UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería **Informática**

Project Report – May 2012

Simplified Enterprise Resource Planning System

Alumno: Miguel Cruz Zúnica (micruzu@ei.upv.es)
Tutor upv: Juan Carlos Ruiz García (sri@inf.upv.es)
Tutor leicester: Stuart Kerrigan (sk233@le.ac.uk)
Titulación: Ingeniería Técnica en Informática de Sistemas

Project Report submitted to the University of Leicester in Partial Fulfillment for the degree of Bachelor of Science.

Word count: 11415 words

Table of contents

0 Abstract pg. 1

1 Introduction

1.1 Research Context pg. 1

1.2 Aims pg. 2

1.3 Objectives pg. 2

2 Requirements And Specifications

2.1 Data pg. 2

2.2 Config pg. 3

2.3 Sales pg. 3

2.4 Purchases pg. 4

3 Design

3.1 Development Methodology pg. 4

3.2 System Design pg. 5

3.3 Programming Language pg. 5

3.4 Database pg. 6

3.5 Development Environment And Plug-ins pg. 6

3.6 GUI Technology And Usability pg. 7

3.7 Software Repository pg. 8

3.8 Limitations Of Users pg. 9

3.9 Performance pg. 9

3.10 Installation Requirements pg. 9

4 Implementation Details And Challenges

4.1 Classes And Layers pg. 9

4.2 Database Access Methods pg. 11

4.3 Backup And Restore pg. 12

4.4 Exception Handling pg. 12

4.5 Privileged And Non-Privileged Sessions pg. 13

4.6 Quotations To Sale Function pg. 13

4.7 Search Dialogs And Filters pg. 13

4.8 Interface Components pg. 14

4.9 Creation, Saving And Deletion Procedures pg. 15

4.10 Total Price Boxes pg. 16

4.11 Finding And Applying Discounts pg. 16

4.12 Database Internal Structure pg. 17

5 Software Testing pg. 18

6 Planning And Timescales pg. 18

7 Usage Tutorial

7.1 Starting/Stopping The Server pg. 20

7.2 Client Log-in pg. 21

7.3 Main Window pg. 21

7.4 Users Management pg. 22

7.5 Restore/Backup pg. 22

7.6 Clients **pg. 24**

7.7 Search **pg. 25**

7.8 Products **pg. 26**

7.9 Suppliers **pg. 27**

7.10 Discounts **pg. 28**

7.11 Quotations **pg. 29**

7.12 Add/Edit Product **pg. 30**

7.13 Sales **pg. 31**

7.14 Purchases **pg. 32**

8 Critical Appraisal **pg. 33**

9 References **pg. 35**

DECLARATION

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross referencing to author, work and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name:

Signed:

Date:

0. Abstract

Simplified Enterprise Resource Planning is a management software developed in *Java* [1]. It is composed of a client and a server. The server encloses an embedded *Derby* relational database [2] which stores all information in the system while preserving its structure according to a schema. Additionally the server permits to specify the port at which it will listen for client's connections and to start or stop itself.
The client uses *Java's Swing* technology to create a simple yet elegant interface focused on efficiency. The features of the client include: creation, modification and deletion of clients, products, suppliers, discounts, quotations, sales and purchases. In addition, only the administrator user can create, modify or delete users as well as create or restore backups of the database.
When creating operations such as quotations, sales and purchases only the identifiers from the products, clients and other data objects involved are required, the rest of the information is obtained from the profile of the respective objects in the database. For selecting these identifiers proper search functionalities are included with which objects can be searched and loaded based on their characteristics even if their internal identifier is not known. Moreover, the system keeps track of the product's stock and when sales or purchases are created stock is adjusted accordingly in order to ensure the consistency of these operations. Additionally there is a unique functionality on quotations and sales to search, and apply if available, an efficient combination of discounts for the current operation based on the products it involves.
Another important characteristic of the system is that it is self-contained and ready to work out of the box. Furthermore, the server's database is already initialized with the tables and fields required by the client and it is presented with empty except for the administrator's user credentials.
Finally, the only requirement of the system to run is to have installed the *Java Runtime Environment* [3], which is available in all major operating systems including *Microsoft Windows* series [4], *Linux* [5] and *Mac* [6]. This enhances the versatility of the product allowing it to run natively on any operating system indistinctly.

1. Introduction

1.1 Research Context

An enterprise resource planning [7](ERP from now on) is a system designed to manage information from a business or an entire organization. It integrates both external and internal information across departments. From sales, accounting and manufacturing to customer relationship, its goal is to automate and enhance control over business' information.
Commonly it consist of a centralized database, a server and multiple clients. The system is meant to work in real time or next to it. Moreover, the range of features it may include is wide and usually requires to be personalized for each company's requirements. For this reason the cost of these products highly depend on the features included as well as the volume of transactions involved.
As mentioned before software integration is one of the goals of these systems in order to avoid dependency on multiple different IT solutions.
Finally, ERPs are becoming a key factor for success in today's high-tech world where competition between companies is fierce. The problem, however, is that often this kind of software is too complex and expensive, specially for smaller businesses which usually end up using a combination of less efficient solutions.

1.2 Aims

The aim of the project is to build a simplified ERP (SERP from now on) that cover the basic features required by businesses in a simple and generic way that will allow even small companies to use it out of the box without big investments.

1.3 Objectives

The objective of the project is to create a modular system with starting features including: quotes, sales, purchases, users management, clients and suppliers profiling, products and discounts specification. The modular nature of the product should permit extending it with additional functionalities easily, avoiding code repetition and benefiting from existing module's resources. For the same reason the look and feel must be consistent among all modules.

To accomplish this, both a client and a server should be developed. The server will authenticate users login credentials and provide information from database when requested from a client. On the other hand, the client will prompt a login screen, after which it will present the modules available to the user depending on his access group.

Furthermore, when creating a quotation, sale or purchase only the identifier from the data objects involved (clients, products or suppliers among others) should be required, while the rest of information will be obtained from their respective database fields. However, as remembering identifier numbers of objects is not realistic nor efficient, it will always be possible to search through them by the means of dialogs. These dialogs should also permit to filter searches by several characteristics related to the objects involved.

Additionally, after creation, deletion or modification of a sale or a purchase, the stock of the products involved must be adjusted as required in order to keep the database consistent with the reality. There should also be constraints applied to operations and fields to maintain consistency among the database. An example of this would be not being allowed to create a sale to an inexistent client.

In addition, a unique feature from this software will be a tool to find and apply an efficient combination of discounts to a quote or a sale. It should also be possible to create a sale out of a quotation with a single button click, preserving all the data from the quotation into the sale and linking both objects between themselves.

Finally, the administrator, and only him, should be able to create, modify and delete users, as well as to create backups from the database and to restore them afterwards.

2. Requirements And Specifications

The system functionalities are organized in four modules. All of them, except configuration one, have in common that their objects can be searched through by the means of dialogs, filtering search results by several relevant characteristics and loading desired result to be displayed and its information completed from the database.

2.1 Data

This module is dedicated to the management of data objects that are themselves used in other modules. Its components are:

- Clients
  Here instances of clients can be viewed, created, deleted or modified. In addition information about them like their first name, last name, address, email and company name can be filled.

- Products
  Here instances of products can be viewed, created, deleted or modified. In addition, information about them like their name, family, model, serial number and price can be filled. A count of the product's stock is displayed but can't be edited. This stock count will be automatically adjusted by the system when processing operations like sales or purchases.

- Suppliers
  Here instances of suppliers can be viewed, created, deleted or modified. In addition, information about them like their first name, last name, address, email and their company name can be filled.

- Discounts
  Here instances of discounts can be viewed, created, deleted or modified. In addition, information about them like their name and the amount of money discounted can be filled. Furthermore, a specific amount of any products can also be specified as a requirement for the discount to be eligible to be applied.

2.2 Config

This module is dedicated to SERP's configuration and privileged operations. For this reason this module is only presented to the administrator user:

- Users
  Here instances of users can be viewed, created, deleted or modified.

- Backup/Restore
  Here backups from the database can be created into a specified destination in the server's machine. In addition, existing backups in the server's machine can be restored specifying their path and name.

2.3 Sales

This module is dedicated to sales:

- Quotations
  Here instances of quotations can be viewed, created, deleted or modified. Furthermore, all the information related with the client or the products is obtained from their respective profiles. Also, there is a button to create a sale out of the current quotation. In addition, a quotation cannot be deleted or modified if there is a sale made out of it. Quotations don't affect the stock of products. Moreover, there is a button to search, and apply if requirements are meet, an efficient combination of discounts based on the products involved in the quotation. Finally, there are three fields that display the total base price of the quotation, the total discount reduction

and the total final price after discount reduction respectively.

- Sales
  Here instances of sales can be viewed, created, deleted or modified. Furthermore, all the information related with the client or the products is obtained from their respective profiles or quotations. In addition, the stock of the products being sold will be updated on creation, modification or deletion of the sale. Moreover, there is a button to search, and apply if requirements are meet, an efficient combination of discounts based on the products involved in the sale. Finally, there are three fields that display the total base price of the sale, the total discount reduction and the total final price after discount reduction respectively.

2.4 Purchases

This module is dedicated to acquisitions:

- Purchases
  Here instances of purchases can be viewed, created, deleted or modified. Furthermore, all the information related with the supplier or the products is obtained from their respective profiles. In addition, the stock of the products being bought will be updated on creation, modification or deletion of the purchase. Finally, there is a field that displays the total amount of the operation.

3. Design

3.1 Development Methodology

The development methodology chosen for the project was a custom one based on the principle of agile development [8], and more specifically on iterative development. Following this principle, on every iteration a small functionality was designed, implemented and tested before to move on to the next step. This effectively prevented code rewriting and helped building a more robust system while minimizing the time spent on debugging.
A number of factors and reasons were considered in order to choose this methodology over the rest:

- The limited experience of the developer in the ambit of software design implied that the design was prone to require changes from the original concept. For this reason, if a formal software development cycle [9] had been chosen, it was likely that during the coding and testing stages design flaws would have arisen forcing the project to return to the design stage and thus increasing the amount of time invested in the design stage as each time the whole system would have to be redesigned instead of a small part of it as happens on a more agile methodology.
- The also limited experience of the developer in both the programming language chosen and the tools used in the project (ie: the database model chosen) presented a challenge during the design stage because the coding techniques used had yet to be researched and tested to make sure the concept designed was feasible. Once again agile development methodologies offer the benefit of dividing development in small iterations so that both tests are more frequent and changes are easier to adapt because less volume of work is involved in each iteration.
- The modular nature of the system propitiated the adoption of an iterative

development so that modules were designed and implemented one by one instead of all being designed at the beginning before any implementation occurred.
- Along the same lines, due to the limited experience of the developer and the existence of a deadline to finish the project, an iterative development methodology allowed the project to grow from core functionalities to extra features ensuring that deadline would not arrive without the project basic implementation being fully finished.

For all these reasons the agile development methodology was considered more appropriate for the project.

3.2 System Design

To design the system, functionality specifications were elaborated and used, and as a result of this, implementation was faster and easier. In addition, another benefit was a reduction in the chances of both committing mistakes and having to rewrite code once in the implementation phrase.
Furthermore, each software product (server and client) was divided in three layers: presentation, logic and storage, although the client only made use of the first two. On the server the presentation layer embraced minimum aspects like switching on and off the server through a user interface and specifying the port at which the server would listen requests from the clients. Moreover, the logic layer of the server comprehends the proper functionalities of the server, that is, accepting request from clients, processing them and answering the clients back. Processing client's requests, in turn, translates to establishing connections with the database, querying it and closing it when appropriate. Finally, the storage layer on the server is the database itself, which is embedded into the server application. On the other hand, the client's presentation layer embraces the user interface and all the features related to them, while its logic layer covers the functions and mechanisms that regulates the client's process as a whole as well as the functions that work as a link between the server and the different modules of the client that require database access by starting, closing and processing connections and requests with the server. The client does not store any information as it was previously mentioned, hence does not make any use of the storage layer.

3.3 Programming Language

The programming language chosen for this project is *Java*. A number of factors were taken into consideration in order to take this decision:
- The application is meant to run on a desktop computer, hence we don't need to consider languages only based on web platforms.
- *Java* is object-oriented.
- *Java* is free (as free beer) and so are most of its developing environments.
- *Java* is a very robust language, with a big community and plenty of documentation available.
- The developer has previous experience both with the language and its tools.
- *Java* was taught during the degree.
- Although producing lower performance than *C* based languages, provides a good balance between the easy of coding characteristic from higher level languages and the performance characteristic from lower level ones.
- *Java* has a number of well integrated database options from where to choose. This is

a requisite of this project for obvious reasons.
- *Java* has excellent networking tools that will be required in the project for the server to communicate with the clients across the Internet.
- It is platform-free and well extended, there are runtime environments available for almost all operating systems and architectures.

This last point, being available in multiple platforms, is the most important reason why *Java* was chosen as the programming language for this project. This is because one of the key features of the project is precisely the flexibility to run it or port it to any of the major operating systems and architectures available. The current trends in the information technology world point towards a diversification in the usage of operating systems, with *Linux*, *Mac* and mobile operating systems gaining a lot of presence both in the industry and within home users. This can potentially target niches of customers that use operating systems other than the ones in the *Microsoft* series and where this kind of management solutions are not widely available. Furthermore, it allows the server to run natively into a different operating system than the clients, what is a common practice in the industry where most servers run in *Linux*, without requiring the use of a virtual machine.
Finally, the system is meant to run under the execution environment *JavaSE-1.6*.

3.4 Database

The database chosen for this project is the *Apache Derby* relational database, also known as *Javadb*, in its version 10.8.2.2. The reasons are because it is free (as in free beer), it is well documented and perfectly integrated into the *Java* platform, it is small sized (2MB footprint) and compact, and on the top of that, because it can be embedded.
The database is embedded into the server in order to minimize the requirements and the installation process. In addition, by embedding the database it can be released initialized with all the tables, fields and configuration required by the project ready to be used out of the box, without relying on third application tools or processes. Finally, an embedded database offers more security as the database is closed and cannot be accessed after the server application is closed. On the other hand, a down side of embedding the database is a potential lose of performance and scalability as the volume of data grows, but it is deemed acceptable because the target of this project are small business which data volume should not represent a problem for the *Derby* database.

3.5 Development Environment And Plug-ins

It was decided to use an integrated development environment (IDE from now) for this project for a number of reasons:
- They allow you to navigate through the code without worrying about *namespaces* or projects.
- They have syntax correction and auto-completion that greatly enhances the coding experience as well as minimizing the errors during the debugging phrase.
- They automatically generate code to create and initialize some objects, which saves time at coding.
- They provide of refactoring functionalities that improve the coding experience.
- They provide of live documentation when hovering over special syntax objects, functions and libraries.
- They warn you of errors as you type, in some cases even prior to compiling the code.

- They provide of an integrated and organized view of the coding environment (console, files, code, warnings among others).
- They provide of advanced debugging tools vital for error fixing.
- They allow visual programming to some extent, as for instance they provide of tools to visually create user interfaces and to automatically generate the code for them.
- They provide ways of testing the software with ease.

To conclude, they enhance and improve the coding experience as well as integrate multiple tools required for developing.

The IDE chosen for this project is the *Eclipse IDE for Java Developers* [10] in its 3.7 version codename *Indigo*. *Eclipse* was chosen over other popular solutions like *NetBeans* [11] or *Idea* [12] because its free (as in free beer), open-source, it has been introduced during the degree, it has a number of plug-ins to integrate multiple tools that were required for this project, it is well documented and has a huge community to support it.
The following plug-ins for eclipse were used in order to provide extra functionality and to integrate all the required developing tools into the IDE:

- *Data Tools Platform Enablement for Apache Derby*: provides tools to implement and connect to the *Derby* database as well as an interface to manage it and interact with it.[13]
- *Swing Designer* and *WindowBuilder*: provides graphic tools to design *Swing* interfaces and to automatically generate their code.[14]
- *Subclipse*: provides graphic tools in the form of a contextual menu to share projects through S*ubversion* protocol, which was required by the university.[15]

3.6 GUI Technology And Usability

Java offers two official components in order to develop graphic interfaces: *Abstract Window Toolkit* (AWT) [16] and *Swing* [17]. In addition to these there is an alternative component called *Standard Widget Toolkit* (SWT) [18] developed by *IBM* [19] and maintained by the *Eclipse Foundation* [20]. SERP uses *Swing* to create all the user graphical interfaces for a number of reasons:

- In contrast with SWT, *Swing* is part of *Java's* library, thus it does not require additional native libraries.
- *Swing* works in all major platforms.
- *Swing* has an integrated GUI editor for *Eclipse*, which is the IDE used to develop the project.
- *Swing* has better support and documentation from sun than SWT.
- *Swing* includes all AWT features as *Swing* is a newer and more sophisticated version of it.

Furthermore, usability was an important factor taken into account when designing the user interfaces during the development process. However, in some cases performance, which is also another form of usability, was prioritized over more descriptive or intuitive designs because SERP not only is supposed to be used strictly by internal members of the company, never by their clients, but it is also expected to be used intensively on a daily basis. In this specific context it is acceptable to trade an instruction period with a reasonable learning curve to get familiarized with the system and its functionalities for better performance and usage speed once the mentioned instruction is completed.

Other aspects that were taken into account when designing the user interface were:

- Since the system is going to be used by non technical users it is important that graphical user interfaces are as simple and minimalistic as possible.
- As the system is going to be used in a business environment, the main theme should make use of plain and soft colors and the font should be a formal one.
- Due to the modular nature of the system, all modules should maintain the same design patterns. This implies for example that if there are multiple objects like buttons that share a similar functionality across different modules, they should look as similar as possible between them, be positioned in the same place and also react to user interaction in the same way.
- Any eventuality should be informed in a clear way to the user by the means of pop-up messages that unambiguously inform of what happened in the system.
- Users should only be presented with the functionalities available with their access group.
- When processing time consuming operations the cursor should change to reflect that the system is processing the operation in order to avoid this event being misinterpreted by the user as a collapse of the system. In addition, a message should be displayed together with the change of cursor.

3.7 Software Repository

The usage of a *Subversion* software repository [21] was a mandatory requirement by the university according to the module study guide. The process of interacting with the repository was handled by the means of the *Eclipse*'s plug-in *Subclipse* as mentioned in the development environment and plug-in section.
Furthermore, repository was structured as follows:

- code
    - serp
        - branches
        - tags
        - trunk
    - serpc
        - branches
        - tags
        - trunk
- Docs
- Other

Other directory is empty. *Docs* contains both the project plan and the project final report. Under *code* we can find two directories: *serp* and *serpc* that corresponds to the server and the client's subprojects respectively. Under each subproject there are three directories: *branches*, *tags* and *trunk*. *Branches* are empty, *trunk* contain the most updated version of the code for the respective subproject and *tags* contain copies of the project that were created at certain points as a milestones. In addition, most if not all of the *tags* and *trunk's* updates were accompanied by a commentary explaining the changes being made in order to help keeping track of the development's time-line.

3.8 Limitation Of Users

Although the first version of the system doesn't have any limitation on the number of concurrent users connected to the system, a future market version would implement a limit and charge for any extra user slot. This is not only a marketing strategy but also is required to ensure that the performance of the system is kept under reasonable values.

3.9 Performance

Only one process per server or client should be created. Furthermore, once the application is closed the process should die and be handled appropriately by the operating system. Moreover, the system should run smoothly without any sensible wait time. However, operations like searches, backups and restores are an exception and should be used with caution by the user.
Regarding memory and cpu consumption there are no special requirements, both the client and the server require around 35MB of ram memory to run.

3.10 Installation Requirements

The system will require the *Java Runtime Environment* to be installed on the system to work. In addition, if the machine hosting the server is required to be accessible from the Internet, the networking devices may need specific configuration to route the external ports used by the server towards the specific machine where the server is located.

4. Implementation Details And Challenges

In this section details about the implementation of the system as well as the internal mechanisms built and challenges faced during the coding and developing of the system will be explained and highlighted.

4.1 Classes And Layers

Both in the server and the client applications the layers are represented by the *namespaces* "*app*", "*data*" and "*ui*" referring to storage, logic and user interface layers respectively. Furthermore, in the server subproject the following classes can be found:

- Serp: Here is where the main() function is located. This creates a new server user interface and makes it visible. Furthermore, it does also contain the functions which start or stop the server. These functions create or stop a *SwingWorker* [22], which is a class from Swing to manage threads so that the server can listen to multiple clients simultaneously. Once the *SwingWorker* is created it does create a SerpServer object, which will listen for client requests and manage their queries.
- Server_gui: This is the server's user interface. When the start or stop buttons are clicked they execute the respective functions to start or stop the server, which are located at the Serp class.
- SerpServer: Creates a *Socket* and a *ServerSocket*. Moreover, the *ServerSocket* is initialized to listen on the port chosen by the user at the user interface. When the server is started it does listen for new connections to arrive to the *ServerSocket*, at which time it will assign the incoming connection to a *Socket*, then it will create and

start a serpServerThread class to process the mentioned Socket. In addition, when the server is stopped it does close the *ServerSocket*.

- serpServerThread: Accepts a Socket as a parameter, then it will wait until an object is received from the Socket by an *ObjectInputStream*, at which time it will process the received object with a serpProtocol class instance, which will return another object as reply, which will be sent by an *ObjectOutputStream* through the *Socket*. When this process finishes it will close the socket, the input and output streams and the serpProtocol.
- serpProtocol: This class will establish a connection with the database by creating both a Connection and a Statement instance, then it will process the object received by the serpServerThread, return an object as a reply to the serpServerThread and finally close the connection established with the database. To process the object received by the serpServerThread it will assume this object is a String, then it will check its start and depending on its start words it will proceed. In detail, five commands are accepted, otherwise an empty object is returned. Furthermore, the accepted commands start by "*Login*", "*Query*", "*COMMIT*", "*ROLLBACK*" and "*RESTORE*" respectively. Moreover, after receiving one of this commands it will proceed as follows:
    - "*Login*": Isolates the credentials from the command as they always follow the same structure, then it will query the database to check if they are correct and answer accordingly with a string "*Ok*" or "*Invalid*".
    - "*Query*": Isolates the second part of the command, which is a query for the database, it will then query the database with the obtained query and it will process and return the output from the database. Moreover, if the query starts by "*SELECT*" the operation will be of the type *Statement.executeQuery()*, while if the query starts by a different string it will be of the type *Statement.executeUpdate()*. Furthermore, to process the output it will assume the answer is in the form of a table and will create an *ArrayList* containing as many *ArrayLists* of objects as rows has the *ResultSet* obtained from the database, then it will fill each of those *ArrayLists* with the objects contained in the *ResultSet*. In addition, to obtain the number of columns from the *ResultSet* a *ResultSetMetaData* object is obtained from the *ResultSet*.
    - "*COMMIT*": This will trigger a *Connection.commit()* in order to commit changes into the database. In this case the returning object is left empty.
    - "*ROLLBACK*": This will trigger a *Connection.rollback()* to rollback changes from the database. In this case the returning object is left empty.
    - "*RESTORE*": This will trigger the restore function which is explained in its own section.

On the other hand, in the client subproject the following classes can be found:

- Main: Contains the main() function, which creates a new thread with a Serpc object.
- Serpc: Maintains a pointer to the current active user interface and to the *Socket* which is used to connect with the server. Furthermore, at the beginning it will create a login interface and make it visible. In addition, it implements the following functions:
    - Current active user interface (*JFrames*) getter, setter and a specific function to switch their visibility.
    - check_login: Creates a connection with the server by the means of a *Socket* and proceeds to check the validity of the user credentials which are obtained

as parameters.
    - Query: Generic function used by the rest of the application to query the server. It does accept an string and an integer as parameters, then it sends the string through the *Socket* and reads another object from the *Socket* as a reply. Finally, depending on the parameter's integer it will end sending through the Socket either a "COMMIT" command, a "ROLLBACK" one or nothing at all.
    - Exit: Takes care of closing the *Socket*, the *Input/Output Streams* and calls the system to *exit()*.
- Additionally, the client implements one class per each user interface involved in the subproject. Furthermore, these classes contain both the interface components and the functionalities associated with them.
- Finally, a class called *SortByDiscount* is implemented to be used as the sort method for an *ArrayList* of discounts to be sorted by profitability.

4.2 Database Access Methods

The following code is used to establish a connection with the database:

"*Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();*
Connection
conn=DriverManager.getConnection("jdbc:derby:MyDB;user=serp;password=1234");
conn.setAutoCommit(false);".

- The first line obtains a new instance of the driver in case is not available already.
- *MyDB* is the name of the database.
- user and password are required to access the database although this is a mere formality because as the database is embedded into the server it should not be accessible from elsewhere.
- The last line sets sets the database property *autocommit* to false, what implies that the user must explicitly force a commit or rollback for changes to take effect permanently in the database. In addition, due to *Derby* specifications after a commit or rollback a new transaction is started automatically. Finally, changes are reflected temporally while the current connection to the database lasts, but if not committed they are not preserved. Special caution must be taken in order to either commit or rollback before closing a connection with the database, otherwise problems may arise on the next connection requiring the database to shutdown and restart.

The first time ever the server is executed, as the database does not exist yet an exception is triggered, caught and handled by initializing the database internal structure (schema, tables and administrator user credentials). To do this a new parameter "*create=true*" needs to be added to the connection code for the database to be created. Afterwards the structure will be initialized by SQL queries. Following is the new connection code with the extra parameter:

"*Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();*
Connection
conn=DriverManager.getConnection("jdbc:derby:MyDB;create=true;user=serp;password=1234");".

Furthermore, after querying the database *Derby* specifications recommends to close the database. To do this not only the *Connection* instance has to be closed but also have a new connection must be established with an additional new parameter. The following is the appropriate code to accomplish this:

"conn.close();
conn =
DriverManager.getConnection("jdbc:derby:MyDB;user=serp;password=1234;shutdown=true");".

- The new parameter "*shutdown=true*" has been added to close the database.
- "*create=true*" parameter cannot be used at the same time as "*shutdown=true*".

4.3 Backup And Restore

Following is the *MySQL* [23] query used to create a backup from the database:

"CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE('path')". [24]

Where path is the full address to the directory where we want the backup to be created. If the directory does not exist it will attempt to create it. In addition, the default name of the backup is "*MyDB*".

Furthermore, to restore a backup of the database we first need to close database because *Derby* specifications require the database to be restored on the very first connection established with the database, otherwise it does not work.
After closing the database (creating a new connection with the parameter "*shutdown=true*") an *SQLException* is fired by the database with the code "08006" to inform that the operation was successful, then a new connection is created with the additional parameter "*restoreFrom=path*" where path must include not only the directory where it resides but also the full name of the backup. [25]
Finally, as the database remains closed a new connection is opened so that the client can continue working with the application.

4.4 Exception Handling

Following are some examples of exception types handled in the project:

- *NumberFormatException*: Handled when trying to convert a string to a number in case the input is not a valid number.
- *SQLException*: Handled in all queries and operations with the database.
- *IOException*: Thrown by several classes like *ServerSocket* when attempting to listen in a specific port.
- *ClassNotFoundException*: Handled when trying to create new objects from not standard classes.
- *ArrayIndexOutOfBoundsException*: Handled when filling a table in case the structure of the data does not match the one of the table.
- *UnknownHostException*: Thrown by *Socket* class when trying to connect to a server.

When any exception is fired the user is informed appropriately by the means of a pop-up alert box using the class *JOptionPane.showMessageDialog()*.

4.5 Privileged And Non-Privileged Sessions

In the client application, after a successful login, when creating the main window of the application, the login-in process checks the credentials of the user and if the user is not the administrator creates a different variant of the main window that does not present the restricted operations only available for the administrator: the user management module and the backup tool.

4.6 Quotation To Sale Function

In the client application, in the quotation module, there is a button on the right panel labeled "*Create sale*" that creates a sale out of the present quotation preserving the content of the same and linking the sale and the quotation together by the field both models have for that purpose.

Furthermore, to accomplish this operation a second constructor for the class *Sales_gui* was created which accepts as parameters information regarding the sale that is going to be created. In addition, the standard constructor was modified to use the previous mentioned new constructor with empty parameters in order to avoid duplicating all the code for both constructors. The modified code for the standard constructor is as follows:

```
"	public Sales_gui() {
		this(null,null,null,null,null,null,null,null,new DefaultTableModel());
		Clear_table();
	}".
```

And the new constructor looks like this:

"public Sales_gui(String id, String budget, String cl_id, String fname, String lname, String address, String email, String company, DefaultTableModel model)".

Where *id* is the identifier for the new sale being created, *budget* is the identifier from the quotation out of which the sale is being created, *cl_id* is the identifier of the client, *fname* and *lname*, *address*, *email*, *company* references the first and last name, address, email address and company name of the client respectively (this could be obtained from the database but this way there is no need to perform that extra query) and the model is the data structure that contains the products table information.

4.7 Search Dialogs And Filters

In order to perform searches through the database, list registered instances and load their data into different fields of various windows, a search functionality and dialogs have been created.
An example of this is found in the client's window: near the identifier field of the client there is a button labeled "?" that, if clicked, opens a dialog where we can search through the clients stored in the database, filter the search by different characteristics associated with those instances and load one by selecting it and clicking on the button labeled "*Accept*" or pressing the enter key while the desired client is selected.

To perform this operation this is an example of the code in the window which opens the dialog:

“ArrayList<Object> clients = new Search_client().searchClient();
if (clients != null && clients.size() > 0)
Load_cl((Integer)clients.get(0));”.

In this case *Search_client* is the class of the search window, *searchClient()* is a function of that class that returns an *ArrayList* of objects when then search process is finished and the user has chosen to load a client.
The last two lines of code ensures that the returned array is not empty and then proceeds to load the data from the first object into the array, which is supposed to be an integer (the identifier of the client), by the means of the *Load_cl()* function which will fill the fields related with the client's information of the window. Moreover, the returned array may contain different data in other similar instances where search dialogs are used and that data may be processed differently as well, for example for completing certain fields without the use of an additional function like *Load_cl* was in this example.

As for the dialog's code, it is opened in modal mode, what means that the focus cannot be changed from the dialog until this is disposed, and the mentioned *searchClient()* function of the example sets the dialog's window visible and returns a result value. Moreover, as the dialog is in modal model, the return command will only be triggered when the window is disposed. For this reason only after the return value has been assigned with the data requested by the user the *dispose()* command is executed, and subsequently, the window closed and the value returned to the window which launched the dialog in the first place.

4.8 Interface Components

The following *Swing* components are used to create the user interfaces:

- *JFrame*: Basic component for regular windows which contains the rest of the components.
- *JDialog*: Basic component for dialogs which contains the rest of the components.
- *JPane*: Used in all *JFrame*s and *JDialog*s as a container for the other components.
- *JScrollPane*: Used as a container for *JTables* so they can be scrolled if the number of rows displayed requires so.
- *JLabel*: Used to display text in a label.
- *JTextField*: Used to create a field for the user to fill information to be captured and processed by the application. In addition, the property “*editable*” is turned off for those fields that should not be editable by the user.
- *JButton*: Used to create buttons the user can click to fire events.
- *JPasswordFiel*d: Used in the client's log-in to hide the password with asterisks.
- *JTable*: Table used to display search results, products and discounts in various modules.
- *JMenuBar*: Used at the main windows to create the upper menu bar.
- *JMenu*: Used to add components to the menu bar.
- *JMenuItem*: Used to add entries to the menus.
- *Event Handlers*: Various event handlers are used to monitor different components like *JButton*s and to fire functions when certain actions like mouse clicking on a

button or pressing enter key are triggered.

4.9 Creation, Saving And Deletion Procedures

Following is a typical procedure to save an instance:

1. Data that composes the fields of the object being saved is gathered from the user interface.
2. Several checks are carried out to ensure all constraints and requirements are meet, otherwise operation is aborted. For example checking that the client exists or that mandatory fields are not empty.
3. If the identifier field is equal to “X” then the instance is considered new and a new identifier should be found. This new identifier will be the immediate highest natural number than the current highest identifier from the respective database table.
4. If there are multiple object types involved, like for example products or discounts and the quotation, sale or purchase is not a new instance (a existing one is being updated) previous relations of products or discounts with the quotation, sale or purchase are deleted from database to avoid repetition.
5. If there are multiple object types involved, like for example products or discounts, data from each of them is gathered individually and a new instance of the relation main_object-secondary (the main object being a quotation, sale or purchase and the secondary the products or discounts) are created through a SQL query. In addition, if the main object is a sale or a purchase the involved product's stock are updated accordingly with another SQL update query.
6. Depending if the object is new or already existed a SQL query performing an insert or an update is created and executed with the information from the main object.
7. If everything else worked without errors a commit command is sent to the database, otherwise a rollback is forced. In addition, the new or updated instance is reloaded to reflect the changes.

Following is the typical procedure to delete an instance:

1. Data that composes the fields of the object being deleted is gathered from the user interface.
2. Checks are carried out to ensure that the object meets the requirements to be deleted. For example in case of a quotation it cannot be linked to an existing sale.
3. If there are products or discounts involved, individual relations between the main object and them are deleted from the database and if its a sale or a purchase the stock of the products is adjusted accordingly.
4. The instance is deleted through a SQL delete query.
5. If everything worked without errors commit command is sent to the database, otherwise a rollback is performed. In addition, the instance with the closest lower identifier is loaded if possible, otherwise the fields are cleared.

A typical query to find a new identifier would be:
“SELECT id FROM TABLE WHERE id >= ALL(SELECT id FROM TABLE WHERE id<"+id+")”.

4.10 Total Price Boxes

The pointers at the bottom of quotations, sales and purchases reflecting the base price, total discount and total price of the operation are updated always that there is some change into the products or discounts table.

Furthermore, the calculation for the total price is obtained from the formula: *(total price) = (base price) – (total discount)*. In addition, if the total price is negative, it is then set to 0.

4.11 Finding And Applying Discounts

Both in the quotation and sale modules there is a button labeled "*Find Discounts*". When one of these buttons is actioned the system searches and attempts to apply the best combination of discounts available. The following represents how the algorithm performs this operation:

1. Empty the discounts table in the user interface.
2. Gather data from products involved in the operation and discounts registered in the database.
3. Create three *HashMaps*: the first is filled with data from the products involved, with the keys being the products ids and the values the amount of units. The second H*ashMap* is filled with information of the discounts requirements, the keys are the identifier from the discounts and the values are *ArrayLists* that contain one *ArrayList* per requirement of the discount with values of the required product id and the amount of units. The third H*ashMap* remains empty and will be filled with the discounts applied to the instance, with the discount identifier as key and the amount of times it has been applied as value.
4. Then the H*ashMap* containing the discounts requirements is iterated filtering only those discounts that meet the requirements of the present quotation or sale. These are stored into an *ArrayList* that contains one *ArrayList* per valid discount, which contains the identifier and the discounted amount as data. To create this list discounts are first discarded when a not valid requirement is meet and then the remaining ones are considered valid.
5. Following, while exists discounts into the list of valid ones, they are ordered by the amount of money they subtract multiplied by the number of time they can be applied, most profitable first and subsequently applied as many times as the units of products involved in the operation permits, removing discounts from the list of valid ones as they are found to violate the requirements. In addition, the identifier and number of time that has been applied is stored in the third H*ashMap* we mentioned.
6. When no more discounts can be applied or existence of products ran out, the data with the discounts that have been applied is introduced into the discounts table in the user interface, what triggers an update in the total discounted amount and total price boxes.

Note the operation is not committed until the instance is saved. In addition, the button labeled "*Clear Discounts*" empties the discounts table in the user interface.

4.12 Database Internal Structure

The following SQL queries were executed to create the database tables:

- "*CREATE SCHEMA SERP*".
- "*CREATE TABLE SERP.LOGIN (USERNAME VARCHAR(255) NOT NULL, PASSWORD VARCHAR(255),PRIMARY KEY(USERNAME))*".
- "*CREATE TABLE SERP.CLIENTS (ID INT NOT NULL,FNAME VARCHAR(255),LNAME VARCHAR(255),ADDRESS VARCHAR(255),EMAIL VARCHAR(255),COMPANY VARCHAR(255),PRIMARY KEY(ID))*".
- "*CREATE TABLE SERP.PRODUCTS (ID INT NOT NULL,NAME VARCHAR(255),MODEL VARCHAR(255),SN VARCHAR(255),PRICE DECIMAL(9,2),STOCK INT,PRIMARY KEY(ID))*".
- "*CREATE TABLE SERP.BUDGETS(ID INT NOT NULL,CLIENT INT NOT NULL,SALE INT,PRIMARY KEY(ID),FOREIGN KEY(CLIENT) REFERENCES CLIENTS(ID),FOREIGN KEY(SALE) REFERENCES SALES(ID))*".
- "*CREATE TABLE SERP.BUDGET_PRODUCTS(ID INT NOT NULL,BUDGET INT NOT NULL,PRODUCT INT NOT NULL,PPU DECIMAL NOT NULL,UNITS INT NOT NULL,DISC DECIMAL NOT NULL,PRIMARY KEY(ID),FOREIGN KEY(BUDGET) REFERENCES BUDGETS(ID),FOREIGN KEY(PRODUCT) REFERENCES PRODUCTS(ID))*".
- "*CREATE TABLE SERP.SALES(ID INT NOT NULL,CLIENT INT NOT NULL,BUDGET INT,PRIMARY KEY(ID),FOREIGN KEY(CLIENT) REFERENCES CLIENTS(ID),FOREIGN KEY(BUDGET) REFERENCES BUDGETS(ID))*".
- "*CREATE TABLE SERP.SALE_PRODUCTS(ID INT NOT NULL,SALE INT NOT NULL,PRODUCT INT NOT NULL,PPU DECIMAL NOT NULL,UNITS INT NOT NULL,DISC DECIMAL NOT NULL,PRIMARY KEY(ID),FOREIGN KEY(SALE) REFERENCES SALES(ID),FOREIGN KEY(PRODUCT) REFERENCES PRODUCTS(ID))*".
- "*CREATE TABLE SERP.SUPPLIERS (ID INT NOT NULL,FNAME VARCHAR(255),LNAME VARCHAR(255),ADDRESS VARCHAR(255),EMAIL VARCHAR(255),COMPANY VARCHAR(255),PRIMARY KEY(ID))* ".
- *CREATE TABLE SERP.PURCHASES(ID INT NOT NULL,SUPPLIER INT NOT NULL,PRIMARY KEY(ID),FOREIGN KEY(SUPPLIER) REFERENCES SUPPLIERS(ID))*".
- "*CREATE TABLE SERP.PURCHASE_PRODUCTS(ID INT NOT NULL,PURCHASE INT NOT NULL,PRODUCT INT NOT NULL,PPU DECIMAL NOT NULL,UNITS INT NOT NULL,PRIMARY KEY(ID),FOREIGN KEY(PURCHASE) REFERENCES PURCHASES(ID),FOREIGN KEY(PRODUCT) REFERENCES PRODUCTS(ID))*".
- "*CREATE TABLE SERP.DISCOUNTS(ID INT NOT NULL,NAME VARCHAR (255),AMOUNT DECIMAL NOT NULL,PRIMARY KEY(ID))*".
- "*CREATE TABLE SERP.DISCOUNT_PRODUCTS(ID INT NOT NULL,DISCOUNT INT NOT NULL,PRODUCT INT NOT NULL,UNITS INT NOT NULL,PRIMARY KEY(ID),FOREIGN KEY(DISCOUNT) REFERENCES DISCOUNTS(ID),FOREIGN KEY(PRODUCT) REFERENCES PRODUCTS(ID))*".
- "*CREATE TABLE SERP.CONFIG(COMPANY VARCHAR(255),TELEPHONE VARCHAR(255),EMAIL VARCHAR(255),ADDRESS VARCHAR(255))*".
- "*CREATE TABLE SERP.BUDGET_DISCOUNTS(ID INT NOT NULL, BUDGET INT NOT NULL, DISCOUNT INT NOT NULL, AMOUNT DECIMAL NOT NULL,*

UNITS INT NOT NULL, SUBTOTAL DECIMAL NOT NULL,PRIMARY KEY(ID), FOREIGN KEY(BUDGET) REFERENCES BUDGETS(ID),FOREIGN KEY(DISCOUNT) REFERENCES DISCOUNTS(ID))".

- "*CREATE TABLE SERP.SALE_DISCOUNTS(ID INT NOT NULL, SALE INT NOT NULL, DISCOUNT INT NOT NULL, AMOUNT DECIMAL NOT NULL, UNITS INT NOT NULL, SUBTOTAL DECIMAL NOT NULL,PRIMARY KEY(ID), FOREIGN KEY(SALE) REFERENCES SALES(ID),FOREIGN KEY(DISCOUNT) REFERENCES DISCOUNTS(ID))*".
- "*INSERT INTO LOGIN VALUES ('root','1234')*".

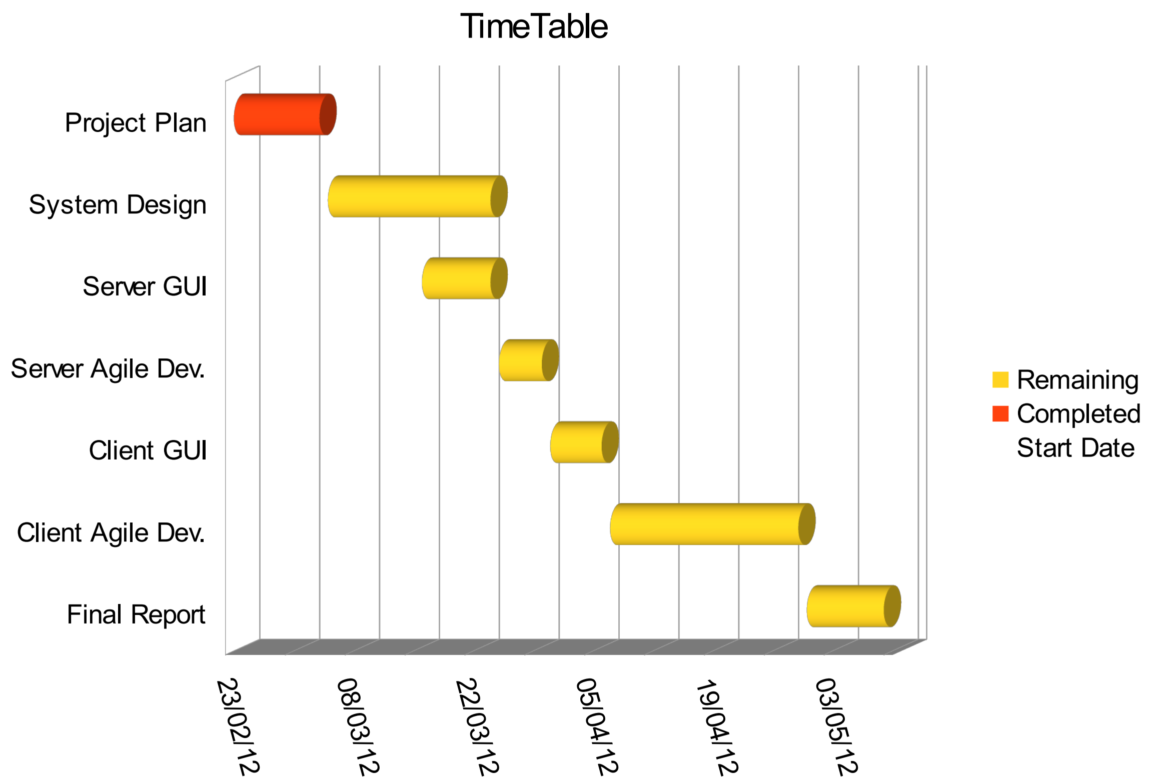
Where budget refers to quotations due to an early naming.

5. Software Testing

After each iteration of the development cycle tests were carried out to ensure the correct functionality of the features being implemented. In addition, after certain milestones, like completely finishing a module, were meet a test involving the entire application was carried out to ensure that the new functionality did not interfere with previous ones.
Furthermore, the mentioned tests were primarily black-box tests where functionality was tested by introducing certain relevant inputs and contrasting outputs with expected ones. Moreover, exceptions were also taken into account and tested to make sure they were properly handled.
The software has also been tested to run in different operating systems like *Linux Debian* 7 [26] and *Microsoft Windows XP*.

6. Planning And Timescales

The development cycle, which was composed of iterations of designing, coding and testing sessions was expected to require the most part of the available time. It was also difficult to predict the exact amount of time invested in this phrase as the very nature of it made it unpredictable. In the rare case that everything went abnormally smooth, additional features could have been implemented as the modular design of the system permits to extend it with ease. The last weeks were supposed to be spent on elaborating the final report.
This was the estimated time-line:



However, in the end there were some modifications: both the system design phrase and the user interfaces construction were merged onto the agile development iteration cycles. In addition, the server development phrase was not totally differentiated from the client one, but in reality after a first approach to both the server was changed again to its final design and then proceeded with the client until the end with only minor tweaks for bug fixing on the server part. Moreover, the final report took more time than was originally estimated. Still, overall the prediction was quite accurate.

Finally, a more realistic time-line would be as follows:

7. Usage Tutorial

This section will present a usage tutorial intended for the user to get familiarized with the user interface and its functionality. In addition, screen captures of each interface will be displayed with informative legends to complement them.
Furthermore, the tutorial will follow a natural cycle of actions in the server, that is: turn on the server, log-in from client as the administrator user, create a new user, log-in with the new user's credentials, create a new client, create a new product, create a new supplier, create a new discount, create a new quotation, create a new sale, create a new purchase, log-out and log-in with the administrator user again, backup and restore the database.

7.1 Starting/Stopping The Server

In the first capture the server is stopped while on the second one is running. In addition, both the caption of the button and the message over the port field informs whether the server is running or not.

7.2 Client Log-in

The server field accepts both *IP* numeric format and alternative formats like 'l*ocalhost*'.

7.3 Main Window

If the login was successful we are presented with the main window. Furthermore, the options available in the *config* column depend on the privileges of the user, in this case the user was the administrator and thus can access both user management and backup/restore functionalities. In addition, there is a menu bar on the top of the window that links with the same options as the buttons displayed. Moreover, in the case of the "*File*" menu bar option, it has two options: *Logout* and *Quit*. *Logout* will return us to the log-in window while *Quit* would terminate the client application completely.

7.4 Users Management

Here the administrator can both search the list of users registered and load one of them, create a new user, delete a user or edit a user's password. The "*Back*" button returns us to the main window.

7.5 Restore/Backup

To create a backup of the database we need to specify the path where we want the backup to be created in the server's machine. Moreover, if the path specified does not exist it will attempt to create it. The name of the backup by default is “*MyDB*” and will be created under the directory we specified. In addition, the status message reflects the current status of the window starting with “*Ready!*”, changes to “*Creating backup, please wait*” while creating backup and finishes with “*Done!*” when the operation is completed. The mouse also changes during the creation of the backup to reflect that the operation is taking place.

The restoration process works in the same way as the backing up one except that this time we have to specify the name of the backup to be restored in addition to the path where it is located in the server's machine.

7.6 Clients

- *ID*: This is the unique identifier of each client.
- Button “*?*”: Opens a dialog to search through clients. In addition, you can filter the search by characteristics and load the desired instance by double-clicking over it in the result list.
- Button “*Prev*”: Loads the immediate lower instance. If current instance is 0, doesn't do anything. If there is no instance loaded yet it will load the instance with the highest identifier as if the list of instances was circular.
- Button “*Next*”: Loads the immediate higher instance. If current instance is already the highest, doesn't do anything. If there is no instance loaded yet it will load the instance with the lowest identifier as if the list of instances was circular.
- Button “*New*”: Empties all fields and writes an “*X*” on the *ID* as a placeholder. It is important to maintain the “*X*” in the *ID* field until the instance is saved for first time as that will mark the instance as new and thus will look for a new *ID* when saving it.
- Button “*Del*”: Deletes the current instance and loads the immediate lower one if exists.
- Button “*Save*”: Saves the current instance. If *ID* is equals to “*X*” consider it a new instance.
- Button “*Back*”: Returns to the main menu.

7.7 Search

This is an example of a typical search dialog.

- It is opened in modal mode, what means that focus cannot be switch to a different window until the dialog is closed.
- By filling the fields you can filter the search result.
- As shown on the second capture *MySQL* syntax can be used in the search filters.
- If filters are left blank it will return the complete list of registered instances.
- There are some interesting complex searches available like for example searching quotations where certain product has been quoted (same applies for sales and purchases).
- There are search dialogs similar to this for each different kind of instance in the system and they can usually be accesses through buttons with the caption "*?*".
- By double-clicking on a row of the result table the dialog will close and the selected instance will be loaded into the field who opened the dialog at start (usually the field near which the "*?*" button was).

7.8 Products

- Similar to Clients.

- Price must be zero or higher.
- Stock cannot be edited and its automatically updated by the system when purchases or sales are committed.

7.9 Suppliers

- Similar to Clients and Products.

7.10 Discounts

In this window discounts can be specified.

- The discounted amount is the net amount of money that will be subtracted when the discount is applied.
- Products can be added by the means of the three buttons on the right panel labeled: “*Add Product*”, “*Edit Product*” and “*Delete Product*” respectively.
- The quotation or sale where the discount will be applied will be required to have at least the same number of units of the specified product in the discount.
- The table of products involved has three columns: *P.ID* that stands for product id, *Name* is the name of the product and *Units* is the amount of units of the product required.

7.11 Quotations

- *Client ID*: is the client to whom the quotation is being made for. The rest of fields from the client are loaded from the database as soon as a client id is introduced.
- *Sale*: equals to *null* if no sale has been made out of present quotation. Otherwise the identifier from the sale will be displayed and the quotation could not be deleted unless the sale made out of it is deleted in first place.
- *Discounts*: each row in this table represents a discount being applied to the quotation. The unit column is the number of times the discount is applied as they can stack as long as there are enough units involved. In this example the quotation has 3 products with id "1" and the discount with id "1" requires at least 1 unit of product "1" to be applicable. Thus up until 3 discounts with id "1" can be applied in this case.
- *Products*: each row in this table represents a product being quoted. Products can be added, edited or deleted by the means of the three buttons in the right panel labeled "*Add Product*", "*Edit Product*" and "*Delete Product*" respectively.
- Button "*Create Sale*": creates a new sale out of this quotation and links them both.
- Button "*Find Discounts*": look for discounts which requirements are meet by present quotation and apply them as efficiently as possible.
- Button "*Clear Discounts*": empty the list of discounts being applied.
- If the base price minus the total discount is negative, the total price will be zero, not negative.
- The table of products has 6 columns: *P.ID* stands for product id, *Name* is the name of the mentioned product, *PPU* stands for price per unit and is the cost of a single unit of the product, *Units* is the number of products being quoted, *%Disc* is an optional percentage of discounted that can be specified for the product, *Price* is the final price obtained from multiplying the *PPU* per the number of units and applying the *%Disc*.

7.12 Add/Edit Product

This is an example of a typical add/edit product dialog opened by a “Add Product” or “*Edit Product*” button. If the dialog is loaded by a “*Edit Product*” the fields will be filled with the information of the row selected in the products table it from the window that launched the dialog.

- Units must be a positive number.
- Discount is a net amount of money discounted from this product specifically and has nothing to do with the discounts object instances.
- When *Accept* is clicked, if everything is correct it will add a new row on the products table with the information we chose (or update the existing one if we opened the dialog by a “*Edit Product*” button).

7.13 Sales

- Similar to quotations but in this case the budget field represents the budget from which the sale was created. Otherwise its equal to *null*.
- When creating, deleting or updating a sale the products involved will have their respective stocks adjusted appropriately.
- Products can only be sold up to their existing stock quantity.

7.14 Purchases

- Represents products the owner of the system buys to its suppliers.
- Similar to quotations and sales.
- No discounts are available in this case.

8. Critical Appraisal

Overall I consider the project a success, however there are a number of aspects that presents room for improvement as well as others which have been decisive for the success of the project.

Regarding the research context, from the beginning the project was conditioned by a late start which cause we can find in an administrative delay. From my point of view there is nothing I could change in this aspect as the matter was out of my hands and I could merely wait for it to get resolved in order to get started. This incident greatly conditioned the amount of time available to develop the project and specially the amount of time available to choose the topic of the project and to design it. Had I had more time to choose the project topic and to research I could have developed the idea further in order to differentiate it more from other enterprise resource planning solutions. Moreover, I think that the project definition is excessively generic, specially regarding the operations available (add clients, products, create quotations, sales and such) and would have been more interesting if I

had chosen and more specific product, more customized to solve a less generic situation. Perhaps that would have differentiated even more this products from others available in the market.

Regarding the aims and objectives I am satisfied, I believe they were realistic and the final outcome has been accurate and approximate to them.

As for the requirements and specifications, I reckon more information in the form of diagrams would have been useful to speed up the design and coding but still I think that the present work in this chapter has been sufficient to accomplish the objectives without much trouble.

Concerning the development methodology I think it was a great decision to adopt an agile strategy, specially after the problems I encountered at various stages during the development where I had to change critical internal mechanisms because of my inexperience with the tools and language I was using. I believe if I had adopted a more formal developing cycle this changes would have had a much bigger and negative impact plus they would have noticeably increased the time required to accomplish the tasks and would have delayed the whole project.

Furthermore, regarding the system design I think it worked and covered the requirements with guarantees. Moreover, I reckon a more abstract and reusable design would have been inappropriate as a first attempt because it would have increased the complexity of a system that had yet to be tested to verify its viability. However, I think that once the first version was released and met all requirements the code could have been improved by refactoring it to make it more reusable, more abstract, to avoid code repetition and in general make it more clear and improve the quality of the code. The problem again was the time constraint imposed by the submission deadline and for this reason the code is not as clear as I would have liked it to be. This I believe is an endemic problem in software engineering, not unique to this project, because of the restrictions in time and budget imposed by the market and clients who only care about immediate results in expense of long term outcomes. From my point of view that is a mistake as I believe quality is more important than quantity in terms of engineering solutions.

With regards of the programming language I believe *Java* was the best option and I have no regrets in this matter. Perhaps some interpreted language like *python* could have also worked and I believe its code is easier to understand, generate and also makes working with data structures easier but then again, given the time constraint I think *Java* was the safest option and also minimized the requirements as the *Java Runtime Environment* is way more popular and common than *python*'s one.

In relation with the database, I think *Derby* was the best option due to its full compatibility and integration with the *Java* platform. The dilemma was whether to have it embedded or not and in the end I think it would not have made a big difference so I'm happy with the decision I took.

As for the integrated development environment, I doubted between *Eclipse* and *NetBeans* among other reasons because *NetBeans* is supposed to have a better tool for graphical programming in *Swing* technology. In the end I went for *Eclipse* because I had more experience with it and felt more comfortable using it and although at the beginning had some difficulties to install all the plug-ins I required, in the end I did not regret my choice as the tools available did the job perfectly.
About the GUI technology *Swing* I have no regrets nor opinion about the other alternatives as I never get to use them extensively.

In the chapter of software testing I wish I had developed a better framework to test the application

properly using *Junit [27]* or something similar. However, due to time constraints I did not have time to do it and had to test the application with black-box tests. This is something I definitely regret and will make sure to improve it and give this aspect more relevance on my next projects.

Furthermore, on a personal note, although I believe that the amount of work and hours I have put into this project has been appropriated I reckon that the distribution of them could have been greatly improved by following a more balanced and constant schedule instead of the more spaced in time working marathons I have performed. In addition, the activities and the environment that are implicit to the Erasmus experience did no help me either in this matter.

Finally, regarding the dissertations and reports I think I should have invested more time on preparing them in order to avoid the pressure I ran into in the end to finish them. Even more considering I had to write them in a different language than my mother tongue.

All in all, I believe I have probed I can successfully carry out a software project from the idea design to the final product and for this reason I consider this experience has been positive and successful. Furthermore, I have gained valuable knowledge in the process of doing it.

9. References

[1] Schildt, Herbert. *Java The Complete Reference,* 8th Edition, (2011).

[2] http://db.apache.org/derby/ The Apache DB Project. 2012.

[3] http://java.com/en/download/index.jsp Download Java Runtime Environment. 2012.

[4] http://windows.microsoft.com Microsoft Windows. 2012.

[5] http://www.linuxfoundation.org/ The linux Foundation. 2012.

[6] http://www.apple.com/mac/ Apple's Mac. 2012.

[7] Shields, Murrell G., *E–Business and ERP: Rapid Implementation and Project Planning*, (2001).

[8] Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J., *Agile Software Development Methods: Review and Analysis*, (2002).

[9] Royce, Winston, *Managing the Development of Large Software System*, (1970).

[10] http://www.eclipse.org/downloads/moreinfo/java.php Eclipse IDE for Java Developers. 2012.

[11] http://www.netbeans.org Netbeans IDE. 2012.

[12] http://www.jetbrains.com/idea/ IntelliJ IDEA.2012.

[13] http://www.eclipse.org/datatools/project_enablement/ Eclipse Enablement Project. 2012.

[14] http://www.eclipse.org/windowbuilder/ Eclipse WindowsBuilder. 2012.

[15] http://subclipse.tigris.org/ Subclipse Project. 2012.

[16] http://java.sun.com/products/jdk/awt/ The AWT. 2012.

[17] James Elliott, Robert Eckstein, Marc Loy, David Wood, Brian Cole, *Java Swing*, O'Reilly, (2002).

[18] Northover, Steve, *SWT: Implementation Strategy for Java Natives*, (2001).

[19] http://www.ibm.com IBM. 2012.

[20] www.eclipse.org Eclipse Foundation. 2012.

[21] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato, *Version Control with Subversion*, O'Reilly, (2008).

[22] http://docs.oracle.com/javase/6/docs/api/javax/swing/SwingWorker.html SwingWorker (Java Platform SE 6). 2012.

[23] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, Derek J. Balling, *High Performance MySQL: Optimization, Backups, Replication, and More*, O'Reilly, *(2008).*

[24] http://db.apache.org/derby/docs/10.0/manuals/admin/hubprnt43.html Backing Up a Database, Apache Derby. 2012.

[25] http://www.ibm.com/developerworks/data/library/techarticle/dm-0502thalamati/index.html An introduction to backup, restore, and rollforward recovery in IBM Cloudscape/Apache Derby. 2012.

[26] http://www.debian.org/ Debian, The Universal Operating System. 2012.

[27] J. B. Rainsberger, Scott Stirling, *JUnit Recipes: Practical Methods for Programmer Testing*. (2004).