
217.     separator.setBounds(20, 241, 640, 10);
218.
219.     statisticsLabel.setFont(new java.awt.Font("Tahoma", 0, 18));
220.     statisticsLabel.setText("Statistics");
221.     getContentPane().add(statisticsLabel);
222.     statisticsLabel.setBounds(20, 257, 69, 22);
223.
224.     editableLabel.setText("* are editable (use doubleclicking)");
225.     getContentPane().add(editableLabel);
226.     editableLabel.setBounds(470, 265, 170, 14);
227.
228.     /** table **/
229.
230.     updateTable();
231.
232.     scrollPane.setViewportViewView(statsTable);
233.
234.     getContentPane().add(scrollPane);
235.     scrollPane.setBounds(20, 290, 640, 155);
236.
237.     /** end of table **/
238.
239.
240.     birthDateField.setFormatterFactory(new DefaultFormatterFactory(new DateFormatter()));
241.     birthDateField.setText(d.getBirthDate());
242.     getContentPane().add(birthDateField);
243.     birthDateField.setBounds(107, 96, 190, 20);
244.     birthDateField.setFormatterFactory(
245.         new DefaultFormatterFactory(new DateFormatter(
246.             new java.text.SimpleDateFormat("dd.MM.yyyy"))));

```

```

247.         birthDateField.setText(d.getBirthDate());
248.
249.         childTreeBtn.setText("Open child distributors in tree");
250.         getContentPane().add(childTreeBtn);
251.         childTreeBtn.setBounds(450, 210, 180, 23);
252.         childTreeBtn.addActionListener(this);
253.
254.         childTableBtn.setText("Open child distributors in table");
255.         getContentPane().add(childTableBtn);
256.         childTableBtn.setBounds(450, 180, 180, 23);
257.         childTableBtn.addActionListener(this);
258.
259.         statusLabel.setText("Status:");
260.         getContentPane().add(statusLabel);
261.         statusLabel.setBounds(326, 99, 35, 14);
262.
263.         consumerCheckbox.setText("Consumer");
264.         getContentPane().add(consumerCheckbox);
265.         consumerCheckbox.setBounds(414, 97, 73, 23);
266.         if (d.getStatus().indexOf('C') != -1) consumerCheckbox.setSelected(true);
267.
268.         salesmanCheckbox.setText("Salesman");
269.         getContentPane().add(salesmanCheckbox);
270.         salesmanCheckbox.setBounds(487, 97, 71, 23);
271.         if (d.getStatus().indexOf('S') != -1) salesmanCheckbox.setSelected(true);
272.
273.         managerCheckbox.setText("Manager");
274.         getContentPane().add(managerCheckbox);
275.         managerCheckbox.setBounds(560, 97, 67, 23);
276.         if (d.getStatus().indexOf('M') != -1) managerCheckbox.setSelected(true);
277.

```

```

278.
279.     //chart
280.     updateChart();
281.
282.     setIconImage(Dialog.createImageIcon(Static.profileIconPath, "Manager").getImage());
283.     setResizable(false);
284.
285.     setSize(690, 750);
286.     setLocationRelativeTo(Dialog.mainWindow); //center
287.     defaultBorder = nameField.getBorder();
288.     setDefaultTitle();
289.
290.     //display
291.     setVisible(true);
292. }
293.
294. /**
295.  * Updates information shown in table.
296.  *
297.  */
298. public void updateTable()
299. {
300.     months = d.months.getAll();
301.     int length = months.length;
302.     //determine header titles
303.
304.
305.     final String[] headerTitles = new String [] {
306.         " ",
307.         "Distributors",
308.         "New dist.",

```

```

309.                                     "New direct dist.",
310.                                     "PW*",
311.                                     "Group PW*",
312.                                     "GW*",
313.                                     "Group GW*",
314.                                     "Provision"
315.                                     };
316. String[][] content = new String [length][9];
317. for (int row=0;row<length;row++)
318. {
319.     for (int col=0;col<9;col++)
320.     {
321.         if (col == 0) //left header
322.         {
323.             content[row][col] = Static.monthByWord(months[length-row-1].getMonth())
324.                               + " "+months[length-row-1].getYear();
325.         }
326.         if (col == 0) //total distributors
327.             content[row][col+1] = ""+months[length-row-1].getNumberOfAllChildren();
328.         else if (col == 1) //new distributors
329.             if (row == length-1)
330.                 content[row][col+1] = ""+months[length-row-1].getNumberOfAllChildren();
331.             else
332.                 content[row][col+1] = ""+(months[length-row-1].getNumberOfAllChildren()
333.                                     -months[length-row-2].getNumberOfAllChildren());
334.         else if (col == 2) //new direct distributors
335.             if (row == length-1)
336.                 content[row][col+1] = ""+months[length-row-1].getNumberOfDirectChildren();
337.             else
338.                 content[row][col+1] = ""+(months[length-row-1].getNumberOfDirectChildren()
339.                                     -months[length-row-2].getNumberOfDirectChildren());

```



```

340.         else if (col == 3) //pw
341.             content[row][col+1] = "+"months[length-row-1].getPw();
342.         else if (col == 4) //group pw
343.             content[row][col+1] = "+"months[length-row-1].getGroupPw();
344.         else if (col == 5) //gw
345.             content[row][col+1] = "+"months[length-row-1].getGw();
346.         else if (col == 6) //group gw
347.             content[row][col+1] = "+"months[length-row-1].getGroupGw();
348.         else if (col == 7) //provision
349.             content[row][col+1] = "+"months[length-row-1].getProvision()+" €";
350.     }
351. }
352.
353. statsTable.setModel(
354.     new NonEditableDefaultTableModel(content, headerTitles)
355.     {
356.         public boolean isCellEditable(int row, int column)
357.         {
358.             //enable editing for pw, group pw, gw and group gw columns
359.             if (column <= 7 && column >= 4) return true;
360.             else return false;
361.         }
362.         public void setValueAt(Object value, int row, int col)
363.         {
364.             //checks if the inserted value to table is numeric
365.             if (Static.isNumeric((String) value))
366.             {
367.                 super.setValueAt(value,row,col);
368.             } else
369.             {
370.                 Dialog.error ("The value inserted has to be numeric!");

```

```

371.         }
372.     }
373. }
374. );
375.
376.
377. statsTable.getModel().addTableModelListener(
378.     new TableModelListener ()
379.     {
380.         public void tableChanged(TableModelEvent e)
381.         {
382.             int row = e.getFirstRow();
383.             int column = e.getColumn();
384.             TableModel model = (TableModel)e.getSource();
385.             String columnName = model.getColumnName(column);
386.             Object data = model.getValueAt(row, column);
387.             Month m = months[months.length-row-1];
388.             if (column == 4) //PW
389.             {
390.                 m.setPw((String)data);
391.                 Dialog.info("PW for "+Static.monthByWord(m.getMonth())
392.                     +" has been successfully updated to "+(String)data+".");
393.             } else if (column == 5) //Group PW
394.             {
395.                 m.setGroupPw((String)data);
396.                 Dialog.info("Group PW for "+Static.monthByWord(m.getMonth())
397.                     +" has been successfully updated to "+(String)data+".");
398.             } else if (column == 6) //GW
399.             {
400.                 m.setGw((String)data);
401.                 Dialog.info("GW for "+Static.monthByWord(m.getMonth())

```

```

402.         +" has been successfully updated to "+(String)data+".";
403.     } else if (column == 7) //Group GW
404.     {
405.         m.setGroupGw((String)data);
406.         Dialog.info("Group GW for "+Static.monthByWord(m.getMonth())
407.             +" has been successfully updated to "+(String)data+".");
408.     }
409.     m.save();
410.     updateTable();
411.     updateChart();
412. }
413. }
414. );
415.
416.
417. //disable left column
418. statsTable.getColumnModel().getColumn(0).setCellRenderer(
419.     new RowHeaderRenderer (statsTable)
420. );
421.
422. //set widths for columns
423. statsTable.getColumnModel().getColumn(0).setPreferredWidth(80);
424. statsTable.getColumnModel().getColumn(1).setPreferredWidth(45);
425. statsTable.getColumnModel().getColumn(2).setPreferredWidth(40);
426. statsTable.getColumnModel().getColumn(3).setPreferredWidth(70);
427. statsTable.getColumnModel().getColumn(4).setPreferredWidth(25);
428. statsTable.getColumnModel().getColumn(5).setPreferredWidth(50);
429. statsTable.getColumnModel().getColumn(6).setPreferredWidth(25);
430. statsTable.getColumnModel().getColumn(7).setPreferredWidth(50);
431. statsTable.getColumnModel().getColumn(8).setPreferredWidth(35);
432.

```

```

433.         //disable dragndrop
434.         statsTable.setDragEnabled(false);
435.         statsTable.getTableHeader().setReorderingAllowed(false);
436.     }
437.
438.     /**
439.      * Updates chart by the newest information from files.
440.      */
441.     public void updateChart()
442.     {
443.         if (chartPanel != null)
444.         {
445.             getContentPane().remove(chartPanel);
446.             chartPanel = null;
447.         }
448.         chartPanel = new ChartPanel(Chart.produce(d.months.getLastMonths(12)));
449.         //chartPanel.setPreferredSize(new Dimension(500, 270));
450.         chartPanel.setBounds(20,450,640,250);
451.         getContentPane().add(chartPanel);
452.         repaint();
453.     }
454.
455.     /**
456.      * Sets the title of the window to current information.
457.      *
458.      */
459.     public void setDefaultTitle()
460.     {
461.         //setting title
462.         String title = "Profile of "+d.getName()+" "+d.getSurname()+
463.             " (" +d.getRegNum();
  
```

```

464.         if (d.isRoot())
465.             title += " - the root distributor>";
466.         else title += " - "+d.getGeneration()+" generation>";
467.
468.         titleLabel.setText(title);
469.         super.setTitle(title);
470.     }
471.
472.     /**
473.      * Defines action when ActionEvent on controls
474.      * at this frame fires
475.      *
476.      * @param y    a sample parameter for a method
477.      * @return     the sum of x and y
478.      */
479.     public void actionPerformed(ActionEvent e)
480.     {
481.         if (e.getSource () == saveBtn)
482.         {
483.             //reset all colors
484.             nameLabel.setForeground(null);
485.             nameField.setBorder(defaultBorder);
486.
487.             surnameLabel.setForeground(null);
488.             surnameField.setBorder(defaultBorder);
489.
490.             emailLabel.setForeground(null);
491.             emailField.setBorder(defaultBorder);
492.
493.             countryLabel.setForeground(null);
494.             countryField.setBorder(defaultBorder);

```

```

495.
496.     registrationDateLabel.setForeground(null);
497.     registrationDateField.setBorder(defaultBorder);
498.
499.
500.     //validation
501.     Vector<String> errorMsgs = new Vector<String>();
502.     MatteBorder border = BorderFactory.createMatteBorder(1, 1, 1, 1, Color.red);
503.
504.     //name
505.     String s = Static.validateString(nameField.getText(),1,true, "Name");
506.     if (s != null)
507.     {
508.         errorMsgs.pushBack(s);
509.         nameLabel.setForeground(Color.red);
510.         nameField.setBorder(border);
511.     }
512.
513.     //surname
514.     s = Static.validateString(surnameField.getText(),1,true, "Surname");
515.     if (s != null)
516.     {
517.         errorMsgs.pushBack(s);
518.         surnameLabel.setForeground(Color.red);
519.         surnameField.setBorder(border);
520.     }
521.
522.     //email
523.     s = Static.validateString(emailField.getText(),2,false, "Email");
524.     if (s != null)
525.     {

```

```

526.         errorMsgs.pushBack(s);
527.         emailLabel.setForeground(Color.red);
528.         emailField.setBorder(border);
529.     }
530.
531.     //country
532.     s = Static.validateString(countryField.getText(),1,true,"Country");
533.     if (s != null)
534.     {
535.         errorMsgs.pushBack(s);
536.         countryLabel.setForeground(Color.red);
537.         countryField.setBorder(border);
538.     }
539.
540.     //registrationDate
541.     s = Static.validateString(registrationDateField.getText(),-1,true,"Registration date");
542.     if (s != null)
543.     {
544.         errorMsgs.pushBack(s);
545.         registrationDateLabel.setForeground(Color.red);
546.         registrationDateField.setBorder(border);
547.     }
548.
549.     String sponsor = "";
550.     if (d.isRoot()) sponsor = "0";
551.     else
552.     {
553.         sponsor = (String) sponsorBox.getSelectedItem();
554.         sponsor = sponsor.substring(0, sponsor.indexOf(" "));
555.     }
556.

```

```

557.     String status = "";
558.     if (consumerCheckbox.isSelected()) status += "C";
559.     if (salesmanCheckbox.isSelected()) status += "S";
560.     if (managerCheckbox.isSelected()) status += "M";
561.     //result
562.     if (errorMsgs.size() == 0)
563.     {
564.         //data is valid
565.         d.setSponsor(sponsor);
566.         d.setCountry(countryField.getText());
567.         d.setRegistrationDate(registrationDateField.getText());
568.         d.setStatus(status);
569.         d.setName(nameField.getText());
570.         d.setSurname(surnameField.getText());
571.         d.setBirthDate(birthDateField.getText());
572.         d.setEmail(emailField.getText());
573.         d.setTelephone(telephoneField.getText());
574.         d.setNote(noteField.getText());
575.         if (d.save())
576.         {
577.             setDefaultTitle();
578.             Dialog.info("Distributor "+d.getName()+" "
579.                 +d.getSurname()+" has been successfully updated!");
580.         }
581.     } else
582.     {
583.         s = "The form is invalid because of following "
584.             +errorMsgs.size()+" reasons:\n";
585.         while (errorMsgs.size() != 0)
586.             s += "- "+errorMsgs.popFront()+"\n";
587.

```



```

588.         Dialog.error (s);
589.     }
590. } else if (e.getSource() == deleteBtn)
591. {
592.     int n = JOptionPane.showConfirmDialog(
593.         this,
594.         "Do you really want to delete "+d.getName()+" "+d.getSurname()+"?"
595.         +"This action will also discard any information about any child\n"
596.         +"distributors associated to this distributor!",
597.         "Delete distributor",
598.         JOptionPane.YES_NO_OPTION);
599.
600.     String name = d.getName()+" "+d.getSurname();
601.     if (n == 0 && d.delete())
602.     {
603.         dispose(); //close window
604.
605.         //update search table
606.         ActionEvent ae = new ActionEvent (Dialog.mainWindow.searchBtn,0,null);
607.         Dialog.mainWindow.actionPerformed(ae);
608.         Dialog.mainWindow.updateCounter();
609.
610.         //inform the user
611.         Dialog.info ("Distributor "+name+" was successfully deleted.");
612.     }
613. } else if (e.getSource() == childTreeBtn)
614. {
615.     if (d.getTree().getNumberOfAllChildren() > 0)
616.         new ChildrenTreeDialog(d, this);
617.     else Dialog.info (d.getFullname()+" does not have any child distributors yet.");
618. } else if (e.getSource() == childTableBtn)

```

```
619.         {
620.             if (d.getTree().getNumberOfAllChildren() > 0)
621.                 new ChildrenTableDialog(d, this);
622.             else Dialog.info (d.getFullname()+" does not have any child distributors yet.");
623.         }
624.     }
```

## File DistributorTreeDialog.java

```

1.  import javax.swing.*;
2.  import java.awt.event.ActionEvent;
3.  import java.awt.event.ActionListener;
4.  import javax.swing.tree.*;
5.  import java.util.Enumeraation;
6.
7.  /**
8.   * Displays JDialog filled with JTree full of distributors.
9.   *
10.   * @author Juraj Masar
11.   * @version 0.1
12.   */
13. public class DistributorTreeDialog extends JDialog implements ActionListener
14. {
15.     String m; //String of given month
16.
17.     //definitions of controls
18.     private JLabel label;
19.     private JScrollPane scrollPane;
20.     private JTree tree;
21.     private JButton continueBtn;
22.     private JButton cancelBtn;
23.     private CheckNode top;
24.
25.     /**
26.      * Constructor for objects of class DistributorTreeDialog.
27.      * Designs and creates the window.
28.      *
29.      * @param m month in String

```

```

30.      */
31.  public DistributorTreeDialog (String m)
32.  {
33.      super (Dialog.mainWindow,"Select distributors", true);
34.
35.      this.m = m;
36.
37.      scrollPane = new JScrollPane();
38.      label = new JLabel();
39.      continueBtn = new  JButton();
40.      cancelBtn = new  JButton();
41.
42.
43.      Thread worker = new Thread()
44.      {
45.          public void run()
46.          {
47.              DistributorManager.init();
48.              Distributor d = DistributorManager.getDistributorBySponsor("0");
49.              top = new CheckNode();
50.
51.              d.getTree().setJTreeNode(top, true);
52.
53.              SwingUtilities.invokeLater(new Runnable() {
54.                  public void run()
55.                  {
56.                      tree = new JTree(top);
57.                      tree.setCellRenderer(new CheckRenderer());
58.
59.                      tree.getSelectionModel().setSelectionMode
60.                          (TreeSelectionModel.SINGLE_TREE_SELECTION);

```

```

61.         tree.addMouseListener(new NodeSelectionListener(tree));
62.         scrollPane.setViewportViewView(tree);
63.     }
64.     });
65. }
66. };
67. worker.start(); // So we don't hold up the dispatch thread.
68.
69. getContentPane().add(scrollPane);
70. scrollPane.setBounds(10, 33, 400, 365);
71.
72. setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
73. getContentPane().setLayout(null);
74.
75.
76. label.setText("Select users for which you want to calculate provision for month "+m+".");
77. getContentPane().add(label);
78. label.setBounds(10, 11, 380, 14);
79.
80. continueBtn.setText("Continue...");
81. getContentPane().add(continueBtn);
82. continueBtn.setBounds(10, 400, 200, 23);
83. continueBtn.addActionListener(this);
84.
85. cancelBtn.setText("Cancel");
86. getContentPane().add(cancelBtn);
87. cancelBtn.setBounds(210, 400, 200, 23);
88. cancelBtn.addActionListener(this);
89.
90.
91. setSize(425,460);

```

```

92.         setResizable(false);
93.         setLocationRelativeTo(Dialog.mainWindow); //center
94.
95.         //display
96.         setVisible(true);
97.     }
98.
99.     /**
100.      * Handles ActionEvents from components form
101.      * DistributorTreeDialog
102.      *
103.      * @param e    ActionEvent object
104.      */
105.     public void actionPerformed(ActionEvent e)
106.     {
107.         if (e.getSource() == continueBtn)
108.         {
109.             Enumeration en = top.breadthFirstEnumeration();
110.             Vector<String> regNums = new Vector<String>();
111.             while (en.hasMoreElements())
112.             {
113.                 CheckNode node = (CheckNode) en.nextElement();
114.                 if (node.isSelected())
115.                 {
116.                     TreeNode[] nodes = node.getPath();
117.                     String s = nodes[nodes.length-1].toString();
118.                     regNums.pushBack(s.substring(0,s.indexOf('(')-1));
119.                 }
120.             }
121.             dispose();
122.

```

```
123.         //start next dialog
124.         new PointsDialog (m, regNums);
125.     } else if (e.getSource() == cancelBtn)
126.     {
127.         dispose();
128.     }
129. }
130.
131. }
132.
```

## File ChangePasswordDialog.java

```
1.  import javax.swing.*;
2.  import java.awt.event.ActionEvent;
3.  import java.awt.event.ActionListener;
4.
5.  /**
6.   * Change password Dialog.
7.   * Customized JDialog.
8.   *
9.   * @author Juraj Masar
10.  * @version 0.1
11.  */
12.
13. public class ChangePasswordDialog extends JDialog implements ActionListener
14. {
15.     //definitions of controls
16.     private JButton cancelBtn;
17.     private JButton changeBtn;
18.     private JPasswordField confirmField;
19.     private JLabel confirmLabel;
20.     private JLabel introLabel;
21.     private JPasswordField oldPasswordField;
22.     private JLabel oldPasswordLabel;
23.     private JPasswordField passwordField;
24.     private JLabel passwordLabel;
25.
26.     /**
27.      * Designs and creates the window.
28.      */
29.     public ChangePasswordDialog(JFrame frame, String title, boolean modal)
```



```
30. {
31.     super (frame, title, modal);
32.     introLabel = new JLabel();
33.     oldPasswordLabel = new JLabel();
34.     passwordLabel = new JLabel();
35.     confirmLabel = new JLabel();
36.     confirmPasswordField = new JPasswordField();
37.     passwordField = new JPasswordField();
38.     oldPasswordField = new JPasswordField();
39.     changeBtn = new JButton();
40.     cancelBtn = new JButton();
41.
42.     setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
43.     getContentPane().setLayout(null);
44.
45.     introLabel.setText("Please, insert the following information:");
46.     getContentPane().add(introLabel);
47.     introLabel.setBounds(10, 11, 190, 14);
48.
49.     oldPasswordLabel.setText("* Old password:");
50.     getContentPane().add(oldPasswordLabel);
51.     oldPasswordLabel.setBounds(10, 34, 80, 14);
52.
53.     passwordLabel.setText("* New password:");
54.     getContentPane().add(passwordLabel);
55.     passwordLabel.setBounds(10, 60, 85, 14);
56.
57.     confirmLabel.setText("* Confirm password:");
58.     getContentPane().add(confirmLabel);
59.     confirmLabel.setBounds(10, 86, 100, 14);
60.
```

```
61.
62.     getContentPane().add(confirmField);
63.     confirmField.setBounds(110, 83, 111, 20);
64.
65.     getContentPane().add(passwordField);
66.     passwordField.setBounds(110, 57, 111, 20);
67.
68.     getContentPane().add(oldPasswordField);
69.     oldPasswordField.setBounds(110, 31, 111, 20);
70.
71.     changeBtn.setText("Change");
72.     getContentPane().add(changeBtn);
73.     changeBtn.setBounds(10, 109, 94, 23);
74.     changeBtn.addActionListener(this);
75.
76.     cancelBtn.setText("Cancel");
77.     getContentPane().add(cancelBtn);
78.     cancelBtn.setBounds(110, 109, 111, 23);
79.     cancelBtn.addActionListener(this);
80.
81.     setSize(231,165);
82.     setResizable(false);
83.     setLocationRelativeTo(Dialog.mainWindow); //center
84.
85.     //display
86.     setVisible(true);
87. }
88.
89. /**
90.  * Defines action when ActionEvent on controls
91.  * at this frame fires
```

```

92.      *
93.      * @param y    a sample parameter for a method
94.      * @return    the sum of x and y
95.      */
96.  public void actionPerformed(ActionEvent e)
97.  {
98.      if (e.getSource () == changeBtn)
99.      {
100.         if (Password.checkPassword(Static.charArrayToString(oldPasswordField.getPassword()))
101.         {
102.             if (Static.charArrayToString(passwordField.getPassword()).equals (
103.                 Static.charArrayToString(confirmField.getPassword()))
104.             {
105.                 Password.setPassword(Static.charArrayToString(passwordField.getPassword()));
106.                 Dialog.info ("New password was successfully saved.");
107.                 dispose();
108.             } else
109.             {
110.                 Dialog.error ("New passwords are not identical.");
111.                 oldPasswordField.setText("");
112.                 passwordField.setText("");
113.                 confirmField.setText("");
114.             }
115.         } else
116.         {
117.             Dialog.error ("The old password inserted is not valid.");
118.             oldPasswordField.setText("");
119.             passwordField.setText("");
120.             confirmField.setText("");
121.         }
122.     } else if (e.getSource () == cancelBtn)

```

```
123.      {  
124.      dispose();  
125.      }  
126.  }  
127. }
```

## File AddDistributorDialog.java

```

1.  import javax.swing.*;
2.  import javax.swing.text.*;
3.  import javax.swing.border.*;
4.
5.  import java.awt.event.ActionEvent;
6.  import java.awt.event.ActionListener;
7.
8.  import java.awt.Color;
9.
10. /**
11.  * Add distributor windows of the application.
12.  * Customized JFrame.
13.  *
14.  * @author Juraj Masar
15.  * @version 0.1
16.  */
17. public class AddDistributorDialog extends JDialog implements ActionListener
18. {
19.     private Border defaultBorder;
20.
21.     //definitions of controls
22.     private JScrollPane scrollPane;
23.     private  JButton  addBtn;
24.     private  JButton  cancelBtn;
25.     private  JLabel  birthDateLabel;
26.     private  JLabel  countryLabel;
27.     private  JLabel  emailLabel;
28.     private  JLabel  headlineLabel;
29.     private  JLabel  nameLabel;

```

```

30.     private JLabel noteLabel;
31.     private JLabel regNumLabel;
32.     private JLabel registrationDateLabel;
33.     private JLabel sponsorLabel;
34.     private JLabel surnameLabel;
35.     private JLabel telephoneLabel;
36.     private JLabel statusLabel;
37.     private JTextFieldMaxLength surnameField;
38.     private JTextFieldMaxLength telephoneField;
39.     private JTextFieldMaxLength regNumField;
40.     private JTextFieldMaxLength nameField;
41.     private JTextFieldMaxLength countryField;
42.     private JTextFieldMaxLength emailField;
43.     private JFormattedTextField registrationDateField;
44.     private JFormattedTextField birthDateField;
45.     private JComboBox sponsorBox;
46.     private JTextPane noteField;
47.     private JCheckBox consumerCheckbox;
48.     private JCheckBox salesmanCheckbox;
49.     private JCheckBox managerCheckbox;
50.
51.     /**
52.      * Designs and creates the window.
53.      */
54.     public AddDistributorDialog(JFrame frame, String title, boolean modal)
55.     {
56.         super (frame, title, modal);
57.         statusLabel = new JLabel();
58.         headlineLabel = new JLabel();
59.         nameLabel = new JLabel();
60.         surnameLabel = new JLabel();

```

```

61.      regNumLabel = new JLabel();
62.      sponsorLabel = new JLabel();
63.      countryLabel = new JLabel();
64.      registrationDateLabel = new JLabel();
65.      birthDateLabel = new JLabel();
66.      emailLabel = new JLabel();
67.      telephoneLabel = new JLabel();
68.      noteLabel = new JLabel();
69.      registrationDateField = new JFormattedTextField();
70.      sponsorBox = new JComboBox();
71.      scrollPane = new JScrollPane();
72.      noteField = new JTextPane();
73.      addBtn = new JButton();
74.      birthDateField = new JFormattedTextField();
75.      cancelBtn = new JButton();
76.      nameField = new JTextFieldMaxLength(20);
77.      surnameField = new JTextFieldMaxLength(20);
78.      emailField = new JTextFieldMaxLength(50);
79.      telephoneField = new JTextFieldMaxLength(20);
80.      regNumField = new JTextFieldMaxLength(10);
81.      countryField = new JTextFieldMaxLength(3);
82.      consumerCheckbox = new JCheckBox();
83.      salesmanCheckbox = new JCheckBox();
84.      managerCheckbox = new JCheckBox();
85.
86.
87.      setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
88.      getContentPane().setLayout(null);
89.
90.      headlineLabel.setFont(new java.awt.Font("Tahoma", 0, 18));
91.      headlineLabel.setText("Add new distributor");
  
```

```
92.         getContentPane().add(headlineLabel);
93.         headlineLabel.setBounds(10, 11, 650, 22);
94.
95.         nameLabel.setText("* Name:");
96.         getContentPane().add(nameLabel);
97.         nameLabel.setBounds(10, 51, 55, 14);
98.
99.         surnameLabel.setText("* Surname:");
100.        getContentPane().add(surnameLabel);
101.        surnameLabel.setBounds(10, 77, 55, 14);
102.
103.        regNumLabel.setText("* Registration number:");
104.        getContentPane().add(regNumLabel);
105.        regNumLabel.setBounds(326, 48, 110, 14);
106.
107.        sponsorLabel.setText("* Sponsor:");
108.        getContentPane().add(sponsorLabel);
109.        sponsorLabel.setBounds(326, 130, 55, 14);
110.
111.        countryLabel.setText("* Country:");
112.        getContentPane().add(countryLabel);
113.        countryLabel.setBounds(326, 74, 55, 14);
114.
115.        registrationDateLabel.setText("* Date of registration:");
116.        getContentPane().add(registrationDateLabel);
117.        registrationDateLabel.setBounds(326, 100, 110, 14);
118.
119.        birthDateLabel.setText("* Date of birth:");
120.        getContentPane().add(birthDateLabel);
121.        birthDateLabel.setBounds(10, 103, 75, 14);
122.
```



```
123.     emailLabel.setText("E-mail:");
124.     getContentPane().add(emailLabel);
125.     emailLabel.setBounds(10, 129, 32, 14);
126.
127.     telephoneLabel.setText("Telephone:");
128.     getContentPane().add(telephoneLabel);
129.     telephoneLabel.setBounds(10, 155, 54, 14);
130.
131.     noteLabel.setText("Note:");
132.     getContentPane().add(noteLabel);
133.     noteLabel.setBounds(10, 178, 27, 14);
134.
135.     getContentPane().add(nameField);
136.     nameField.setBounds(107, 45, 190, 20);
137.
138.     getContentPane().add(surnameField);
139.     surnameField.setBounds(107, 71, 190, 20);
140.
141.     getContentPane().add(emailField);
142.     emailField.setBounds(107, 123, 190, 20);
143.
144.     getContentPane().add(telephoneField);
145.     telephoneField.setBounds(107, 149, 190, 20);
146.
147.     getContentPane().add(regNumField);
148.     regNumField.setBounds(437, 45, 190, 20);
149.
150.     getContentPane().add(countryField);
151.     countryField.setBounds(437, 71, 190, 20);
152.
153.     getContentPane().add(registrationDateField);
```

```

154.      registrationDateField.setBounds(437, 97, 190, 20);
155.      registrationDateField.setFormatterFactory(
156.          new DefaultFormatterFactory(new DateFormatter(
157.              new java.text.SimpleDateFormat("dd.MM.yyyy"))));
158.      registrationDateField.setText("01.01.2011");
159.
160.      String[] sponsors = new String[DistributorManager.getCount()];
161.      ItemRegNum item = (ItemRegNum) DistributorManager.listRegNum.getFirst();
162.      for (int i=0;i<DistributorManager.getCount();i++)
163.      {
164.          sponsors[i] = item.getKey()+" (" +item.treeItem.name+" "+item.treeItem.surname+" )";
165.          item = (ItemRegNum) item.getNext();
166.      }
167.      sponsorBox.setModel(new DefaultComboBoxModel(sponsors));
168.
169.      getContentPane().add(sponsorBox);
170.      sponsorBox.setBounds(437, 128, 190, 20);
171.
172.      scrollPane.setViewportViewView(noteField);
173.
174.      getContentPane().add(scrollPane);
175.      scrollPane.setBounds(107, 176, 190, 51);
176.
177.      addBtn.setText("Add distributor");
178.      getContentPane().add(addBtn);
179.      addBtn.setBounds(326, 176, 301, 23);
180.      addBtn.addActionListener(this);
181.
182.      getContentPane().add(birthDateField);
183.      birthDateField.setBounds(107, 97, 190, 20);
184.      birthDateField.setFormatterFactory(
  
```

```

185.         new DefaultFormatterFactory(new DateFormatter(
186.             new java.text.SimpleDateFormat("dd.MM.yyyy"))));
187. birthDateField.setText("01.01.2011");
188.
189. cancelBtn.setText("Cancel");
190. getContentPane().add(cancelBtn);
191. cancelBtn.setBounds(326, 205, 301, 23);
192. cancelBtn.addActionListener(this);
193.
194. statusLabel.setText("Status:");
195. getContentPane().add(statusLabel);
196. statusLabel.setBounds(326, 152, 35, 14);
197.
198. consumerCheckbox.setText("Consumer");
199. getContentPane().add(consumerCheckbox);
200. consumerCheckbox.setBounds(414, 150, 73, 23);
201.
202. salesmanCheckbox.setText("Salesman");
203. getContentPane().add(salesmanCheckbox);
204. salesmanCheckbox.setBounds(487, 150, 71, 23);
205.
206. managerCheckbox.setText("Manager");
207. getContentPane().add(managerCheckbox);
208. managerCheckbox.setBounds(560, 150, 67, 23);
209.
210. setSize(640, 270);
211. setResizable(false);
212. setLocationRelativeTo(Dialog.mainWindow); //center
213. defaultBorder = nameField.getBorder();
214.
215. //display

```

```

216.         setVisible(true);
217.     }
218.
219.     /**
220.      * Defines action when ActionEvent on controls
221.      * at this frame fires
222.      *
223.      * @param e ActionEvent object
224.      */
225.     public void actionPerformed(ActionEvent e)
226.     {
227.
228.         if (e.getSource () == cancelBtn)
229.         {
230.             dispose();
231.         } else if (e.getSource () == addBtn)
232.         {
233.             //reset all colors
234.             nameLabel.setForeground(null);
235.             nameField.setBorder(defaultBorder);
236.
237.             surnameLabel.setForeground(null);
238.             surnameField.setBorder(defaultBorder);
239.
240.             regNumLabel.setForeground(null);
241.             regNumField.setBorder(defaultBorder);
242.
243.             emailLabel.setForeground(null);
244.             emailField.setBorder(defaultBorder);
245.
246.             countryLabel.setForeground(null);

```

```

247.         countryField.setBorder(defaultBorder);
248.
249.         registrationDateLabel.setForeground(null);
250.         registrationDateField.setBorder(defaultBorder);
251.
252.
253.         //validation
254.         Vector<String> errorMsgs = new Vector<String>();
255.         MatteBorder border = BorderFactory.createMatteBorder(1, 1, 1, 1, Color.red);
256.
257.         //name
258.         String s = Static.validateString(nameField.getText(),1,true,"Name");
259.         if (s != null)
260.         {
261.             errorMsgs.pushBack(s);
262.             nameLabel.setForeground(Color.red);
263.             nameField.setBorder(border);
264.         }
265.
266.         //surname
267.         s = Static.validateString(surnameField.getText(),1,true,"Surname");
268.         if (s != null)
269.         {
270.             errorMsgs.pushBack(s);
271.             surnameLabel.setForeground(Color.red);
272.             surnameField.setBorder(border);
273.         }
274.
275.         //regNum
276.         s = Static.validateString(regNumField.getText(),0,true,"Registration number");
277.         if (s != null)

```

```

278.         {
279.             errorMsgs.pushBack(s);
280.             regNumLabel.setForeground(Color.red);
281.             regNumField.setBorder(border);
282.         }
283.
284.         //email
285.         s = Static.validateString(emailField.getText(),2,false, "Email");
286.         if (s != null)
287.         {
288.             errorMsgs.pushBack(s);
289.             emailLabel.setForeground(Color.red);
290.             emailField.setBorder(border);
291.         }
292.
293.         //country
294.         s = Static.validateString(countryField.getText(),1,true, "Country");
295.         if (s != null)
296.         {
297.             errorMsgs.pushBack(s);
298.             countryLabel.setForeground(Color.red);
299.             countryField.setBorder(border);
300.         }
301.
302.         //registrationDate
303.         s = Static.validateString(registrationDateField.getText(),-1,true, "Registration date");
304.         if (s != null)
305.         {
306.             errorMsgs.pushBack(s);
307.             registrationDateLabel.setForeground(Color.red);
308.             registrationDateField.setBorder(border);

```

```

309.     }
310.
311.     String sponsor = (String) sponsorBox.getSelectedItem();
312.     sponsor = sponsor.substring(0, sponsor.indexOf(" "));
313.
314.     String status = "";
315.     if (consumerCheckbox.isSelected()) status += "C";
316.     if (salesmanCheckbox.isSelected()) status += "S";
317.     if (managerCheckbox.isSelected()) status += "M";
318.     //result
319.     if (errorMsgs.size() == 0)
320.     {
321.         //data is valid
322.         if (DistributorManager.addDistributor(
323.             regNumField.getText(),
324.             sponsor,
325.             countryField.getText(),
326.             registrationDateField.getText(),
327.             status,
328.             nameField.getText(),
329.             surnameField.getText(),
330.             birthDateField.getText(),
331.             emailField.getText(),
332.             telephoneField.getText(),
333.             noteField.getText()
334.         ))
335.         {
336.             Dialog.info("Distributor has been successfully added to the database!");
337.             dispose();
338.         }
339.         else

```

```

340.         {
341.             //such regNum exists!
342.             regNumLabel.setForeground(Color.red);
343.             regNumField.setBorder(border);
344.             Dialog.error("Distributor with identical registration number already\n"
345.                 +"exists in the database. Please, choose different "
346.                 +"registration number!");
347.         }
348.     } else
349.     {
350.         s = "The form is invalid because of following "
351.             +errorMsgs.size()+" reasons:\n";
352.         while (errorMsgs.size() != 0)
353.             s += "- "+errorMsgs.popFront()+"\n";
354.
355.         Dialog.error (s);
356.     }
357. }
358. }
359. }
360.

```



## File PointsDialog.java

```

1.  import javax.swing.*;
2.  import java.awt.event.ActionListener;
3.  import java.awt.event.ActionEvent;
4.  import javax.swing.table.*;
5.
6.  /**
7.   * Customized version of JDialog.
8.   * Displays form implemented using JTable for inserting Distributors' points
9.   *
10.   * @author Juraj Masar
11.   * @version 0.1
12.   */
13. public class PointsDialog extends JDialog implements ActionListener
14. {
15.     Distributor[] distributors;
16.     String monthFormatted;
17.
18.     //definitions of components
19.     private JButton cancelBtn;
20.     private JButton finishBtn;
21.     private JLabel label;
22.     private JScrollPane scrollPane;
23.     private JTable table;
24.
25.     /**
26.      * Constructor for objects of class PointsDialog
27.      * It designs the form.
28.      *
29.      * @param m month in String

```

```

30.      * @param regNums Vector of strings with registration numbers
31.      */
32.  public PointsDialog(String m, final Vector<String> regNums)
33.  {
34.      super (Dialog.mainWindow, "Insert Distributors' points", true);
35.
36.      monthFormatted = m.substring(3,7)+m.substring(0,2);
37.
38.      // initialise instance variables
39.      label = new JLabel();
40.      scrollPane = new JScrollPane();
41.      table = new JTable();
42.      finishBtn = new JButton();
43.      cancelBtn = new JButton();
44.
45.      setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
46.      getContentPane().setLayout(null);
47.
48.      label.setText("Calculation of provisions of selected users for month "+m
49.                  +" requires providing the following information:");
50.      getContentPane().add(label);
51.      label.setBounds(10, 11, 535, 14);
52.
53.      Thread worker = new Thread()
54.      {
55.          public void run()
56.          {
57.              DistributorManager.init();
58.              distributors =
59.                  DistributorManager.getDistributorsWithDirectChildrenByRegNums(regNums);
60.              if (distributors.length == 0)

```

```

61.     {
62.         dispose();
63.         Dialog.warning("You have to choose some distributors. Please, try again.");
64.     }
65.     final String[][] content = new String[distributors.length][5];
66.
67.     for (int i=0;i<distributors.length;i++)
68.     {
69.         Month month = distributors[i].months.getMonth(monthFormatted);
70.         if (month == null)
71.         {
72.             month = new Month(distributors[i]);
73.             month.setPw("0");
74.             month.setGroupPw("0");
75.             month.setGw("0");
76.             month.setGroupGw("0");
77.         }
78.         content[i] = new String[] {distributors[i].getFullname()+" ("
79.             +distributors[i].getRegNum()+")",
80.             month.getPw(),
81.             month.getGroupPw(),
82.             month.getGw(),
83.             month.getGroupGw()
84.         };
85.     }
86.     SwingUtilities.invokeLater(new Runnable() {
87.         public void run()
88.         {
89.             String[] headerTitles = new String [] {
90.                 "Distributor", "PW", "Group PW", "GW", "Group GW"
91.             };

```

```

92.         table.setModel(
93.             new NonEditableDefaultTableModel(content, headerTitles)
94.             {
95.                 public boolean isCellEditable(int row, int column)
96.                 {
97.                     //enable editing for pw, group pw, gw and group gw columns
98.                     if (column >= 1) return true;
99.                     else return false;
100.                 }
101.                 //checks if the inserted value to table is numeric
102.                 public void setValueAt(Object value, int row, int col)
103.                 {
104.                     if (Static.isNumeric((String) value))
105.                     {
106.                         super.setValueAt(value,row,col);
107.                     } else
108.                     {
109.                         Dialog.error ("The value inserted has to be numeric!");
110.                     }
111.                 }
112.             }
113.         );
114.
115.         //disable left column
116.         table.getColumnModel().setCellRenderer(
117.             new RowHeaderRenderer (table)
118.         );
119.
120.         //set widths for columns
121.         table.getColumnModel().getColumn(0).setPreferredWidth(200);
122.

```

```

123.             //disable dragndrop
124.             table.setDragEnabled(false);
125.             table.getTableHeader().setReorderingAllowed(false);
126.
127.         }
128.     });
129. }
130. };
131. worker.start(); // So we don't hold up the dispatch thread.
132.
133. scrollPane.setViewportView(table);
134.
135. getContentPane().add(scrollPane);
136. scrollPane.setBounds(10, 31, 535, 361);
137.
138. finishBtn.setText("Finish");
139. getContentPane().add(finishBtn);
140. finishBtn.setBounds(10, 398, 261, 23);
141. finishBtn.addActionListener(this);
142.
143. cancelBtn.setText("Cancel");
144. cancelBtn.addActionListener(this);
145. getContentPane().add(cancelBtn);
146. cancelBtn.setBounds(277, 398, 268, 23);
147.
148.
149. setSize(560,460);
150. setResizable(false);
151. setLocationRelativeTo(Dialog.mainWindow); //center
152.
153. //display

```

```

154.         setVisible(true);
155.     }
156.
157.     /**
158.      * Listens to ActionEvents within this JDialog.
159.      *
160.      * @param e    ActionEvent object
161.      */
162.     public void actionPerformed(ActionEvent e)
163.     {
164.         if (e.getSource() == cancelBtn)
165.         {
166.             dispose();
167.         } else if (e.getSource() == finishBtn)
168.         {
169.             for (int i=0;i<distributors.length;i++)
170.             {
171.                 Month month = distributors[i].months.getMonth(monthFormatted);
172.                 if (month == null)
173.                 {
174.                     month = new Month(distributors[i]);
175.                     month.setYear(monthFormatted.substring(0,4));
176.                     month.setMonth(monthFormatted.substring(4,6));
177.                 }
178.                 String pw = (String)table.getValueAt(i,1);
179.                 String groupPw = (String)table.getValueAt(i,2);
180.                 String gw = (String)table.getValueAt(i,3);
181.                 String groupGw = (String)table.getValueAt(i,4);
182.
183.                 if (pw == null || pw == "") pw = "0";
184.                 if (groupPw == null || groupPw == "") groupPw = "0";

```

```

185.         if (gw == null || groupGw == "") gw = "0";
186.         if (groupGw == null || groupGw == "") groupGw = "0";
187.
188.         if (!Static.isNumeric(pw)) pw = "0";
189.         if (!Static.isNumeric(groupPw)) groupPw = "0";
190.         if (!Static.isNumeric(gw)) gw = "0";
191.         if (!Static.isNumeric(groupGw)) groupGw = "0";
192.
193.
194.         month.setPw(pw);
195.         month.setGroupPw(groupPw);
196.         month.setGw(gw);
197.         month.setGroupGw(groupGw);
198.         month.save();
199.     }
200.     dispose();
201.     Dialog.info("Data have been successfully inserted to database.");
202. }
203. }
204.
205. }

```

## File ChildrenTreeDialog.java

```
1.  import javax.swing.*;
2.  import javax.swing.event.TreeSelectionEvent;
3.  import javax.swing.event.TreeSelectionListener;
4.  import javax.swing.tree.*;
5.  /**
6.   * Displays JDialog filled with JTree full of children distributors.
7.   *
8.   * @author Juraj Masar
9.   * @version 0.1
10.  */
11. public class ChildrenTreeDialog extends JDialog implements TreeSelectionListener
12. {
13.     Distributor d;
14.     //definitions of controls
15.     private JLabel label;
16.     private JScrollPane scrollPane;
17.     private JTree tree;
18.
19.
20.     /**
21.      * Constructor for objects of class ChildrenTreeDialog.
22.      * Designs and creates the window.
23.      */
24.     public ChildrenTreeDialog (final Distributor d, ProfileWindow profile)
25.     {
26.         super (profile,"Children distributors in tree", true);
27.
28.         this.d = d;
29.
```



```

30.     scrollPane = new JScrollPane();
31.     label = new JLabel();
32.     tree = new JTree();
33.
34.     Thread worker = new Thread()
35.     {
36.         public void run()
37.         {
38.             final DefaultMutableTreeNode top = new DefaultMutableTreeNode();
39.             d.getTree().setJTreeNode(top, true);
40.
41.             SwingUtilities.invokeLater(new Runnable() {
42.                 public void run()
43.                 {
44.                     tree.setModel(new DefaultTreeModel(top));
45.                     //tree.setRootVisible(false);
46.                 }
47.             });
48.         }
49.     };
50.
51.     tree.getSelectionModel().setSelectionMode
52.         (TreeSelectionMode.SINGLE_TREE_SELECTION);
53.
54.
55.     //set icons
56.     DefaultTreeCellRenderer renderer = (DefaultTreeCellRenderer) tree.getCellRenderer();
57.     renderer.setLeafIcon(Dialog.createImageIcon(Static.profileIconSmallPath, "Manager"));
58.     renderer.setOpenIcon(Dialog.createImageIcon(Static.profileIconSmallPath, "Manager"));
59.     renderer.setClosedIcon(Dialog.createImageIcon(Static.profileIconSmallPath, "Manager"));
60.     //renderer.setLeafIcon(null);

```

```

61.         //renderer.setOpenIcon(null);
62.         //renderer.setClosedIcon(null);
63.
64.
65.         //Listen for when the selection changes.
66.         tree.addTreeSelectionListener(this);
67.
68.         scrollPane.setViewportViewView(tree);
69.
70.         getContentPane().add(scrollPane);
71.         scrollPane.setBounds(10, 33, 235, 365);
72.
73.         worker.start(); // So we don't hold up the dispatch thread.
74.
75.
76.
77.
78.         setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
79.         getContentPane().setLayout(null);
80.
81.
82.         label.setText("Click on the distributor to open thier profile.");
83.         getContentPane().add(label);
84.         label.setBounds(10, 11, 225, 14);
85.
86.
87.         setSize(260,440);
88.         setResizable(false);
89.         setLocationRelativeTo(Dialog.mainWindow); //center
90.
91.         //display

```

```

92.         setVisible(true);
93.     }
94.
95.     /**
96.      * Defines action when TreeSelectionEvent on JTree fires
97.      *
98.      * @param e TreeSelectionEvent
99.      */
100.    public void valueChanged(TreeSelectionEvent e)
101.    {
102.        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
103.            tree.getLastSelectedPathComponent();
104.
105.        if (node == null)
106.            //Nothing is selected.
107.            return;
108.
109.        Object nodeInfo = node.getUserObject();
110.        dispose();
111.        Dialog.displayProfile(
112.            ((String)nodeInfo).substring(0, ((String)nodeInfo).indexOf(" "))
113.        );
114.
115.    }
116. }

```

## File ChildrenTableDialog.java

```
1. import javax.swing.*;
2. import java.awt.event.*;
3.
4. /**
5.  * Displays JDialog filled with JTable full of children distributors.
6.  *
7.  * @author Juraj Masar
8.  * @version 0.1
9.  */
10. public class ChildrenTableDialog extends JDialog
11. {
12.     Distributor d;
13.     //definitions of controls\
14.     private JLabel label;
15.     private JScrollPane scrollPane;
16.     private JTable table;
17.
18.     /**
19.      * Constructor for objects of class ChildrenTableDialog.
20.      * Designs and creates the window.
21.      */
22.     public ChildrenTableDialog (final Distributor d, ProfileWindow profile)
23.     {
24.         super (profile, "Children distributors of "+d.getFullname()+" in table", true);
25.
26.         this.d = d;
27.
28.         scrollPane = new JScrollPane();
29.         label = new JLabel();
```

```

30.     table = new JTable();
31.
32.     //disable dragndrop
33.     table.setDragEnabled(false);
34.     table.getTableHeader().setReorderingAllowed(false);
35.
36.     table.addMouseListener(new MouseAdapter()
37.     {
38.         public void mouseClicked(MouseEvent e)
39.         {
40.             if (e.getClickCount() == 2)
41.             {
42.                 JTable target = (JTable)e.getSource();
43.                 int row = target.getSelectedRow();
44.
45.                 String regNum = (String)table.getValueAt(row, 0);
46.                 dispose();
47.                 Dialog.displayProfile(regNum);
48.             }
49.         }
50.     });
51.
52.
53.     Thread worker = new Thread()
54.     {
55.         public void run()
56.         {
57.             Distributor[] distributors = d.getAllChildrenDistributors();
58.             final Object[][] objects = new Object[distributors.length][9];
59.
60.             for (int i=0;i<distributors.length;i++)

```

```

61.      {
62.          objects[i][0] = distributors[i].getRegNum();
63.          objects[i][1] = distributors[i].getName();
64.          objects[i][2] = distributors[i].getSurname();
65.          objects[i][3] = distributors[i].getEmail();
66.          objects[i][4] = distributors[i].getCountry();
67.          objects[i][5] = distributors[i].getStatus();
68.          objects[i][6] = distributors[i].getRegistrationDate();
69.          objects[i][7] = distributors[i].getTelephone();
70.          objects[i][8] = distributors[i].getNote();
71.      }
72.
73.      SwingUtilities.invokeLater(new Runnable() {
74.          public void run()
75.          {
76.              //insert new data
77.              String[] tableHeader = new String [] {
78.                  "Reg. num.", "Name", "Surname", "E-mail", "Country",
79.                  "Status", "Reg. date", "Telephone", "Note"
80.              };
81.              table.setModel(
82.                  new NonEditableDefaultTableModel(objects, tableHeader)
83.              );
84.
85.              //set email column width
86.              table.getColumnModel().getColumn(0).setPreferredWidth(25);
87.              //set email column width
88.              table.getColumnModel().getColumn(3).setPreferredWidth(100);
89.              //set country column width
90.              table.getColumnModel().getColumn(4).setPreferredWidth(25);
91.              //set status column width

```

```

92.             table.getColumnModel().getColumn(5).setPreferredWidth(25);
93.         }
94.     });
95. }
96. };
97. worker.start(); // So we don't hold up the dispatch thread.
98.
99. scrollPane.setViewportView(table);
100.
101. getContentPane().add(scrollPane);
102. scrollPane.setBounds(10, 31, 837, 451);
103.
104. setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
105. getContentPane().setLayout(null);
106.
107.
108. label.setText("Doubleclick on the distributor to open thier profile.");
109. getContentPane().add(label);
110. label.setBounds(10, 11, 240, 14);
111.
112.
113. setSize(865,520);
114. setResizable(false);
115. setLocationRelativeTo(Dialog.mainWindow); //center
116.
117. //display
118. setVisible(true);
119. }
120. }

```

## File CheckNode.java

```
1.  import javax.swing.tree.DefaultMutableTreeNode;
2.
3.  /**
4.   * Customizes version of DefaultMutableTreeNode
5.   * for Checkbox implementation to JTree.
6.   *
7.   * Inspired by:
8.   * http://www.crionics.com/products/opensource/faq/swing\_ex/SwingExamples.html
9.   *
10.  * @author Juraj Masar
11.  * @version 0.1
12.  */
13. class CheckNode extends DefaultMutableTreeNode
14. {
15.     public final static int SINGLE_SELECTION = 0;
16.
17.     public final static int DIG_IN_SELECTION = 4;
18.
19.     protected int selectionMode;
20.
21.     public boolean isSelected;
22.
23.     /**
24.      * Constructor of the object.
25.      * Sets the userObject to null
26.      */
27.     public CheckNode()
28.     {
29.         this(null);
```

312



```
30.  }
31.
32.  /**
33.   * Constructor of the object.
34.   * Sets the userObject to value given
35.   *
36.   * @param userObject Object
37.   */
38.  public CheckNode(Object userObject)
39.  {
40.      this(userObject, true, false);
41.  }
42.
43.  /**
44.   * Constructor of the object.
45.   * Sets the object variables to values given.
46.   *
47.   * @param userObject Object
48.   * @param allowsChildren boolean
49.   * @param isSelected boolean
50.   */
51.  public CheckNode(
52.      Object userObject,
53.      boolean allowsChildren,
54.      boolean isSelected
55.  ) {
56.      super(userObject, allowsChildren);
57.      this.isSelected = isSelected;
58.      setSelectionMode(DIG_IN_SELECTION);
59.  }
60.
```

```

61.  /**
62.   * Sets selection mode of the node to value given.
63.   *
64.   * @param mode selection mode
65.   */
66.  public void setSelectionMode(int mode)
67.  {
68.      selectionMode = mode;
69.  }
70.
71.  /**
72.   * Returns selection mode of the node.
73.   *
74.   * @return selectionMode
75.   */
76.  public int getSelectionMode()
77.  {
78.      return selectionMode;
79.  }
80.
81.  /**
82.   * Sets the node isSelected value to value given.
83.   *
84.   * @param isSelected whether the object is selected or not
85.   */
86.  public void setSelected(boolean isSelected)
87.  {
88.      this.isSelected = isSelected;
89.  }
90.
91.  /**

```

```
92.    * Returns the isSelected value of the object.
93.    *
94.    * @return isSelected boolean
95.    */
96.    public boolean isSelected()
97.    {
98.        return isSelected;
99.    }
100. }
101.
```

## File CheckRenderer.java

```
1. import java.awt.Color;
2. import java.awt.Component;
3. import javax.swing.UIManager;
4. import javax.swing.plaf.ColorUIResource;
5. import javax.swing.JCheckBox;
6. import javax.swing.JPanel;
7. import javax.swing.JTree;
8. import java.awt.Dimension;
9. import javax.swing.tree.TreeCellRenderer;
10.
11. /**
12.  * CheckRenderer is an enhanced version of TableCellRenderer meant to be used
13.  * in JTree with implemented checkboxes.
14.  *
15.  * Inspired by:
16.  * http://www.crionics.com/products/opensource/faq/swing\_ex/SwingExamples.html
17.  *
18.  * @author Juraj Masar
19.  * @version 0.1
20.  */
21. class CheckRenderer extends JPanel implements TreeCellRenderer
22. {
23.     protected JCheckBox check;
24.
25.     protected TreeLabel label;
26.
27.     /**
28.      * Constructor of the object. Sets the object variables
29.      * to default values.
```

```

30.    */
31.    public CheckRenderer()
32.    {
33.        setLayout(null);
34.        add(check = new JCheckBox());
35.        add(label = new TreeLabel());
36.        check.setBackground(UIManager.getColor("Tree.textBackground"));
37.        label.setForeground(UIManager.getColor("Tree.textForeground"));
38.    }
39.
40.    /**
41.     * Returns the instance of the CheckRender class with
42.     * object variables set to values given in parameters.
43.     *
44.     * @param tree      JTree object
45.     * @param value      Object instance
46.     * @param isSelected whether the option is selected or not
47.     * @param expanded  whether the path to option is expanded or not
48.     * @param leaf       whether the option is leaf or not
49.     * @param row        the row in JTree object
50.     * @param hasFocus   whether the option is focused or not
51.     * @return           Component object
52.     */
53.    public Component getTreeCellRendererComponent(
54.        JTree tree,
55.        Object value,
56.        boolean isSelected,
57.        boolean expanded,
58.        boolean leaf,
59.        int row,
60.        boolean hasFocus

```

```

61. ) {
62.     String stringValue = tree.convertValueToText(value, isSelected,
63.         expanded, leaf, row, hasFocus);
64.     setEnabled(tree.isEnabled());
65.     check.setSelected(((CheckNode) value).isSelected());
66.     label.setFont(tree.getFont());
67.     label.setText(stringValue);
68.     label.setSelected(isSelected);
69.     label.setFocus(hasFocus);
70.
71.     //set icon
72.     label.setIcon(Dialog.createImageIcon(Static.profileIconSmallPath, "Manager"));
73.
74.     return this;
75. }
76. /**
77.  * Generates preferred size of the whole component.
78.  *
79.  * @return Dimension object
80.  */
81. public Dimension getPreferredSize()
82. {
83.     Dimension d_check = check.getPreferredSize();
84.     Dimension d_label = label.getPreferredSize();
85.     return new Dimension(d_check.width + d_label.width,
86.         (d_check.height < d_label.height ? d_label.height
87.             : d_check.height));
88. }
89.
90. /**
91.  * Sets the locations of components of which this

```

```

92.     * Component consists.
93.     */
94.     public void doLayout()
95.     {
96.         Dimension d_check = check.getPreferredSize();
97.         Dimension d_label = label.getPreferredSize();
98.         int y_check = 0;
99.         int y_label = 0;
100.         if (d_check.height < d_label.height)
101.             y_check = (d_label.height - d_check.height) / 2;
102.         else
103.             y_label = (d_check.height - d_label.height) / 2;
104.
105.         check.setLocation(0, y_check-2);
106.         check.setBounds(0, y_check-2, d_check.width, d_check.height);
107.         label.setLocation(d_check.width, y_label);
108.         label.setBounds(d_check.width, y_label, d_label.width, d_label.height);
109.     }
110.     /**
111.      * Sets the background of the component to
112.      * Color given in parameters
113.      *
114.      * @param Color Color object
115.      * */
116.     public void setBackground(Color color)
117.     {
118.         if (color instanceof ColorUIResource)
119.             color = null;
120.         super.setBackground(color);
121.     }
122. }

```

## File NodeSelectionListener.java

```

1. import java.awt.event.MouseAdapter;
2. import java.awt.event.MouseEvent;
3. import javax.swing.JTree;
4. import javax.swing.tree.TreePath;
5. import javax.swing.tree.DefaultTreeModel;
6. /**
7.  * NodeSelectionListener is an enhanced version of MouseAdapter meant to be used
8.  * in JTree with implemented checkboxes.
9.  *
10. * Inspired by:
11. * http://www.crionics.com/products/opensource/faq/swing\_ex/SwingExamples.html
12. *
13. * @author Juraj Masar
14. * @version 0.1
15. */
16. class NodeSelectionListener extends MouseAdapter
17. {
18.     JTree tree;
19.     /**
20.      * Constructor - sets the pointer to JTree
21.      * to JTree given in parameters.
22.      *
23.      * @param tree JTree instance
24.      */
25.     NodeSelectionListener(JTree tree)
26.     {
27.         this.tree = tree;
28.     }
29.

```



```

30.  /**
31.   * Catches the MouseEvent object and produces
32.   * appropriate action.
33.   *
34.   * @param e MouseEvent instance
35.   */
36.  public void mouseClicked(MouseEvent e)
37.  {
38.      int x = e.getX();
39.      int y = e.getY();
40.      int row = tree.getRowForLocation(x, y);
41.      TreePath path = tree.getPathForRow(row);
42.      //TreePath path = tree.getSelectionPath();
43.      if (path != null)
44.      {
45.          CheckNode node = (CheckNode)path.getLastPathComponent();
46.          boolean isSelected = ! (node.isSelected());
47.          node.setSelected(isSelected);
48.          if (node.getSelectionMode() == CheckNode.DIG_IN_SELECTION && isSelected)
49.              tree.expandPath(path);
50.
51.          ((DefaultTreeModel) tree.getModel()).nodeChanged(node);
52.          // I need revalidate if node is root. but why?
53.          if (row == 0)
54.          {
55.              tree.revalidate();
56.              tree.repaint();
57.          }
58.      }
59.  }
60. }

```

## File TreeLabel.java

```
1.  import javax.swing.JLabel;
2.  import javax.swing.UIManager;
3.  import java.awt.Dimension;
4.  import java.awt.Graphics;
5.  import java.awt.Color;
6.  import javax.swing.plaf.ColorUIResource;
7.  import javax.swing.Icon;
8.  /**
9.   * TreeLabel is an enhanced version of JLabel meant to be used
10.  * in JTree with implemented checkboxes.
11.  *
12.  * Inspired by:
13.  * http://www.crionics.com/products/opensource/faq/swing\_ex/SwingExamples.html
14.  *
15.  * @author JuraJ Masar
16.  * @version 0.1
17.  */
18. class TreeLabel extends JLabel
19. {
20.     boolean isSelected;
21.
22.     boolean hasFocus;
23.
24.     /**
25.      * Constructor of the class.
26.      * It sets the local variables to default values.
27.      */
28.     public TreeLabel()
29.     {
```

```

30.         isSelected = false;
31.         hasFocus = false;
32.     }
33.
34.     /**
35.      * Sets the background color to Color given
36.      * in parameters.
37.      *
38.      * @param Color Color object
39.      */
40.     public void setBackground(Color color)
41.     {
42.         if (color instanceof ColorUIResource)
43.             color = null;
44.         super.setBackground(color);
45.     }
46.
47.     /**
48.      * Paints the object to canvas given in parameters
49.      *
50.      * @param g Graphics object
51.      */
52.     public void paint(Graphics g)
53.     {
54.         String str;
55.         if ((str = getText()) != null)
56.         {
57.             if (0 < str.length())
58.             {
59.                 if (isSelected)
60.                 {

```

```

61.         g.setColor(UIManager
62.             .getColor("Tree.selectionBackground"));
63.     } else
64.         g.setColor(UIManager.getColor("Tree.textBackground"));
65.
66.     Dimension d = getPreferredSize();
67.     int imageOffset = 0;
68.     Icon currentI = getIcon();
69.     if (currentI != null)
70.         imageOffset = currentI.getIconWidth()
71.             + Math.max(0, getIconTextGap() - 1);
72.
73.     g.fillRect(imageOffset, 0, d.width - 1 - imageOffset,
74.         d.height);
75.     if (hasFocus)
76.     {
77.         g.setColor(UIManager
78.             .getColor("Tree.selectionBorderColor"));
79.         g.drawRect(imageOffset, 0, d.width - 1 - imageOffset,
80.             d.height - 1);
81.     }
82. }
83. }
84. super.paint(g);
85. }
86.
87. /**
88.  * Generates preferred size for the object according
89.  * to typical JLabel preferred size.
90.  *
91.  * @return Dimension object

```

```
92.     */
93. public Dimension getPreferredSize()
94. {
95.     Dimension retDimension = super.getPreferredSize();
96.     if (retDimension != null)
97.         retDimension = new Dimension(retDimension.width + 3,
98.             retDimension.height);
99.     return retDimension;
100. }
101.
102.
103. /**
104.  * Sets the object selected value to value
105.  * given in parameters.
106.  *
107.  * @param isSelected boolean
108.  */
109. public void setSelected(boolean isSelected)
110. {
111.     this.isSelected = isSelected;
112. }
113.
114. /**
115.  * Sets the object hasFocus value to value
116.  * given in parameters.
117.  *
118.  * @param hasFocus boolean
119.  */
120. public void setFocus(boolean hasFocus)
121. {
122.     this.hasFocus = hasFocus;
```

```
123.      }  
124.    }
```

## File RowHeaderRenderer.java

```
1. import javax.swing.*;
2. import javax.swing.table.*;
3. import java.awt.Component;
4.
5. /**
6.  * Enhances the TableCellRenderer to make the left header in JTable possible.
7.  *
8.  * Originally from http://www.esus.com/docs/GetQuestionPage.jsp?uid=1276
9.  *
10. * @author Juraj Masar
11. * @version 0.1
12. */
13. class RowHeaderRenderer extends JLabel implements TableCellRenderer
14. {
15.     /**
16.      * Sets the properties of the TableCellRenderer to wished
17.      *
18.      * @param table JTable
19.      */
20.     RowHeaderRenderer(JTable table)
21.     {
22.         JTableHeader header = table.getTableHeader();
23.         setOpaque(true);
24.         setBorder(table.getTableHeader().getBorder());
25.         setHorizontalAlignment(LEFT);
26.         setForeground(header.getForeground());
27.         setBackground(header.getBackground());
28.         setFont(header.getFont());
29.         setToolTipText(getText());
```

```
30.     }
31.     /**
32.      * Renders cells
33.      *
34.      * @param table JTable instance
35.      * @param value  Object instance
36.      * @param isSelected  boolean
37.      * @param hasFocus  boolean
38.      * @param row
39.      * @param column
40.      */
41.     public Component getTableCellRendererComponent(JTable table, Object value,
42.         boolean isSelected, boolean hasFocus, int row, int column)
43.     {
44.         setText((value == null) ? "" : value.toString());
45.         return this;
46.     }
47. }
```



## File NonEditableDefaultTableModel.java

```
1.  import java.awt.event.*;
2.  import java.awt.Point;
3.
4.  /*****
5.   * Copyright (c) 1998, 2009 Oracle. All rights reserved.
6.   * This program and the accompanying materials are made available under the
7.   * terms of the Eclipse Public License v1.0 and Eclipse Distribution License v. 1.0
8.   * which accompanies this distribution.
9.   * The Eclipse Public License is available at http://www.eclipse.org/legal/epl-v10.html
10.  * and the Eclipse Distribution License is available at
11.  * http://www.eclipse.org/org/documents/edl-v10.php.
12.  *
13.  * Contributors:
14.  *      Oracle - initial API and implementation from Oracle TopLink
15.  *****/
16.
17.
18. public class NonEditableDefaultTableModel extends javax.swing.table.DefaultTableModel {
19.
20.     public NonEditableDefaultTableModel() {
21.         super();
22.     }
23.
24.     public NonEditableDefaultTableModel(java.lang.Object[][] data, java.lang.Object[] columnNames) {
25.         super(data, columnNames);
26.     }
27.
28.     public NonEditableDefaultTableModel(java.lang.Object[] columnNames, int numRows) {
29.         super(columnNames, numRows);
```

```

30.     }
31.
32.     public NonEditableDefaultTableModel(int numRows, int numColumns) {
33.         super(numRows, numColumns);
34.     }
35.
36.     public NonEditableDefaultTableModel(java.util.Vector columnNames, int numRows) {
37.         super(columnNames, numRows);
38.     }
39.
40.     public NonEditableDefaultTableModel(java.util.Vector data, java.util.Vector columnNames) {
41.         super(data, columnNames);
42.     }
43.
44.     public void moveColumn ()
45.     {
46.
47.     }
48.
49.     /**
50.      * Always return false
51.      */
52.     public boolean isCellEditable(int row, int column)
53.     {
54.         return false;
55.     }
56. }

```

## File Chart.java

```
1. import java.awt.Color;
2. import org.jfree.chart.ChartFactory;
3. import org.jfree.chart.JFreeChart;
4. import org.jfree.chart.axis.NumberAxis;
5. import org.jfree.chart.plot.CategoryPlot;
6. import org.jfree.chart.plot.PlotOrientation;
7. import org.jfree.chart.plot.DatasetRenderingOrder;
8. import org.jfree.chart.renderer.category.LineAndShapeRenderer;
9. import org.jfree.data.category.CategoryDataset;
10. import org.jfree.data.category.DefaultCategoryDataset;
11. import org.jfree.chart.axis.CategoryAxis;
12. import org.jfree.chart.axis.Axis;
13. import org.jfree.chart.axis.AxisLocation;
14. import org.jfree.chart.axis.CategoryLabelPositions;
15. import org.jfree.chart.title.LegendTitle;
16. import org.jfree.chart.labels.StandardCategoryToolTipGenerator;
17. /** DOSSIER: HL mastery 16 - use of additional libraries */
18. /**
19.  * Designes Chart object to be used in Profile Window.
20.  *
21.  * @author Juraj Masar
22.  * @version 0.1
23.  */
24. public class Chart
25. {
26.     /**
27.      * Produces chart filled by information given in parameters.
28.      *
29.      * @param months array with data
```

331

```

30.      * @return JFreeChart object
31.      */
32.  public static JFreeChart produce (Month[] months)
33.  {
34.      //design
35.      JFreeChart chart = ChartFactory.createLineChart(
36.          "", // chart title
37.          "", // domain axis label
38.          "Points", // range axis label
39.          null, // data
40.          PlotOrientation.VERTICAL, // orientation
41.          true, // include legend
42.          true, // tooltips
43.          false // urls
44.      );
45.      //set native background color
46.      chart.setBackgroundPaint(Dialog.mainWindow.getBackground());
47.
48.      CategoryPlot plot = (CategoryPlot) chart.getPlot();
49.
50.      //set background color for plot
51.      plot.setBackgroundPaint(Color.lightGray);
52.      plot.setRangeGridlinePaint(Color.white);
53.
54.      //set axis
55.      NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
56.      rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());
57.      try
58.      {
59.          NumberAxis rangeAxis2 = (NumberAxis)rangeAxis.clone();
60.          rangeAxis2.setLabel ("Distributors");

```

```

61.         plot.setRangeAxis(1,rangeAxis2);
62.         plot.setRangeAxisLocation(1, AxisLocation.BOTTOM_OR_RIGHT);
63.
64.         NumberAxis rangeAxis3 = (NumberAxis)rangeAxis.clone();
65.         rangeAxis3.setLabel ("Provision");
66.         plot.setRangeAxis(2,rangeAxis3);
67.         plot.setRangeAxisLocation(2, AxisLocation.TOP_OR_LEFT);
68.
69.     } catch (java.lang.CloneNotSupportedException e) {}
70.
71.     Axis axis = plot.getDomainAxis();
72.     axis.setTickMarksVisible(false);
73.     ((CategoryAxis)axis).setCategoryLabelPositions(CategoryLabelPositions.UP_45);
74.
75.     //set renderer
76.     LineAndShapeRenderer renderer
77.     = (LineAndShapeRenderer) plot.getRenderer();
78.     renderer.setShapesVisible(true);
79.     renderer.setDrawOutlines(true);
80.     renderer.setUseFillPaint(true);
81.
82.     //populate with data
83.     DefaultCategoryDataset disDataset = new DefaultCategoryDataset();
84.     DefaultCategoryDataset pwDataset = new DefaultCategoryDataset();
85.     DefaultCategoryDataset provisionDataset = new DefaultCategoryDataset();
86.     for (int i=months.length-1;i>=0;i--)
87.     {
88.         pwDataset.addValue(
89.             months[i].getPwInt(),"PW",months[i].getMonth()+"/"+months[i].getYear()
90.         );
91.         pwDataset.addValue(

```

```

92.         months[i].getGwInt(), "GW", months[i].getMonth()+"/"+months[i].getYear()
93.     );
94.     pwDataset.addValue(
95.         months[i].getGroupPwInt(), "Group PW", months[i].getMonth()+"/"+months[i].getYear());
96.     pwDataset.addValue(
97.         months[i].getGroupGwInt(), "Group GW", months[i].getMonth()+"/"+months[i].getYear());
98.
99.     disDataset.addValue(
100.         months[i].getNumberOfAllChildren(),
101.         "All distributors",
102.         months[i].getMonth()+"/"+months[i].getYear()
103.     );
104.     disDataset.addValue(
105.         months[i].getNumberOfDirectChildren(),
106.         "Direct distributors",
107.         months[i].getMonth()+"/"+months[i].getYear()
108.     );
109.
110.     provisionDataset.addValue(
111.         months[i].getProvision(),
112.         "Provision",
113.         months[i].getMonth()+"/"+months[i].getYear()
114.     );
115. }
116.
117. //pin datasets to axes
118. plot.setDataset(0, pwDataset);
119. plot.setDataset(1, disDataset);
120. plot.setDataset(2, provisionDataset);
121. plot.mapDatasetToRangeAxis(1,1);
122. plot.mapDatasetToRangeAxis(2,2);

```

```
123.  
124.     LineAndShapeRenderer renderer1 = new LineAndShapeRenderer();  
125.     renderer1.setBaseToolTipGenerator(  
126.         new StandardCategoryToolTipGenerator());  
127.     renderer1.setShapesVisible(true);  
128.     renderer1.setDrawOutlines(true);  
129.     renderer1.setUseFillPaint(true);  
130.     plot.setRenderer(1,renderer1);  
131.  
132.     LineAndShapeRenderer renderer2 = new LineAndShapeRenderer();  
133.     renderer2.setBaseToolTipGenerator(  
134.         new StandardCategoryToolTipGenerator());  
135.     renderer2.setShapesVisible(true);  
136.     renderer2.setDrawOutlines(true);  
137.     renderer2.setUseFillPaint(true);  
138.     plot.setRenderer(2,renderer2);  
139.  
140.     return chart;  
141. }  
142. }
```

## File PlainDocumentMaxLength.java

```

1. import javax.swing.text.AttributeSet;
2. import javax.swing.text.BadLocationException;
3. import java.awt.Toolkit;
4. import javax.swing.text.PlainDocument;
5.
6. /**
7.  * JTextFieldMaxLength enhances JTextField and
8.  * adds maxLength functionality.
9.  *
10. * Originally from:
11. * http://www.java2s.com/Code/Java/Swing-JFC/JTextFieldMaxLength.htm
12. *
13. * @author Juraj Masar
14. * @version 0.1
15. */
16.
17. class PlainDocumentMaxLength extends PlainDocument
18. {
19.     private int maxLength; //holds the maximum length of content
20.
21.     /**
22.      * Constructor of the class, it sets the object variable
23.      * maxLength to the value given in parameters.
24.      *
25.      * @param maxLength maximum length of the content
26.      */
27.     public PlainDocumentMaxLength(int maxLength)
28.     {
29.         this.maxLength = maxLength;

```



```

30.  }
31.
32.  /**
33.   * Inserts String to the content - it checks
34.   * whether the total length of the content
35.   * does not exceed the maximum length set.
36.   * If it does, it produces beeping.
37.   *
38.   * @param offset offset of insertion
39.   * @param str String to insert
40.   * @param a AttributeSet object
41.   */
42.  public void insertString (
43.      int offset,
44.      String str,
45.      AttributeSet a
46.  ) throws BadLocationException
47.  {
48.      if (getLength() + str.length() > maxLength)
49.          Toolkit.getDefaultToolkit().beep();
50.      else
51.          super.insertString(offset, str, a);
52.  }
53. }

```

## File JTextFieldMaxLength.java

```
1. import javax.swing.JTextField;
2.
3. /**
4.  * JTextFieldMaxLength enhances JTextField and
5.  * adds maxLength functionality.
6.  *
7.  * Originally from:
8.  * http://www.java2s.com/Code/Java/Swing-JFC/JTextFieldMaxLength.htm
9.  *
10. * @author Juraj Masar
11. * @version 0.1
12. */
13. public class JTextFieldMaxLength extends JTextField
14. {
15.     /**
16.      * Constructor of the class. It initializes object variables
17.      * to length given.
18.      *
19.      * @param length maximum length of the content
20.      */
21.     public JTextFieldMaxLength(int length)
22.     {
23.         this(null, length);
24.     }
25.
26.     /**
27.      * Constructor of the class. It initializes the text
28.      * and length variables to values given in parameters.
29.      *
```

```
30.     * @param text content
31.     * @param length maximum length of the content
32.     */
33.     public JTextFieldMaxLength(String text, int length)
34.     {
35.         super(new PlainDocumentMaxLength(length),text,length);
36.     }
37. }
```

## File CustomPasswordField.java

```
1.  import javax.swing.JPasswordField;
2.
3.  /**
4.   * Creates PasswordField which generates Tab+Enter when Enter is pressed.
5.   * Useful for JOptionPane.showOptionDialog
6.   *
7.   * @author tommylee
8.   * @version 0.1
9.   *
10.  * Originally from:
11.  * http://forums.devshed.com/java-help-9/
12.  * is-there-a-password-field-option-for-joptionpane-showinputdialog-566325.html
13.  */
14.  class CustomPasswordField extends JPasswordField
15.  {
16.      /**
17.       * Constructor - enhances the JPasswordField
18.       */
19.      public CustomPasswordField()
20.      {
21.          addKeyListener(new java.awt.event.KeyAdapter()
22.          {
23.              @Override
24.              public void keyPressed(java.awt.event.KeyEvent kEvt)
25.              {
26.                  if (kEvt.getKeyCode() == java.awt.event.KeyEvent.VK_ENTER)
27.                  {
28.                      kEvt.consume();
```

```

29.         // auto generate TAB + Enter keypress events
30.         try
31.         {
32.             java.awt.Robot robot = new java.awt.Robot();
33.             robot.setAutoDelay(100);
34.             robot.keyPress(java.awt.event.KeyEvent.VK_TAB);
35.             robot.keyPress(java.awt.event.KeyEvent.VK_SPACE);
36.             robot.keyRelease(java.awt.event.KeyEvent.VK_SPACE);
37.         }
38.         catch (java.awt.AWTException awtEx) {
39.             awtEx.printStackTrace();
40.         }
41.     }
42. }
43. });
44. }
45. }

```

341

## C: Usability

Feature	Evidence
Compulsory fields in data input are easy to recognize (they are marked with *)	Figures 28,38
User is noticed by dialog when compulsory fields are not fulfilled	Figure 29
User is noticed by dialog when data inserted has wrong format	Figure 35
Input fields with problems are easily distinguishable (they have red border)	Figure 30
User is noticed by dialog when changes in data has been done	Figure 32
In table editable and non-editable rows are easy to distinguish (editable are marked with *)	Figures 28,38
Information is editable directly where it is displayed (e. g. in profile window)	Figure 38
Data are presented not only in text format, but also graphically (chart)	Figures 38, 42
Searching for users can be done by combinations of various criteria (individual criteria does not have to be necessarily precise – they can be only parts of searched information)	Figure 23
Children distributors are presented in various forms – in table, but also in JTree which shows the hierarchy of Distributors	Figures 40, 41
Selection of distributors is implemented via customized checkboxes in JTree	Figures 34
Options in data input are provided whenever possible instead of text input (e. g. selection of sponsors)	Figures 28,38
If more erroneous fields are present in data input, only one message with all problems is displayed	Figure 29

## C2: Handling errors

The application takes care of two types of user generated errors which generally occur:

- Missing files
- Erroneous/invalid data input

### Missing files

The program data are stored in *data/distributors.dat* and in individual files for each distributor in format *data/dist{registration number}.dat*.

If the main file *distributors.dat* is missing, the program assumes it has never been started before. In order to allow successful operation, it runs the first launch process, which – after gaining the

permission and initial information from user – creates new *distributors.dat* with default information and launches the program. (Figures 17-18 )

If individual distributor file is missing, program just automatically creates particular file with default values, enabling the successful operation of other features of the program.

### Erroneous/invalid data input

Proper validation of data input is provided everywhere. The validation process generally consists of following phases.

In **information phase** the user is able to see which data is compulsory and which is not. Compulsory data typically are marked with \*. (Figures 28,38)

During the **data entering phase** certain fields do not allow to input all information. If the range of values possible in particular field is limited (e. g. Selecting sponsor – an existing distributor from the system), options – for instance in form of pull down selection – are provided. If the value has to be inserted in particular format (e. g. date of birth), the required format is prefilled and the content of the field does not change until required format is inserted<sup>6</sup>.

In the **submitting phase**, the format of all data is checked. If any compulsory fields are empty, message is produced. If fields have wrong format messages are produced, too. All problematic messages are gathered and displayed in one single dialog. (Figure 29)

In the **post-submitting phase** if the dialog has not been successfully closed, problematic fields are marked. Typically the label marking the field is colored in red and borders of the problematic fields are market red, too. (Figure 30)

## C3: Success of the program

#	Criterion	Implementation	Evidence
1.	Program should intuitive and suitable for using by non-skilled PC user. It works on Microsoft Windows XP and newer.	The program does work on Microsoft Windows XP and newer. The program interface has been designed according to user's wants and therefore, it fits her needs.	All screenshots in the D1 section.
2.	Program is supposed to work with private data – the access to all information should be protected	The application is password protected. Data stored by application is encrypted.	D1: Figure 19 – login screen, Figure 43 - excerpt from distributors.dat file
3.	Inserting new distributors to the system should be painless and intuitive with proper validation	Inserting new distributors to the system is done through simple user-friendly dialog with strong but user-friendly validation	Figures 28-32

<sup>6</sup> This feature is typically implemented using JFormattedTextField

4. Inserting points earned by distributors to the system should be painless and intuitive with proper validation	Inserting new points to the system is done through simple user-friendly dialog with strong but user-friendly validation. User has the option to select only those users she is interested in.	Figures 32-37
5. Application should allow an intelligent search for distributors by various properties	Search by combination of tree criteria has been implemented. It allows searching by non-precise criterion (e. i. only parts of information)	Figures 22-23
6. Profile of each distributor should display well-arranged material about the distributor and allows the user to immediately access all important information	The user profile window is a standalone window which has been designed with strong cooperation with user. All data is easy to follow, the dialog show all existing information about particular distributor.	Figures 38-39, Figure 42
7. Application should offer multiple ways of finding out the changes in productivity of the distributor (tabulated data, graphs)	Monthly information about particular distributor is displayed in table, as well as in interactive chart.	Figures 38-97, Figure 42
8. The program should offer an easy-to-use interface for displaying and browsing the structure of distributors registered below any distributor	Displaying children distributors for every distributor is implemented in two ways: as a table and interactive clickable tree of distributors.	Figures 40-41



## D: Documentation

### D1: Hardcopy of program output

In this section I will gradually go through the program and show screenshots of its operation.

#### First run of the program

If the program recognizes that files needed for program operation are not ready, it assumes this is the first run of the application. Therefore, it runs the initiation process.

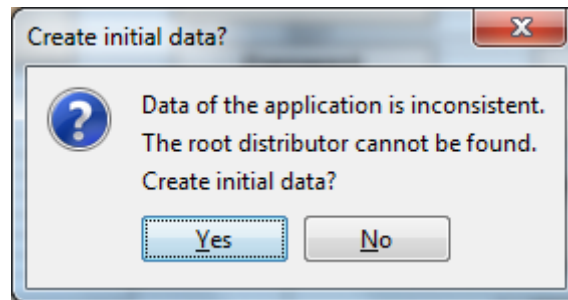


Figure 16: The first launch screen

Since the data could be corrupted if a hardware error occurred for instance, not always user really wants to generate initial data. That is why a question is displayed. If user answers "no", the application is terminated. Otherwise, the initialization process continues to next window.

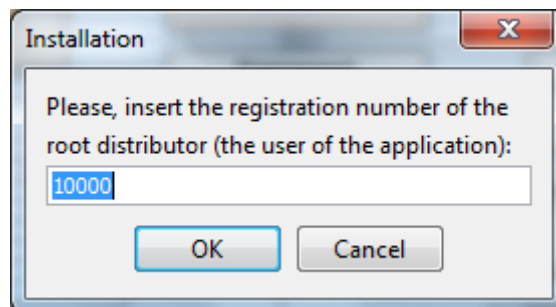


Figure 17: Insertion of the registration number of the root distributor

Now the user inserts the registration number of the root distributor in the application. The registration number is the only kind of data which is unchangeable in the application. That is the reason why it is the first thing the program requires to be inserted. If the user continues in the process, following message is displayed:

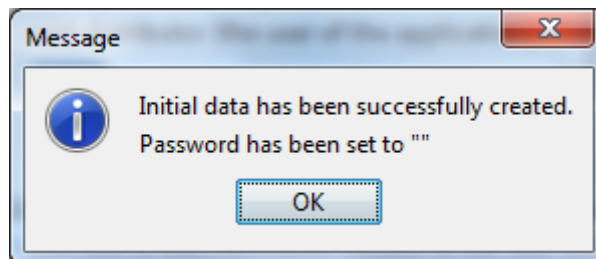


Figure 18: Message: Initial data has been successfully created

## Logging in to the system

If application's data are consistent, program continues to work. The first thing it is interested in after launch is the password.

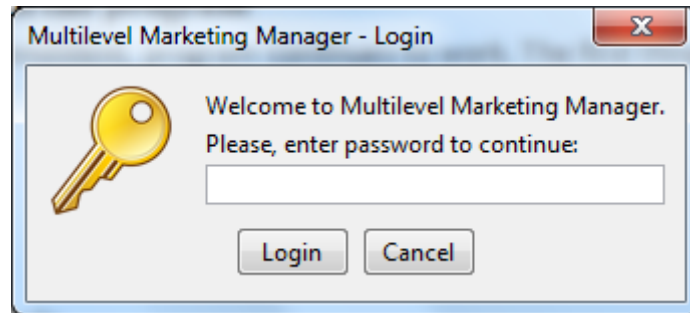


Figure 19: Login screen

If wrong password is inserted, warning message will be displayed.

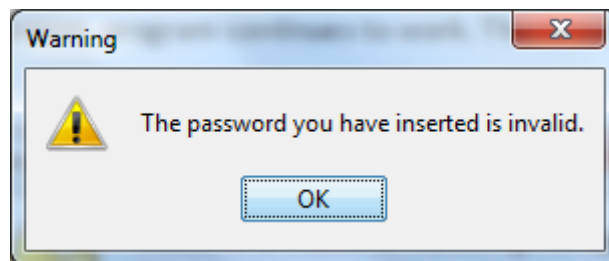


Figure 20: Invalid password

## Main window of the application

Otherwise, the main window of the application is launched.

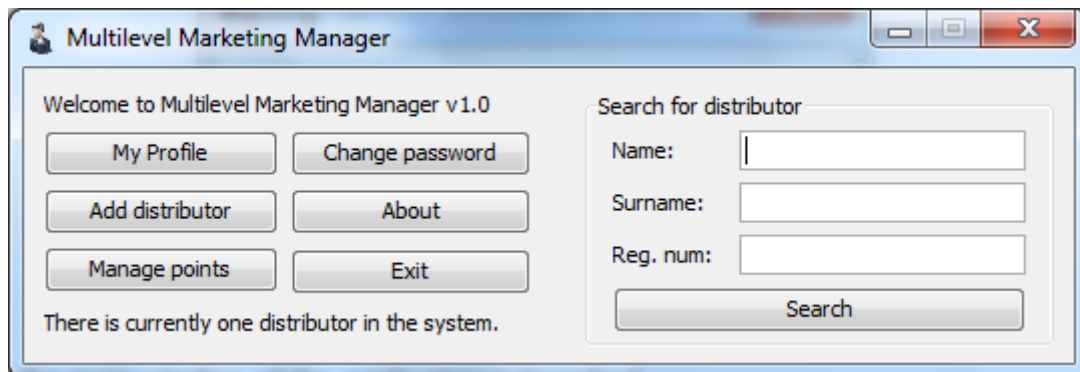


Figure 21: Main window of the application

At this point all the following screenshots will be done with consistent example data inserted in the program so that features of the application are more apparent.

If more distributors are present in the system, it makes sense to search for the distributor. This can be done according to combination of tree criteria in the right top corner of the application.

Welcome to Multilevel Marketing Manager v1.0

Search for distributor

Name:

Surname:

Reg. num:

There are currently 11 distributors in the system.

Search Results for //1

Reg. num.	Name	Surname	E-mail	Country
10000	Juraj	Masar	mail@jurajmasar.com	SK
158000	John	Cash	juraj.masar@gmail.com	CZ
54871	Peter	Gabriel	peter@gmail.com	GB
41205	Suzanne	Porta	porta@biz.com	SP
10500	Richard	Nixon		US
78105	Barbra	Stresissand		US
78401	Romeo	Swift		CZ
12345	Marian	Velarine		SK

Figure 22: Main window of the application with search results displayed

If a search has been done, results of such search are displayed in the same window, which height is prolonged. In case more criteria have been fulfilled, there are fewer results.

Welcome to Multilevel Marketing Manager v1.0

Search for distributor

Name:

Surname:

Reg. num:

There are currently 11 distributors in the system.

Search Results for J/as/1

Reg. num.	Name	Surname	E-mail	Country
10000	Juraj	Masar	mail@jurajmasar.com	SK
158000	John	Cash	juraj.masar@gmail.com	CZ

Figure 23: Fewer results of the search done with more concrete criteria.

If any row in the search results table is double-clicked, the profile of particular distributor opens in separate window.

### About dialog

Let's now move on to particular features of the application. **About button** displays a simple message concerning the application.

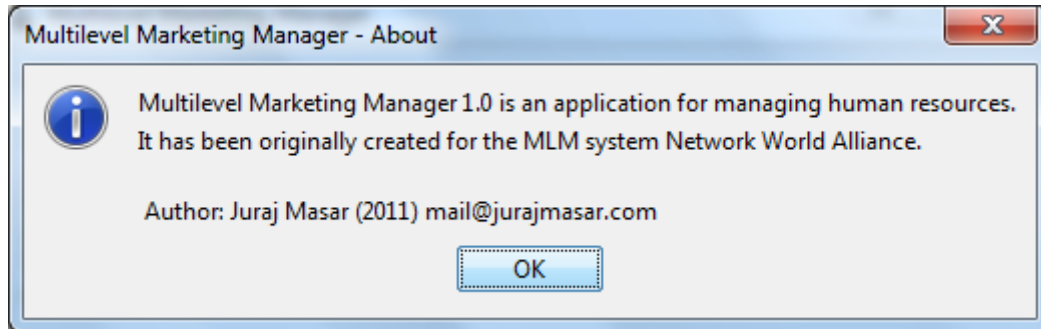


Figure 24: Message displayed by the about button

### Change password dialog

The **change password button** opens dialog for password changing.

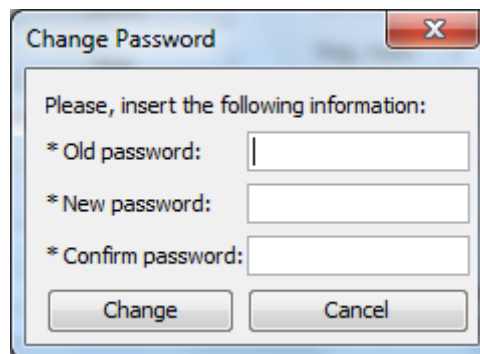


Figure 25: Password changing dialog

Since all the fields are required, all of them are marked with \*. If the "old password" does not correspond to the password actually set in the application, error message appears. This also happens when two "new passwords" are not identical.

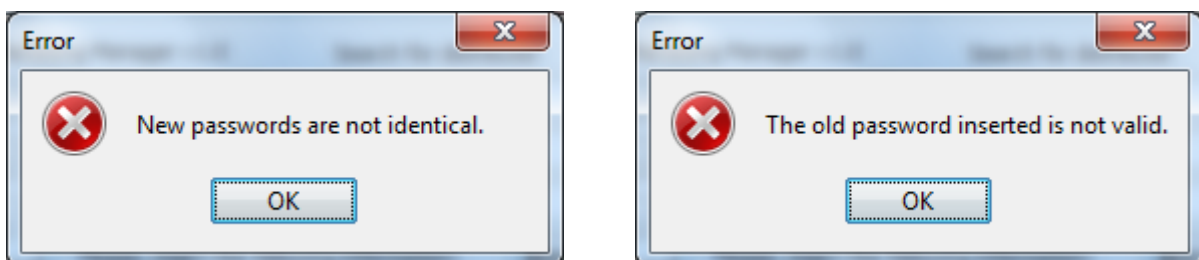


Figure 26: Change password dialog - error messages

If all inserted passwords are correct, the actual password in the application gets changed and user is again informed by a message.

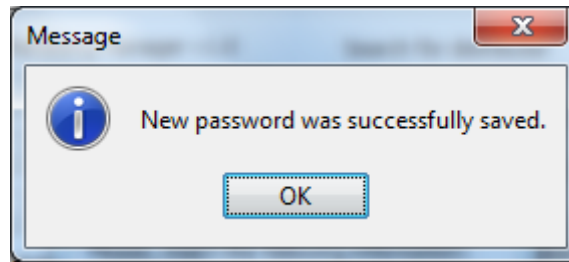


Figure 27: Password has been successfully changed dialog

### Add distributor dialog

The **Add distributor** button on the main window open the add distributor dialog.

The 'Add Distributor' dialog box has a title bar with the text 'Add Distributor' and a close button. The main area is titled 'Add new distributor'. It contains several input fields: '\* Name:' (empty), '\* Surname:' (empty), '\* Date of birth:' (containing '01.01.2011'), 'E-mail:' (empty), 'Telephone:' (empty), and 'Note:' (empty). On the right side, there are: '\* Registration number:' (empty), '\* Country:' (empty), '\* Date of registration:' (containing '01.01.2011'), '\* Sponsor:' (a dropdown menu showing '10000 (Juraj Masar)'), and 'Status:' with three checkboxes: 'Consumer', 'Salesman', and 'Manager'. At the bottom right, there are two buttons: 'Add distributor' and 'Cancel'.

Figure 28: Add distributor dialog

Again, compulsory fields are marked with \* and options are provided whenever possible (choosing sponsor using a select box, choosing status using checkboxes).

If an incomplete or erroneous form is submitted, error message with particular warnings is displayed.

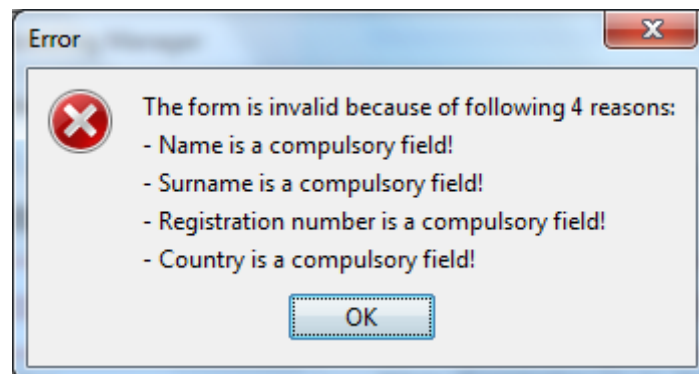


Figure 29: Add distributor dialog validation - error message with individual problems

After the message is closed, the problematic fields in the dialog are highlighted.

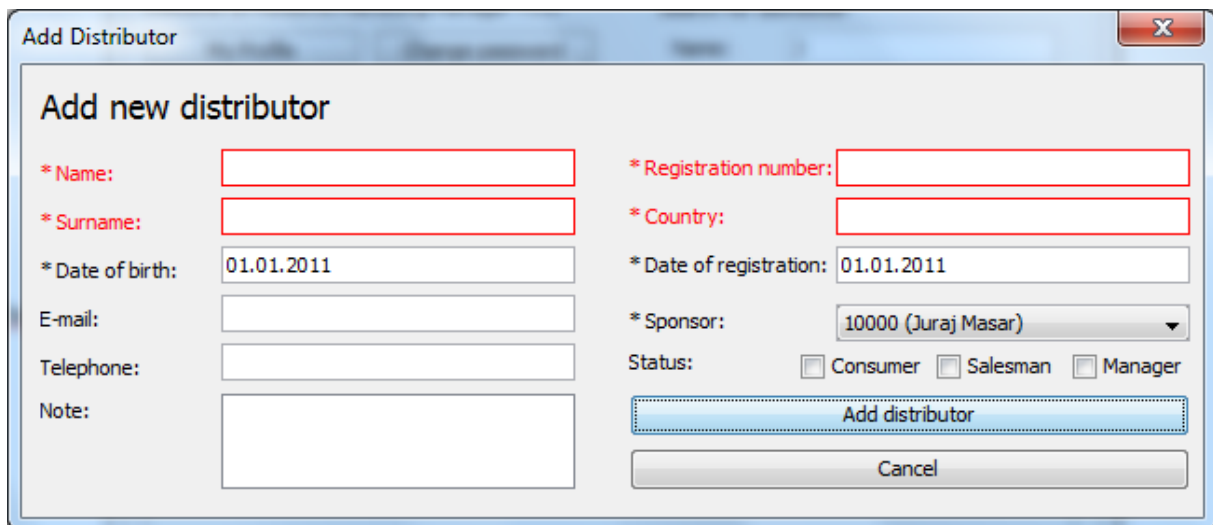


Figure 30: Problematic fields are highlighted

The validation is quite sophisticated. For instance name and surname are checked for consisting only from letters and registration number is checked to be unique.

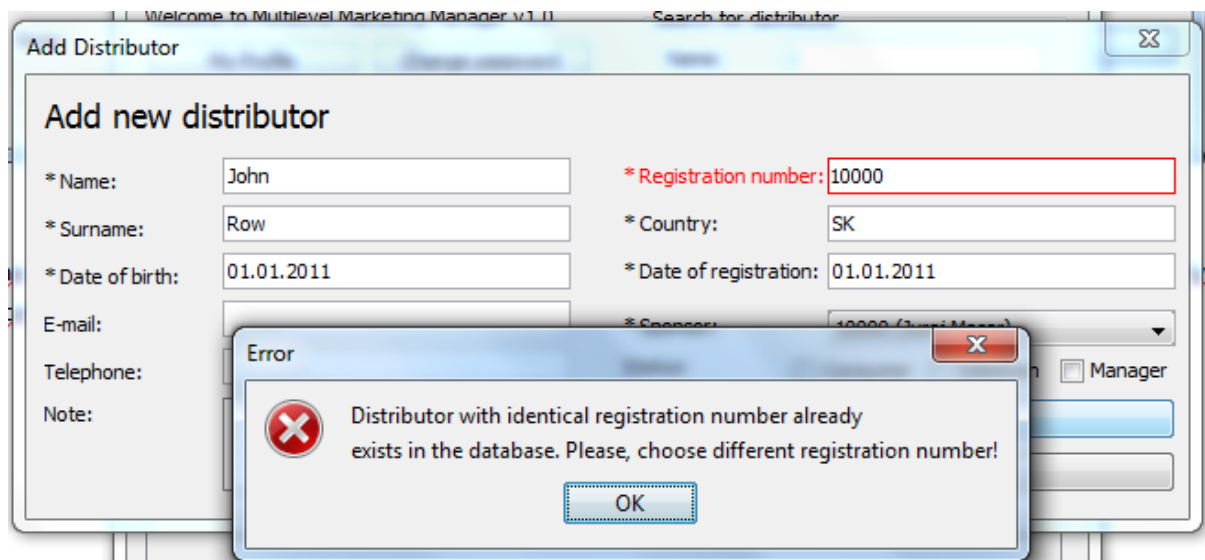


Figure 31: Sophisticated validation. Registration number is checked to be unique.

If all data has been inserted successfully, user is noticed.

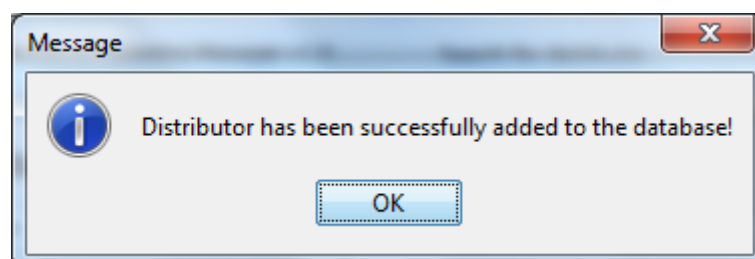


Figure 32: Distributor has been successfully added

## Points management

Another button on the main window labeled **manage points** runs the series of dialogs dedicated for updating points for distributors in the system. I will now briefly describe this process.

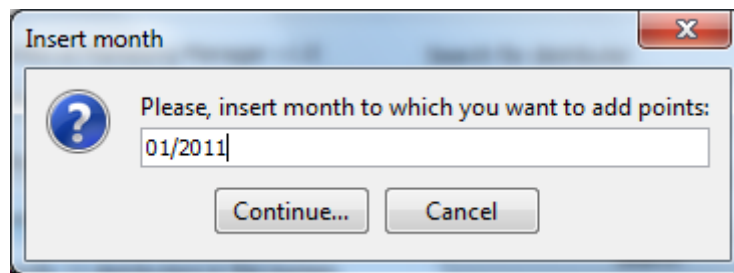


Figure 33: Managing points - choosing month

Firstly, the user chooses the month for which they want to manage points. It can be inserted only in format provided – error warning is, of course, present if something goes wrong throughout the whole process.

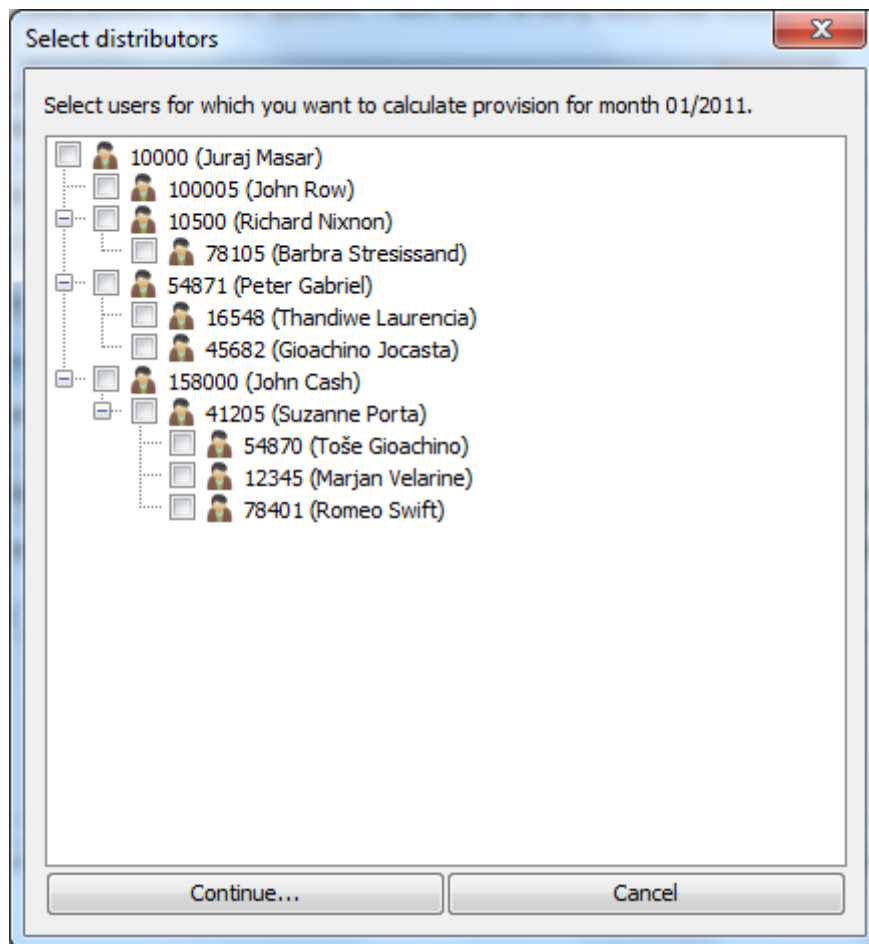


Figure 34: Choosing distributors for which we want to manage points

In this dialog the user selects distributor for which they want to administer they points for selected month. This is done using custom designed control – JTree with checkboxes.

On the following screen a table with distributors appears. In the table there are typically additional distributors to those actually selected. This is because of the fact that provision of a distributor is calculated upon their points, but also upon points of their direct children distributors.

Therefore, if one wants to calculate provision for distributor they select, all children of that distributor appear in the table, too.

This table can be edited after double-clicking a cell. If the user tries to input a character which is not numeric, error message is displayed.

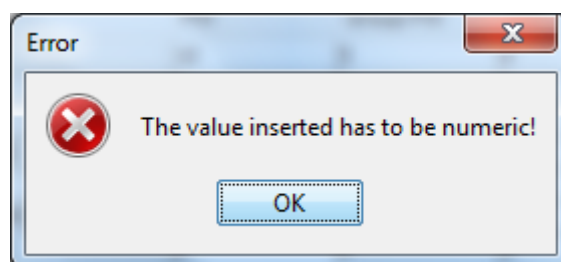
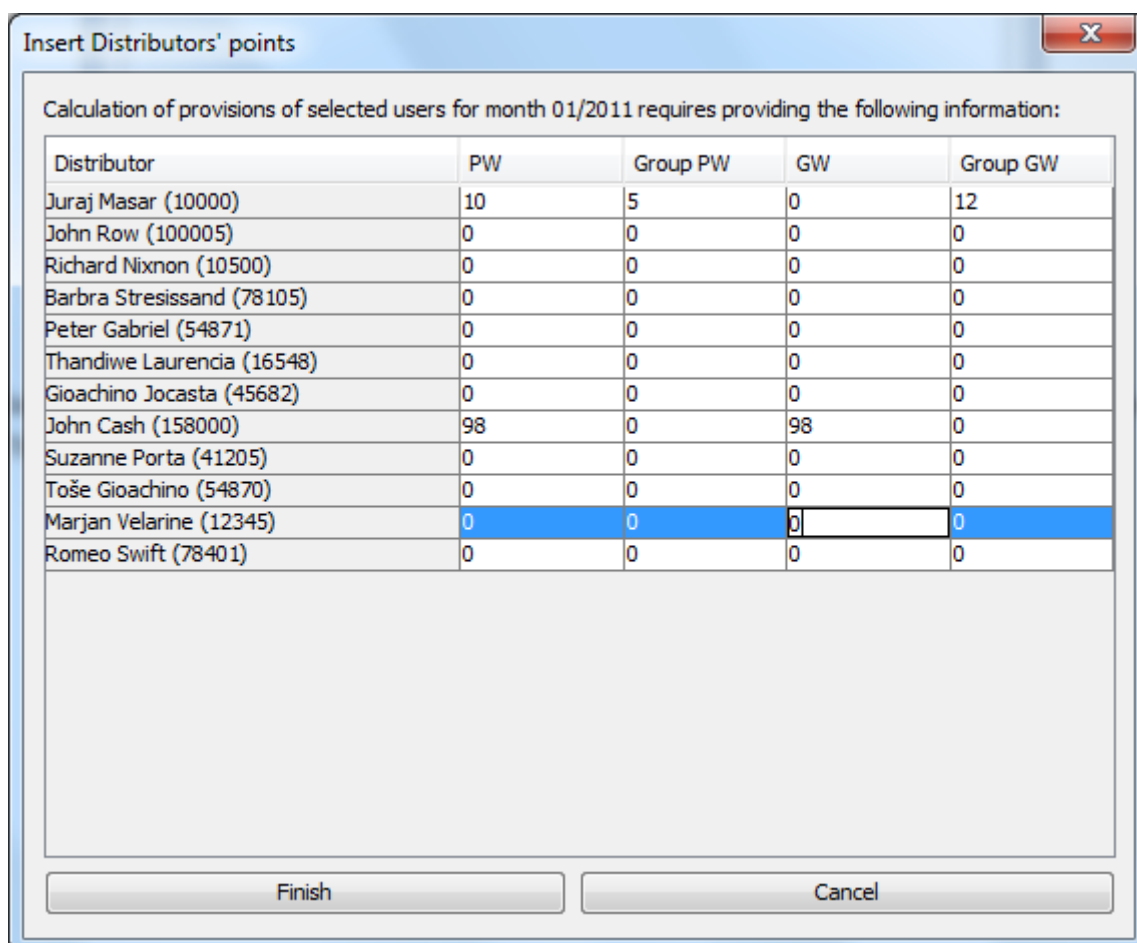


Figure 35: Error message from the points management table

A dialog box titled "Insert Distributors' points" with a blue title bar and a red 'X' icon. It contains a table with 5 columns: Distributor, PW, Group PW, GW, and Group GW. The table lists 14 distributors. The row for "Marjan Velarine (12345)" is highlighted in blue. Below the table are "Finish" and "Cancel" buttons.

Distributor	PW	Group PW	GW	Group GW
Juraj Masar (10000)	10	5	0	12
John Row (100005)	0	0	0	0
Richard Nixon (10500)	0	0	0	0
Barbra Stresissand (78105)	0	0	0	0
Peter Gabriel (54871)	0	0	0	0
Thandiwe Laurencia (16548)	0	0	0	0
Gioachino Jocasta (45682)	0	0	0	0
John Cash (158000)	98	0	98	0
Suzanne Porta (41205)	0	0	0	0
Toše Gioachino (54870)	0	0	0	0
Marjan Velarine (12345)	0	0	0	0
Romeo Swift (78401)	0	0	0	0

Figure 36: Points management table. Individual cells are editable after double-clicking.

When Finish button is pressed data in the application gets updated and success message is displayed.



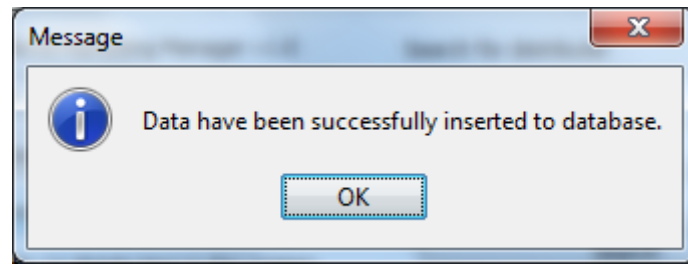


Figure 37: Points management - Points have been successfully inserted to the system.

### Distributor profile

When the **My profile** button on the main window is clicked the window of the root distributor is opened. Also, if a row in the search results table on the main window is double-clicked, profile of particular distributor is displayed.

The application takes care of the windows management of profiles – each profile can be opened only once in the same time.

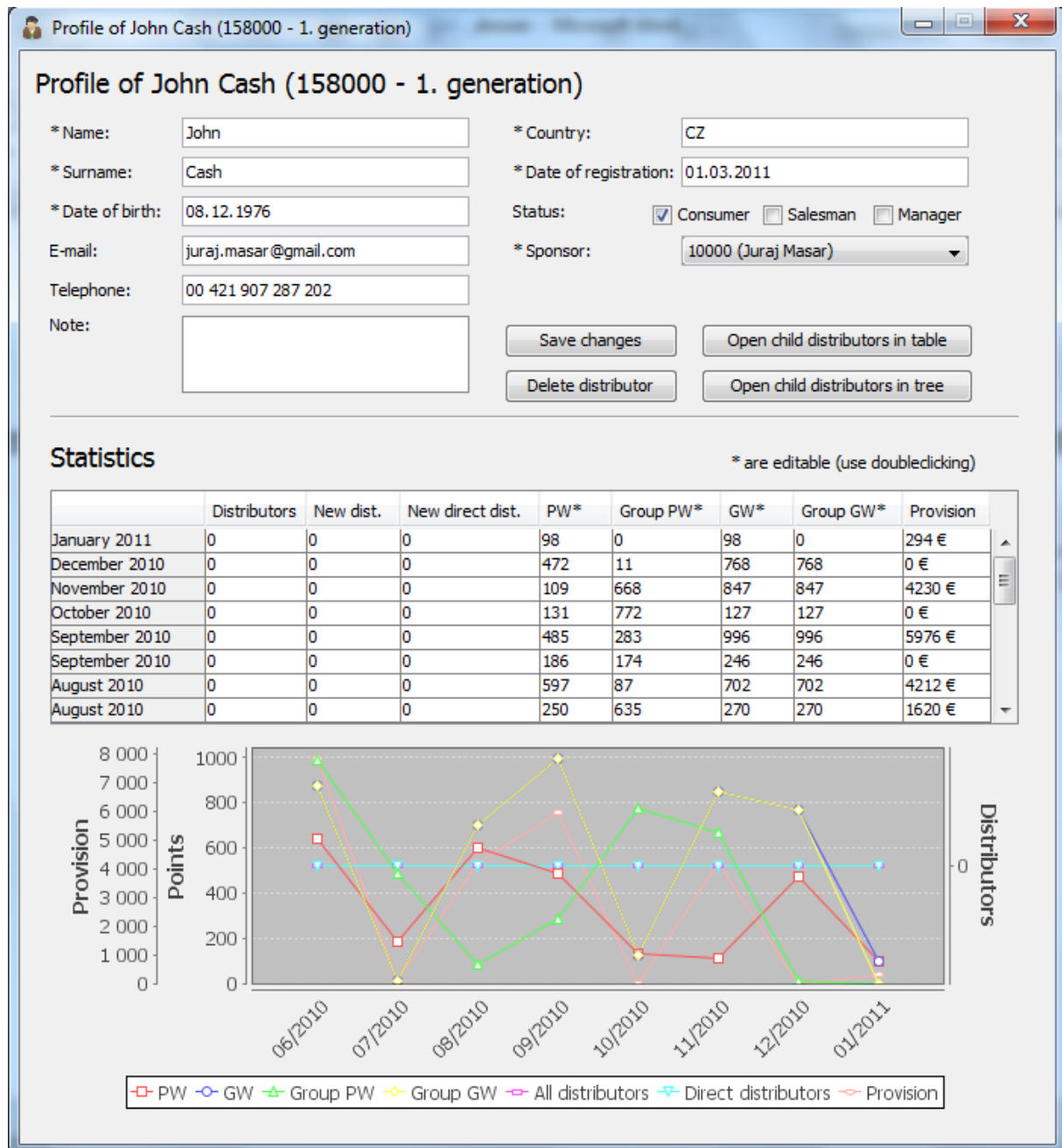


Figure 38: Distributor's profile

The profile window of a distributor consists of two main parts: the information about the distributor, and statistics of their progress.

The information part works very similarly to the add distributor dialog. Every field is fully validated and error message is displayed when attempting to submit incorrect information. For columns in the table in the statistics section of the profile marked with the \* are editable. This provides an easy interface for editing distributor's points right from their profile without the need to opening the Points management dialog.

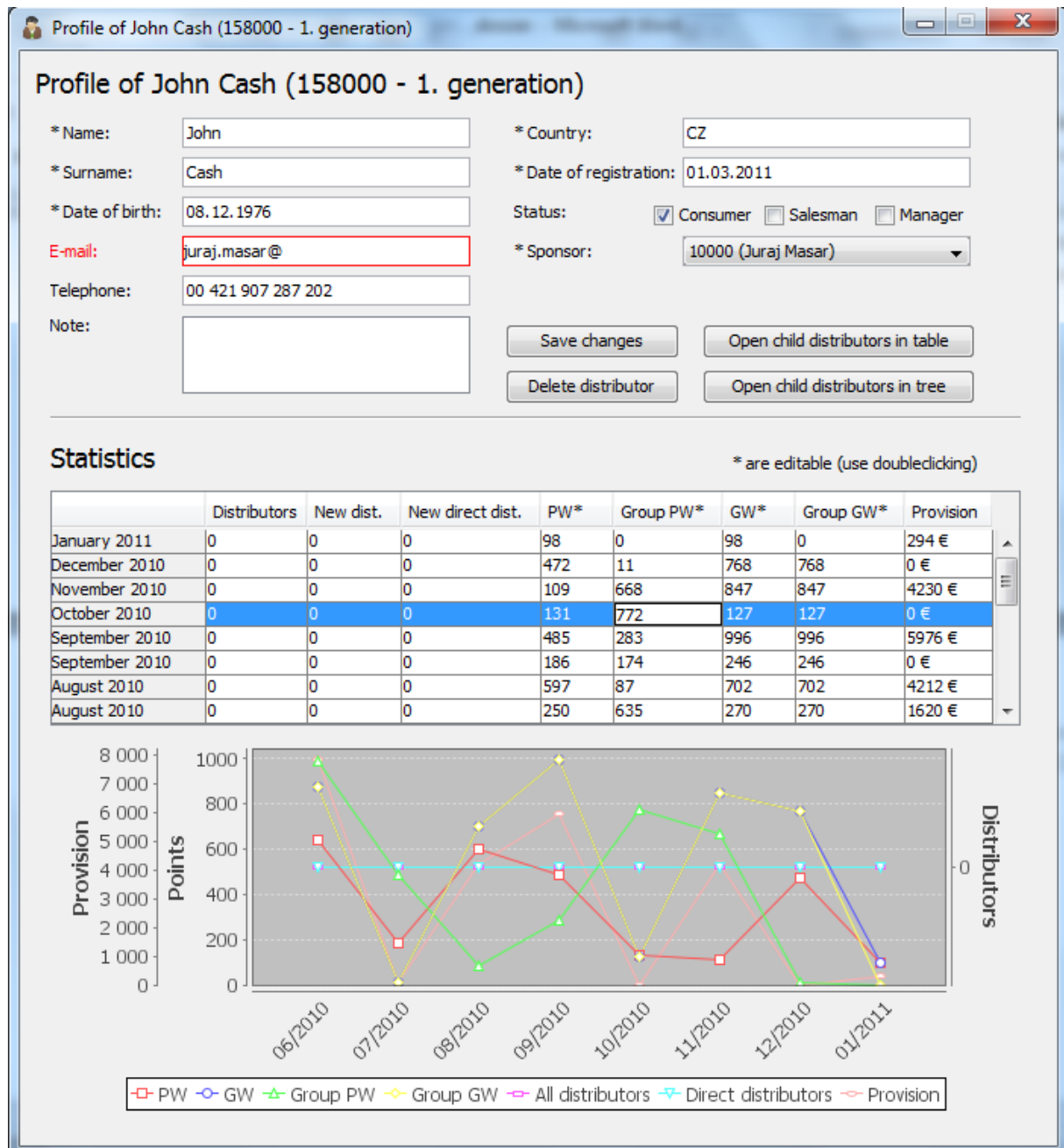


Figure 39: Profile of a distributor with displayed error notification (because of invalid) email and edition of raw in a particular column of the table.

The chart in the profile is a 3<sup>rd</sup> party control freely available – JFreeChart. Its current implementation in the profile allows advanced features, such as zooming to particular ranges of data or having different axes for different data series.

There are two additional buttons on the profile page of a distributor. **Open child distributors in table** and **Open child distributors in tree**. Their functionality is similar; both of them display the child distributors of the particular distributor. They, however, use different controls for doing so. One emphasizes the hierarchical structure of the distributors in the system; the other allows going through distributors' details one by one.

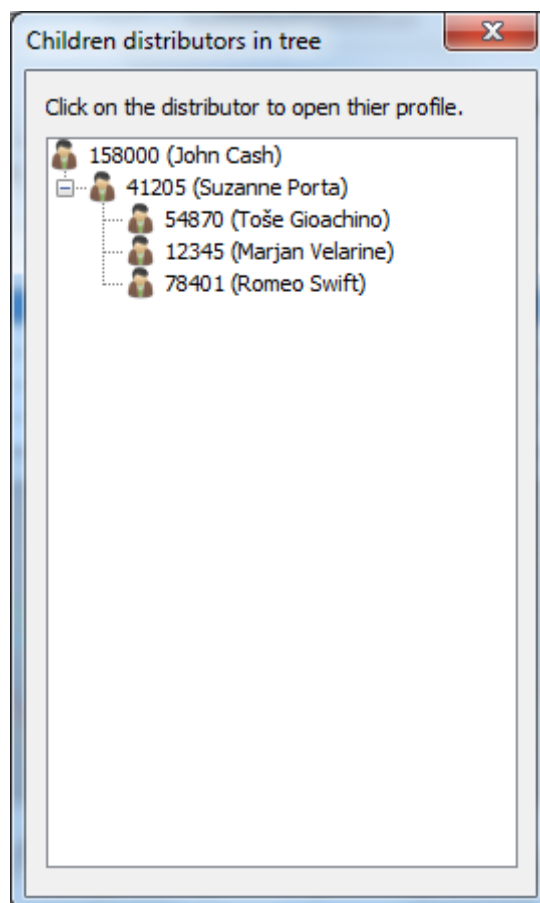


Figure 40: Children distributors in a tree

Children distributors of John Cash in table

Doubledclick on the distributor to open thier profile.

Reg. n...	Name	Surname	E-mail	Country	Status	Reg. date	Telephone	Note
41205	Suzanne	Porta	porta@biz.com	SP	S	01.01.2005		
54870	Toše	Gioachino		SK	C	01.01.2011		
12345	Marjan	Velarine		SK	C	01.01.2011		
78401	Romeo	Swift		CZ	C	01.05.1995		

Figure 41: Children distributors in a table

When any dialog of the application is opened, it is opened modally. Therefore, the rest of the application is not clickable. The only exception to this is the profile of the distributor. More profiles of distributors together with the main window of the application can be opened in the same time. This is because if user wants to compare individual distributors, there is nothing easier as putting two windows side by side each other and compare statistics.

The following screenshot shows two profile windows opened. Note, that the profile window of the root distributor is slightly different from the profile window of other distributor. The *delete distributor* button does not appear on the profile window of the root distributor, since this distributor cannot be deleted.

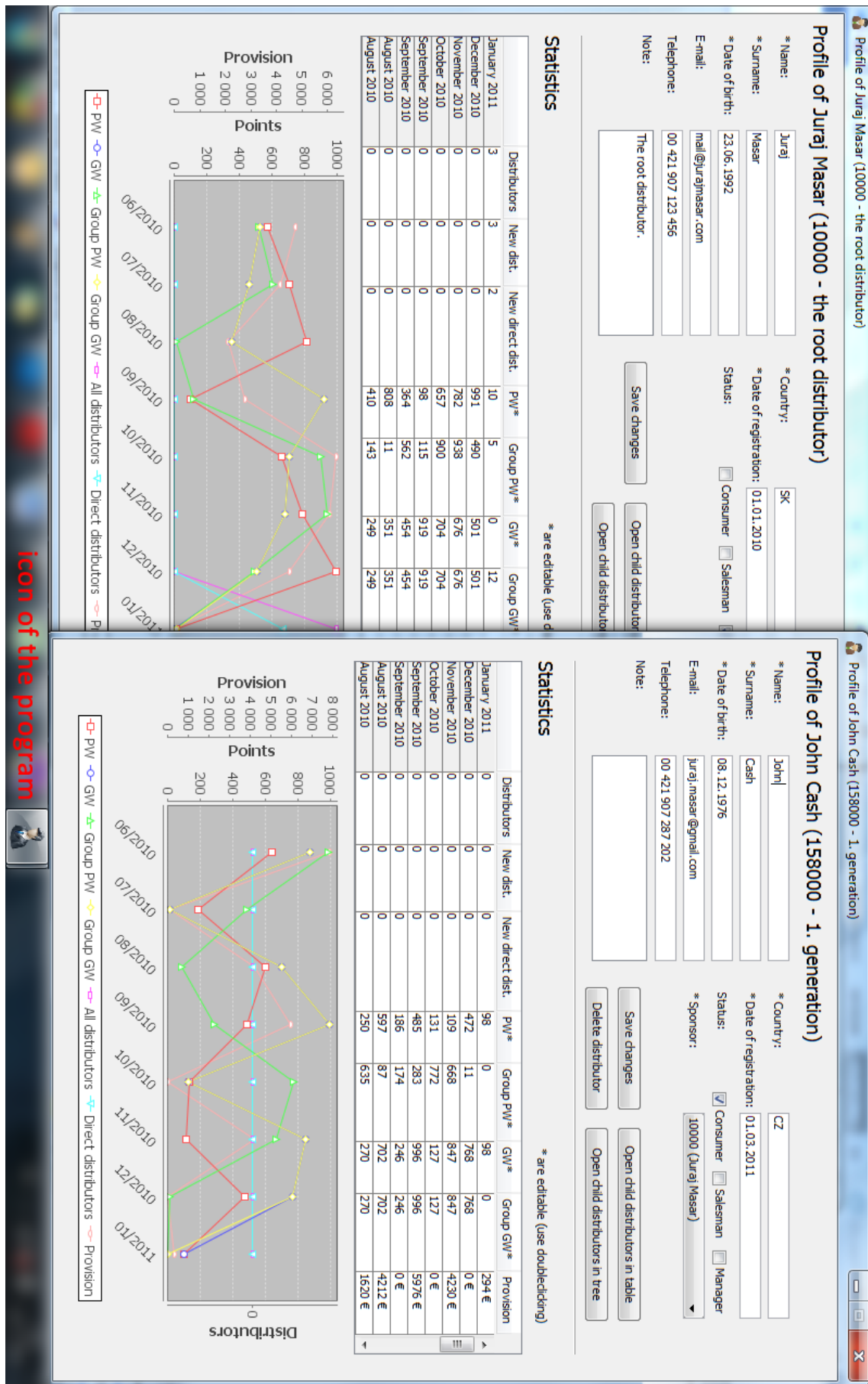


Figure 42: Differences between the profile window of the root distributor and generic profile window

### Excerpt from the program's data file

As it has been mentioned previously, data is encrypted in the file of the program, too.

This is the excerpt from the file *distributors.dat* which contains the information about individual distributors:

```
Ô2F9b4CDPeBqHiaEE1o6TnsZSjmNDcQXBP2hh2HopTnN4c4oz+ZcY43/1ZDrnQHQjN1a
C17gtT8F8LII3u51AJOkYIm1qmhvZpxV6hnjee8izH75ZFtMfD8BHxv//pHRwqhSW74
Q5ZPHmhlLaB8IIw==
ÔMdEGeogqciqQm91stZ+/ciXf4+HzNtT+ElqOHEGBbjvUKeyPbeJC5R9sv3xA4iJ6BWU
x1iHV7m3WFKylsSE7gn5dXL7ETKyxrP+2rKVg3n/IO+ixbILLsDQFnry5Flr
ÔtwpNaPkZ6qdqcd6UmGTqa6tObXPKzb4FRAhACW4YMrBFX+5wnukvoDMDovpWt5kGFc7
sSi69fN91jthcgaAG2r+BU+b6hOgm
Ô6LazMf+DgJ5xvBoVsYCBbQFtB4rYzAcOVYeufkjmZeNDyRzbUN6uVap2NNYSGJ4S9b
vduqy4X3hgQ06bE0ilRyM6u0YUuwv
ÔvRdcCY4LnIZnZhSqNOvOe3JVPwIDfxlUwvtDBYeXyMSbtTFTB3Wr7Jj+7wzkJyJxZp3
uyy1FaYFqiAKJXmvTNA==
ÔZB7+JgGv2REgE2XZAGoaudSdsXpslLrK8Hsgf0m9Z5bgfWIsaL2fcp453NJfC6xaUdv
7d68BrGMj9zm3bTy+og==
ÔrFCQmRib1EmFK1JdgKwxQkOPlaVGsgQE8phj2UK4LaMilsc69ibKqTwdSari3vKofTR
vvoOoXkM=
Ô+qy7m8r4MHKfYED64n4xhw2KhM3B5SavSvRtdsUxGbSA7sJzIPuih5nNCYwZbmrhb4t
nbJFhGP7Kg3C3x5pXyQ==
ÔubqvFjHUEG52fN7jRuovKc5UluJ0vynT7ny1HO/VvW65tXC0tcoyNCF4SFqKtRrFyRD
uVR7lNa6NM6wSXNdYFA==
ÔCKoMVTRdrlZt5k6X1cAznqIKZoh/rSMusiJpvaNf2Zhwknu328A7p8LPkVsEgQUIoBX
vemWbaKxp5HQ2IQOj9A==
ÔKpnZ+muMGvecwmXak0W6mNDXV7q68oHN9RBzygAsMulpv1YcWWrLWsBrdDCU+N792R2
2grs0qjtIAmyR1l73Bg==
Ô7T30RETlJxQAkd4NJ7w/VVyqsA02q+9rjfhhaMYB7klIW/t+rB2jZhS336u362yEA05
A8tV+Jsk=
```

Figure 431: Excerpt from distributors.dat file

## D2: Evaluation

The program has fulfilled the criteria set for its success, therefore, it is considered as successful. It works for all of the possible values which can be inserted to the system since proper validation of all data inputs is done.

The initial design has been appropriate since the program has been developed without any major changes since the stage B – design.

The main aim of the application was to be as user-friendly as possible. According to feedback from the user it is assumed that this has been successfully accomplished, too.

All calculations done on the data provided by user are valid and using the program instead of the current solution should save the user a considerable amount of time. All information entered to the program is safely **encrypted** on hard drive. The access to the application is **protected by password** as wished by the user, too.

As for the efficiency of the program, the program tries to be **as efficient as possible**. Whenever it is likely that it will have to cope with large amount of data, *indexing in memory* has been designed. All algorithms have been designed with the *best time complexity* possible, as known to the author.

If talking about efficiency improvements, the efficiency of using memory could be improved. Variables throughout the program sometimes contain more values than necessary (even though that this was typically avoided). Also, records saved in the RandomAccessFile have a constant length. Therefore, more disk space is consumed than is actually required. This is, however, not such big issue since disk spaces is now in ranges of GB and we are dealing with small text information.

Other improvements are always possible, however. The implementation of such improvements would have to be considered more early in the development - in the analysis stage. The program currently works only for one user, so **multiuser** support should be considered.

In case of implementing support for more users, since all of the users are operating in the same network, some data could be potentially shared among users. Thanks to this feature, insertion of duplicated data would be avoided.

Also, the program currently operates only on Microsoft Windows. Since it has been developed in Java, which offers **cross-platform support**, it is likely that running this program on other operating systems would be possible, too. This would, however, require extensive testing of all features of the application on the different operating system.

At this point all data the application operates with have to be directly inserted by the user. If a digital form of this data would be provided – for instance a structured tabulated format directly generated by the system used in the MLM company NWA, our program could potentially **parse this data and process it automatically**, without other actions from the user. This however assumes existence of such digital form of data, which is not available at this time.

Last but not least, a totally **different form of application** should be considered. If the application has been designed for multiuser support, it had better be developed in form of **web application**. The advantages of such solutions are clear, in this case: system would require no maintenance from the



user, since it could be centrally administered by a professional person. All data the application is working with could be automatically shared among users if necessary. The application would be also easily accessible from all operating systems, since the only software requirement would be working internet browser. Also, expansion to other platforms such as tablets or mobile phones would be much easier.

## Mastery

#	Mastery	Evidence	Page
1.	Adding data to RandomAccessFile by direct manipulation of the pointer using the seek method	DirectAccessFile.java line 145	112
2.	Deleting data from an instance of the RandomAccessFile class by direct manipulation of the file pointer using the seek method.	DirectAccessFile.java line 259	116
3.	Searching for specified data in a file.	DistributorManager.java line 395	133
4.	Recursion	Static.java line 263	80
5.	Merging two or more sorted data structures	Static.java line 295	81
6.	Polymorphism	Distributor.java line 390	186
7.	Inheritance	Distributor.java line 9	147
8.	Encapsulation	Item.java line 14	227
9.	Parsing a text file or other data stream	Record.java line 97 Password.java line 58	158 152
10.	Implementing a hierarchical composite data structure.	DistributorManager.java line 625	141
11.	The use of any five standard level mastery factors:		
	1. Arrays	Static.java line 175	77
	2. User-defined objects	Main.java line 13	97
	3. Objects as data records	Distributor line 8	174
	4. Simple selection (if-else)	Static.java line 182	77
	6. Loops	Static.java line 179	77
	7. Nested loops	Static.java 236	79
	8. User-defined methods	Static.java line 178	77
	9. User-defined methods with parameters	Static.java line 202	78
	10. User-defined values with appropriate return values	Static.java line 364	83
12.-15.	implementation of abstract data types (ADTs)	Vector.java line 9	190
16.	Use of additional libraries (such as utilities and graphical libraries not included in Java Examination Tool Subsets)	Chart.java line 17	331
17.	Inserting data into an ordered sequential file without reading the entire file into RAM.	Password.java line 24	151
19.	Arrays of two or more dimensions.	Static.java line 286	81