Base Tutorial:
From Newbie to Advocate in a one, two... three!

BASE TUTORIAL:

From Newbie to Advocate in a one, two... three!

Step-by-step guide to producing fairly sophisticated database applications with OpenOffice.org Base, from initial problem to final product complete with forms and reports.

by Mariano Casanova

Base Tutorial: From Newbie to Advocate in a one, two... three!

By Mariano Casanova

First Edition: August 2010
Second Edition: September 2010

Please note that the author can not provide software support. Please contact the appropriate software developers of Base or HSQL at: www.openoffice.org and www.hsqldb.org or their fantastic fan base and forum experts.

The author has taken every precaution possible to ensure the correctness and appropriateness of the information provided in this text, including the testing of the code supplied. However, due to possible human or mechanical error from the sources, the constant changing and evolution of the software described and known and unknown issues in the code, its functioning and compatibility, the author can assume no responsibility for errors or omissions or for damages resulting from the use of the information provided here. The author does not guarantee the accuracy, adequacy or completeness of this information and shall not be liable to any person, legal or natural, with respect to any loss or damage caused or allegedly caused directly or indirectly by the use of such information, including but not limited to, business interruption, loss of profits or loss of data. The information is provided "as is" with no warranties whatsoever of its appropriateness or fitness for any purpose. You use this information at your own risk.

To everyone involved in the creation, distribution and documentation of free and open software

And in particular to the developers, writers and administrators at OpenOffice.org

~Thanks!

Content overview:

Part I: Things you need to know before you create a database with Base.

Where we introduce this tutorial and its scope. We start by analyzing what a database is and describe its different components: Tables, relationships, primary and foreign keys, columns, column attributes, deletion attributes and relationship cardinalities and finally we provide a definition of database and Base database. We later comment on forms and reports and on modeling data and goals of proper design, after which we provide an overview of UML diagrams to use as a visual vocabulary. We then summarily review phases in database design and the importance of Normal Form. We also review First, Second and Third normal forms and how they aid in class extraction. After this we review the way that Base records attributes and the nature of the variables it uses, analyzing text variables, numeric variables, date and time variables and object variables and how choosing them properly will affect the performance of a Base application.

Part II: Things you need to do before you code a database with Base.

Where we offer a real case example and, using the elements presented in part I, we start with a problem and end with a database design. We cover class formation, class extraction, the importance of atomization and descriptors. We apply normalization and other tools to decide on data structure and finally come up with a complete UML diagram of our database. We then stress the importance of using auxiliary elements like a Variables Definition List for the aid they provide in coding our application with Base. After this we analyze the problem of duplicity of roles in order to introduce the concept of super-class formation. We then analyze the forms and reports that we will use in conjunction with the design we have produced and describe the kind of features we will want them to have. Now that we understand how to design our tables, its connections, forms and reports, and what do the options that Base offers mean, we are ready to sit down in front of Base and start coding our database.

Part III: Things you need to do while coding a database with Base.

Where we describe how to use Base to create the database it took us so long to design, complete with forms and reports. We star by analyzing the setup and explain how to use SQL commands for the creation of tables and relationships, analyzing the “Create Table” and “Constraint” instructions in detail. We also review how to read their descriptions when consulting the Wiki or other sources. Then we describe in depth how to create forms to populate our tables and how to use radio buttons, sub-forms, list boxes and other widgets to simplify data entry. We later take our time to analyze the “Select” command to produce queries and extract information from our data and finally explain in detail how to produce the reports using both the Report Wizard and the SUN Report Builder.

Table of Contents

Introduction:

Databases are very useful programs that allow us to store information, organize it, retrieve it and even extract new information from it. OpenOffice.org offers a very powerful database system with Base. But because Base is a powerful and flexible application, you need to be able to make some informed decisions while working with it and this, in turn, requires some preparation.

This tutorial will try to help you better understand the options offered by Base while attempting to develop a functional application of a medium level of complexity. To achieve this, the first part of the tutorial will review some important concepts in the design of databases, that is, on how to organize the information that you need to collect. The later part of this tutorial will show you how to implement those decisions while developing an actual application with Base.

Parts I and II of this tutorial cover some fundamental notions for organizing your information that applies to any attempt to design a database, including Base; although they also describe some elements that are specific to it. Part III is solely focused on Base and the way to implement your decisions with it. However, the three parts are necessary in order to understand how to effectively design database applications with Base.

If you are reading this, chances are that you are a non-expert looking for answers to concrete questions and with very little time to spare. I hope that by following along these lines you will find concepts and tips that will help you design more useful, flexible and reliable databases. You can be sure that there is a learning curve, but it is not steep. After all, the whole purpose of Base is to make the development process easier. In any event, practice does perfect and mistakes are a problem only if you are planning not to learn from them.

This tutorial cannot cover all aspects of Base. Base has many features like the ability to be a front end for other database systems. We will focus on Base working with its own embedded database engine called HSQL. Even in this narrower topic we cannot cover all of its functionality. The text will attempt to provide elements that could help you build a working database model with which to keep track of resources and processes like customers, rendition of services and cash flow. However, I hope that this tutorial will provide a foundation from which to continue your exploration of this application.

Please note that this text has been arranged as a tutorial and not as a reference manual. This means that the information here has been organized thinking more on aiding the generation of meaning. I strongly suggest that you make notes in the margins and compile summaries from this text and form with them your own reference manual later.

Base is a great tool at your disposal and, the more you know about it, the more functionality and flexibility you can get from it. Don't hesitate to read everything you can about Base. Particularly, you will discover that there is a very active and knowledgeable com-

munity at www.ooforum.com, to which I am particularly grateful and where you can find answers to questions, post your own questions if they happen to be unique and maybe even share your own wisdom. Don't hesitate to participate!

I have written this tutorial as a way to say thank you to the open source community in general and the good folks involved with OpenOffice.org in particular. It is a lot of hard work to keep alive the ideals the open source software represents but is also most beneficial -and at several levels- for the rest of us. So hey, Thanks!

I hope that you find this tutorial helpful and that you start taking advantage of all the computational power that a database application like Base can provide.

Mariano Casanova
New York, Summer of 2010

Acknowledgments

Many people have helped me compile this text. I am very grateful of Jean Hollis Weber who thought that a tutorial like this was necessary and swiftly connected me to people who could provide relevant information. I am also grateful of Andrew Jensen who very kindly explained to me the different data types used by Base working with HSQL and their different memory requirements. Mr. Jensen also provided an early SQL code for this tutorial, answered all my questions on how it works and also tested the SQL code provided as examples in the tutorial.

I also found great support and very important information at the OO.o Forum -Base on several concrete topics and specific questions I had during the development of this text. In particular I am very appreciative of the help provided by Romke P. Groeneveld who kindly showed me how to use mathematical operators with queries and the casting of variables to other data types.

I also want to thank all the people who sent me words of encouragement and comments about the text or the code so it could be corrected or perfected.

Many people have offered their help proofreading this text and I want to express my gratitude to all of them, even if different circumstances (including a terrible earthquake) prevented us from working together. In particular, I am very appreciative of the patience and unassuming help by Ms. Judy Dimmick and the in-depth comments by Joe Smith.

Finally, many thanks to everyone who thought that this was a worthwhile project and gave me the encouragement to see it through.

Thank you very much!

Part I

Things you need to know before you create a database with Base.

Contents:

Chapter 1

Introduction to databases.

A database allows for the semi-permanent storage of data. Semi-permanent means that you can later correct or update the record but, until you do so, it remains just like you left it. Later on you can retrieve that piece of data and use it in a meaningful way. Databases are amazing beasts. I am sure that you know that you can store names and addresses in a database and later generate mailing lists and bulk mail or that you can keep track of your CD collection with them. This is already impressive. But you can do much more than that. Databases can help you identify the weaker links in your production process or rendition of services, they can help you identify where your public comes from and how they discovered you; they can help you keep track of the status of clients' orders and cash-flow and calculate payments for workers and fee per service professionals according to the number of hours worked, project involvement and differential hourly fee; hey! they can even help you learn about yourself. Basically, a database helps you collect and organize data[1] and then, if you ask appropriately, discover trends in your data that can help you organize your resources better.

The key concept here is "ask appropriately". To help you achieve this you need to make sure that your data faithfully represents the fact that you need to record, that you can find this data when you needed it and that you can use it with no restrictions or complications. This in turn is largely dependent on the way you design your database. So, the first thing that we are going to do in this tutorial is to describe some practices that allow for better design. We will start by naming the parts and bits that make up our database in order to make communication easier.

In the later part of this tutorial we are going to assemble a database of medium complexity with Base. That will be the ***how-to***. But in order to understand the options we will exercise then, we need to review the ***why*** behind our actions, which also rests on some concepts in database design. Let's get into this right away. Instead of just providing arid definitions, we will work a plausible example to illustrate the concepts.

Anatomy of a database: Tables, attributes and relationships.

Let's start with some general statements that we will then attempt to explain: A database consists of a group of tables that are interrelated. A table records information about objects that have the same characteristics. These characteristics are called attributes and are decided by you depending on the purpose of your database and the information you find necessary to collect and store. One attribute in a table can reference a record in another table, allowing tables to connect information in ways that are very flexible and powerful when you later want to retrieve this information from your database. We will examine

1 And there is a little bit of us in the data we find relevant to collect and the way we organize it

this in depth. When we say “database structure” we mean the collection of attributes you have chosen to build your tables with and the way these tables connect with one another. The process of deciding on a database structure is called “data modeling”. The heart of database design, then, consists of forming tables and deciding how to interconnect them. This topic will occupy the rest of part I and part II of this tutorial.

Let's imagine that you have collected over 10.000 books (I know someone who has done this!). Maybe you don't read them all but like to know that you have them, who wrote them, when and things like that. Memory will fail with a big number like this and keeping and consulting a written log can be very cumbersome as well. Enter Base and the help of databases.

A database is a way of storing information that can be easily retrieved later. You can retrieve a particular record (e. g. Who is the author of “Around the World in 80 days”?) or a summary of information (e. g. list all books by Jules Verne).

The information in a database is organized in tables. We will see later why it makes sense to use many tables in one database instead of just one big table. If you can find an example where just one table would suffice, then you might find it more straightforward to work it with Calc instead.

Tables are organized in columns (from top to bottom) and rows (from left to right). The columns represent different attributes that you want to store. For example, you can create the table “Authors” with the columns to store the attributes: First Name, Surname, Date of Birth, Country of Birth and Date of Death. Each row will hold one particular author, like: Jules Verne, Alexander Dumas or Pablo Neruda. Each row is called a 'record'. Each cell -the intersection of a row with an attribute- can also be called a 'record', just to confuse you. Rows are said to store “objects” (see figure 1).

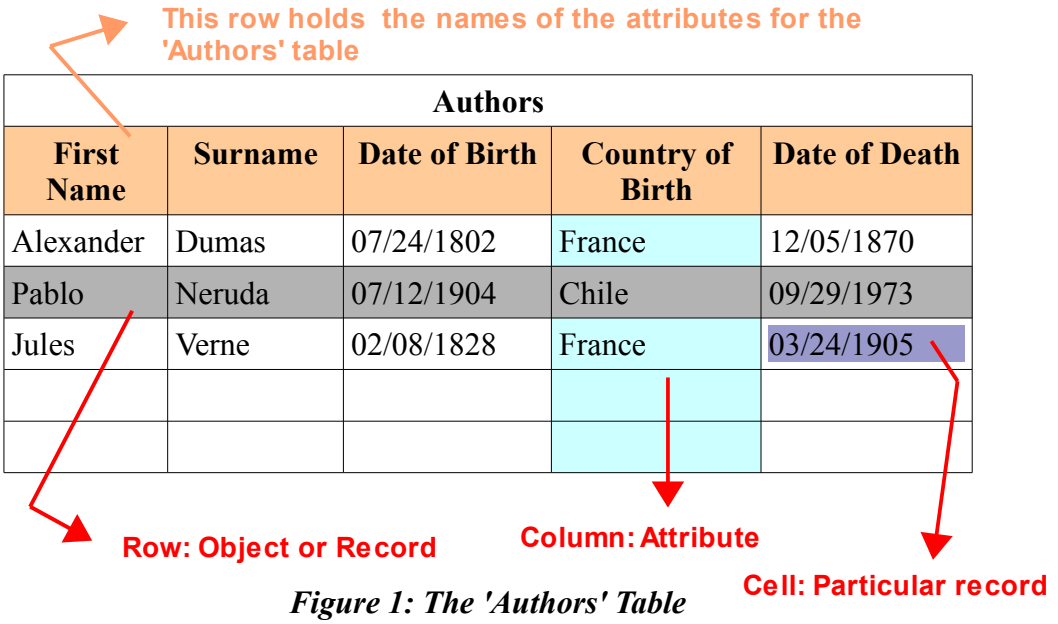


Figure 1: The 'Authors' Table

Notice this: some of the authors could not be dead but your table will not become obsolete if (or rather when) this happens. This shows that in designing your tables, which information you decide to include and which not, will have an impact on the overall usefulness of the database. Don't panic just yet, as we will be describing some systematic ways to decide on what data is relevant to collect. Besides, I am counting on your imagination and intelligence to sort this out.

Now that we can collect information about the authors, we want to record the books they wrote as well. Most of us would think about just adding more columns to record the books. For this you would need at least one column for each title assuming that you only record the name of the book. If you also want to record the year of publication and the country of first publication, for example, you would need three columns per title (see figure 2).

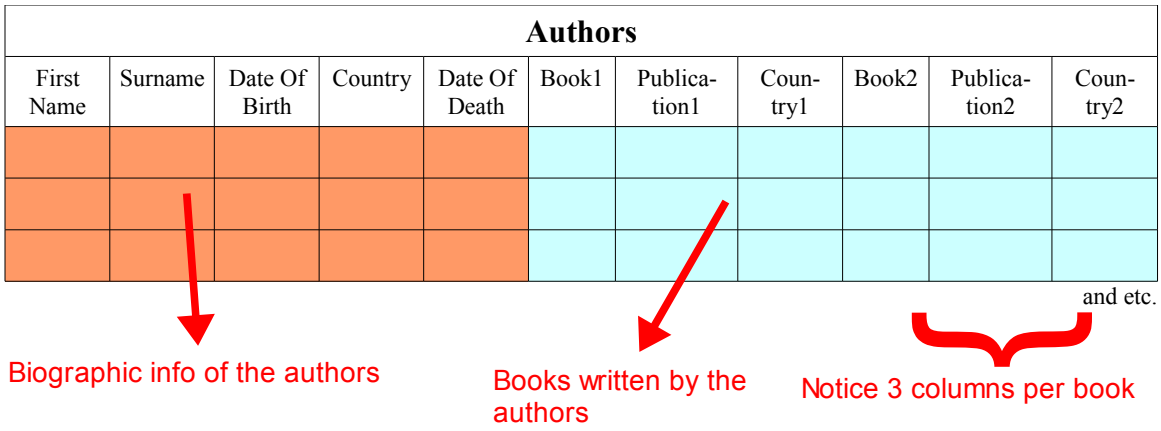


Figure 2: Unpopulated table for authors and their books

Let's say that each author has four or five titles, except for one that has twenty. You will need to create 20 columns (or 60!) if you want to store your data faithfully, most of which would not be used, wasting computer memory and speed. And what if this author later writes a 21st book? You will need to re-structure your database- adding new columns- and face unforeseen consequences that can come to haunt you later.

Instead of taking this route we could decide to focus on the books instead and create a "Titles" table, with the attributes we need for each book, and later add the columns for Author's name, surname and the rest. This way it would not matter how many books one author writes and we would not waste space with cells that would never be populated (see figure 3).

Titles						
Book Name	**Year**	**In Country**	**Author**	**Date of Birth**	**Country of Birth**	**Date of Death**
The Three Musketeers	1844	France	A. Dumas	07/24/1802	France	12/05/1870
The Count of Monte Cristo	1846	France	A. Dumas	07/24/1802	France	12/05/1870
20 Love Poems and a song of despair	1924	Chile	P. Neruda	07/12/1904	Chile	09/29/1973
One hundred love sonnets	1960	Argentina	P. Neruda	07/12/1904	Chile	09/29/1973
Around the world in 80 days	1873	France	J. Verne	02/08/1828	France	03/24/1905
Invasion of the sea	1904	France	J. Verne	02/08/1828	France	03/24/1905

Figure 3: A 'Titles' table with author information

Still, this would create new problems: Note that we need to repeat the author's data for every book the author wrote or writes in the future. This complicates maintenance of the data. For example, if one author dies, you need to locate every book she wrote in order to add that data. You also open the door for inconsistencies. What if some books have one date of birth for this author while others have a different day? One (or both dates) is wrong. Now you need to know which one is the correct day and find and modify each wrong record.

Instead of trying to merge both tables into one we will keep them separated, but we will find a way to connect the info in one table to the info in the other table.

This mistake, trying to create one big comprehensive table, is very common and arises because beginners want to make sure that all the relevant data is connected and forget that there are tools that can link them in more flexible and powerful ways.

This is your first lesson in database design: each table needs to cover one topic and one topic only. Creating tables that hold the attributes of both authors and titles is a bad idea. Creating tables that mix employee and department attributes is a bad idea. Creating tables that mix customers and services fields is a bad idea. Instead, you need to create one table for books, one table for authors, one table for employees, one table for departments, one

table for customers, one table for services and so on. If you are taking notes, write this down because I will ask you later.

Establishing relationships within tables.

How exactly do we link the 'Authors' table with the 'Titles' table? How do we make sure that, for example, the object 'Jules Verne' in the 'Authors' tables can reference 'Around the world in 80 days' in the 'Titles' table AND all the other books written by him? To accomplish this, you need to chose a field in the 'Authors' table that uniquely identifies each record. The chosen field that uniquely identifies each record is called a *'Primary Key'*. Now you create an extra column in the 'Titles' table that is going to hold the value of the primary key. This column, that stores the primary key value of the linked table, is said to store a *'Foreign Key'*.

For instance, you could decide that the 'Surname' field in the 'Authors' table will be the primary key. You then create a new column in the 'Titles' table -that you can call 'author' if you like- and write the surname of the author with each book record, correspondingly, in the column that holds the foreign key: Verne, Dumas, Neruda, etc. This 'author' column in the 'Titles' table unequivocally links each book with one author (see figure 4). Base will use that index to link and retrieve the information that you will be asking for later.

Titles			
Book Name	**Year published**	**Country of Publication**	**Author (FK)**
The Three Musketeers	1844	France	Dumas
The Count of Monte Cristo	1846	France	Dumas
20 Love Poems and a song of despair	1924	Chile	Neruda
One hundred love sonnets	1960	Argentina	Neruda
Around the world in 80 days	1873	France	Verne
Invasion of the sea	1904	France	Verne

Figure 4: 'Titles' table linked to the 'Authors' table through the 'Author' foreign key

This way, for example, you could ask your database to list all “Book Name” where “Author” equals “Neruda”. Although you had not explicitly recorded such a list, the database can create it for you. Imagine how useful this is if, instead of six records, you really have ten thousand!

Using a surname as a primary key works fine many times, but has one drawback: what happens if you have two authors with the same surname?

One solution is to use more than just one field to form the primary key. In this case it is said that you have a *compound primary key*. For example, you could use the first name and the surname, together, to form the primary key. This solution is possible and perhaps in many cases the right one. Of course, we can also think of instances where two authors have the same name and surname (like Alexander Dumas, father and son, for example). We could extend the notion of a compound key and work with a combination of three and even four fields to form a unequivocal primary key, but think of all the calculations the computer would have to do just to handle rare exceptions.

To simplify this, Base can automatically generate a unique number for each record in a table. This way each author would have a unique number (Dumas father would have one number, Dumas son would have a different number) that you can record in the column for the foreign key in the 'Titles' table. Instead of writing 'Verne', 'Dumas' 'Neruda', the computer will write 003, 004, 005 or something like this. You, as a user of the database, might not even be aware of the existence of these numbers or what numbers they are exactly. The relevant thing, however, is that these automatically generated numbers allow for a unequivocal connection between one object in the 'Authors' table with one or more objects in the 'Titles' table.

At other times, however, you could be using primary keys that are not numbers or primary keys that are compound. Keep this in mind.

Our example uses only one foreign key in the 'Titles' table, but don't be surprised if you find yourself needing to record two or more foreign keys in a table. This is how tables are related!

The primary key is important in more ways than one, and we will check this when we review the process of Normalization.

Cardinality and optionality of a relationship.

Notice this: in our example one author will have at least one and possibly many titles. On the other hand, one title can have exactly one author only (let's leave collaborations aside for now. I promise to include them by the end of this section). When you say : “one author, many titles” you are talking about the cardinality of the relationship.

Note again that the relationship is not the same for authors and titles: one author can have many titles, titles have exactly one author. So, when we analyze the relationship from author to title, the cardinals are: 1..n (one to many). When we analyze the relationship from title to author, the cardinals are: 1..1 (one to one). Strictly speaking then, relationships always have two sets of cardinals.

The first number of the set is called “Optionality” and is usually a zero or a one. In the case it is a zero it means that a member of the class has the option to not relate to the second class. For example, if the cardinality from title to author had been 0..1, that would have meant that you can register a book even if you don't know who the author is. Here

you have something extra to think about: whether your database needs, or needs to avoid, objects in one table that are not associated to objects in the other table. Of course, both options are valid and which one you choose depends on the way you define their relationships.

The options for cardinality include:

- ➔ Zero to one (0..1)
- ➔ Zero to many (0..n)
- ➔ One to one (1..1)
- ➔ One to many (1..n)
- ➔ Many to many(n..m)

Zero to one implies the possibility that an object of the class is associated with none or at most one object of the other class. **Zero to many** implies the possibility that one object in the first table is associated to no object in the second table, to exactly one or to many. The zero optionality opens the possibility for the existence of an object even if is not associated to other objects in other tables.

One to one cardinality connotes properties of objects in one table that are expanded in the second table. Let's say that some of your books have beautiful illustrations -paintings, engravings, photos, etc- and others don't. You would want to register who was the artist, the name of the art piece and other data. It would be wasteful to include these attributes in the 'Titles' table, because most books would leave these records unpopulated. Notice the Set-subset relationship here. You have a set: books, and a subset:books with illustrations. You record the attributes common to all in the 'Titles' table and then create a new table -"Art info" for example- where you record the extra attributes of the subset. Oh! And of course, the foreign key. Every time you have a set-subset situation a 1..1 cardinality comes in handy. Note that if this cardinality were the same at both sides of the relationship (in the direction subset-set) then maybe both classes are really one big class[2].

A **one to many** cardinality implies that one object of the first class can relate to one or more objects of the second class and that an object in the second class can not exist by itself.

Many to many relationships can not be performed without the use of an intermediate table. For example, one author could write one or more books; at the same time, one book could be written by several authors (in collaboration). In order to keep track of this, you will need a simple table -maybe with only two columns- between the 'Author' and 'Titles' tables that can record all the combinations of book and author. This will change the n..m cardinality to a manageable 1..n or so. Every time you encounter a n..m cardinality, you know that you will be using an intermediate table.

2 Also note that the one optionality precludes having information about illustrations if it is not associated to a book.

Managing relationships with column and delete options.

In order to enforce and make tidy the inclusion of primary keys, foreign keys and the cardinality of the relationships, Base allows you to specify certain options for the columns in your tables. In this tutorial we will consider the following ones:

Key: This option tells Base that this column will hold the primary key of the table. Base will prepare then to see it associated with other tables.

Unique: When you specify this option, Base will make sure that records in this column are not repeated. If, for example, you specify that the 'surname' column be unique, then the second time you try to enter 'Dumas', Base will reject it as an invalid entry. It makes a lot of sense to make sure that a column set to KEY is also set to UNIQUE.

Not null: This option means that records cannot be left with this attribute empty. Base will display an error message if you try to enter a record that leaves a NOT NULL column empty. This forces whomever is using your database to at least have the information requested in the NOT NULL columns if they want to enter the record. For example, if you set the 'surname' and 'date of birth' as NOT NULL, then a user can not input a new author if he doesn't have at least the surname and the date of birth. Again, it makes sense that Key columns are also set to NOT NULL. This option can also affect cardinalities. If you have decided that your database should be able to accept a book entry even if it is not associated to an author, you can not set the foreign key to NOT NULL. On the other hand, if you will not accept a book entry unless it is associated to an author, the foreign key should be set to NOT NULL in order to enforce this.

Because being able to relate the object in one table to another object in another table is so important, special care must be placed on the subject of deleting records. Think about this: Let's say that there is an author that wrote only one book and you discover that you no longer have that book in your collection. As you update your database and erase that title, what will happen to the 'author' information? Should it be deleted too? Should it be kept as an historical record or in case you find and buy one of his books again?

Actually, both options are valid and you can chose the one that reflects the purpose of your database better. But your application will not know what to do unless you make explicit what your preference is. For this reason, when you are developing your application and defining relationships, Base will need instructions on how to handle the deletion of records and will offer you the following options. This is what they mean:

No action: Base will delete the record you want but will not delete the records associated with it. This way, you can delete the missing book and keep the record of the author that wrote it.

Delete Cascade: With this option, Base will delete the record you are requesting to delete and will also delete all other records that have this record's key as a foreign key. This option is called cascade because it elicits the image of a deletion creating further deletions.

Following the example, if you delete the author, any book associated to him will be deleted too.

Set Null: With this option, Base will delete the record you are requesting but will not delete the other records related to it. Instead it will erase their foreign keys to reflect that they are no longer associated to other objects. Note that this requires that NOT NULL is not a condition of the foreign key column. If you decide to erase the author in the example, the book record would not be deleted but you would find that it no longer has data for the foreign key, i. e. It would be set to null.

Set Default: When deleting an object, the foreign key column of the associated tables will be populated with a default parameter that you previously specify. This way, instead of just leaving an empty foreign key in the book record, Base could write, for example (and this is completely arbitrary) "000", which you know means "I have no author info for this book in this database".

A definition of database and database design.

It took some time, but now that we have described a database and its properties, we are ready to offer a definition of Database and Base Database.

We have seen that database design uses many tables that are all related one way or another. Each table will only cover one particular topic or unity (authors, titles, places, events, etc.). These topics are called classes. A class is a collection of objects that have the same attributes. In database theory, each class translates to a table.

With that said, we can attempt to define a Relational Database as a collection of objects -organized in classes- and their relationships[3].

This definition didn't take long to write but you won't be deceived by this illusory simplicity. You can appreciate that 'object' has a rather precise meaning and is a central element in the conformation of your classes and that there is a big chunk of knowledge around the concept of relationship, particularly because of its fundamental role in connecting information.

Database design, then, is about deciding on class structure (which classes to work with and which attributes to record in each) and the structure of connections they establish (which table connects with which and with what cardinality). Of course, you also need a method to add records to your tables. And you need a method to retrieve useful information -like particular records or summaries- to produce reports. Base offers forms, queries and reports for this. For this reason, when we say a 'Base database', we mean not only the data and their relationships but also the forms, queries and reports to use it.

3 Just to be clear, the name "Relational Database" derives from the use of the word 'relation' which is a way used in mathematics (Set Theory) to say table. In consequence, when we say 'relational database' we mean: 'database that uses tables', which is a distinct feature when we compare this to other database models (e.g. Hierarchical, Network, etc.) and is not an emphasis on our intent to *relate* -or connect- data.

Talking about forms and reports...

Forms allow you to enter data to populate your tables. With Base, you can build them by using the Form Wizard, which simplifies the task, or in design mode, which gives you maximum flexibility. When building forms you will need to think about what you want them to achieve and this in turn requires that you understand what do the different options offered in building them mean. We will review this more closely in part III. Not surprisingly, the time you spend thinking about the design of your forms will be paid back with better performance and efficiency.

Among these considerations, you need to make sure that the user of your form understands exactly what she/he is being asked (e.g. To the entry “Sex:” do you input “Male” or “Scarcely”?). It is not uncommon that we use words believing that they have a very precise meaning but later discover that they can be ambiguous. That is because when we use a word, it correlates strongly with an image in our mind, but our readers could have other associations. The context in which they appear can also suggest meanings that were not intended. To avoid traps like these you need to test and test and test your forms, asking friends to read them and give you feedback about what they understand and how they think they should answer. Even if you think that you will be the only one using your forms, you never know when someone will show up to help you with data entry or when someone will ask you for a copy of your application to use it herself.

The other important aspect of forms is that you want to ensure that data entry is uniform, that is, that the same element is entered in the same way every time. In general, Base will consider as different entities two spellings of the same word. Also, if you are not consistent with the use of capitals, Base can consider 'Verne' and 'verne', for example, as two different objects. This could lead to wrong summaries of data or records impossible to find. Base offers a type of variable that treats words with the same letters as the same word, no matter how you capitalize them. This can be very helpful. However, if you are keeping customer data, you would not want to have them receive mail with their names carelessly treated, as in “Dear Mr. SMith”, so you still need to monitor input.

Base offers several ways to handle this. One of my favorites is the drop-down menu, where you point-and-click your entry. You can also use check lists and circular buttons. What they have in common is that they automatize the entry of data. Base also offers functions with which you standardize data entry, for example, changing all letters to small caps (as in “proCEduRe” to “procedure”). Of course, this means more work while you are developing your form, but Base makes it really easy and it's worth the extra effort.

For data output you have the Reports. They are usually built by queries and you also have the options of either getting help from the Wizard or using the full flexibility of Design Mode. We will also spend time in part III studying this. What I want to point out is that reports need to be easy to read, unambiguous and provide the necessary information. This means that spending some time in their design is also bound to repay you later with added efficiency. For example, I always recommend to include time data in a report so we can be sure about when the information presented was valid.

So get as many people as you can to test your forms and reports. At the same time, test your database design. See if someone can fool your input options or find exceptions that are possible but that your design can't handle. Test, Test, Test. It is easier to make changes at earlier stages than trying to repair a live version.

Modeling data and goals of proper design.

In designing your database, you will need to define the classes that you will be working with (books, authors, etc.) and what relationships within them you need. Then you need to decide on the structure of each table, that is, what attributes you need to record for each class (name, surname, etc. or title, year of publication, etc.) and what properties you will give to the columns (not null, unique, etc.). You can't attempt to collect ALL information about your classes and you will need to edit your selections according to the **purpose** of your database. You will then need to refine the structure of connections, deciding not only what table connects with which, but also the cardinality and optionality they use, the type of primary keys that you choose and how to treat deletions. These decisions might also influence, or be influenced by, the order in which your forms ask for the information and the reports that will make sense to produce. When you are deciding all this and fine tuning your decisions, it is said that you are modeling your data.

Don't feel dismayed by all this. The subject of database design is vast and complex enough to comprise several years of college education. So it is going to take some time and patience for you to feel comfortable with these concepts and the ways to implement them. But you don't need to be a rocket scientist to become quite proficient at working with Base. The description offered in the previous paragraph is quite a good summary. Read it and translate it to a numbered list and then try to visualize your actions at each step. This is exactly what we will do in part II of the tutorial.

One of the most important aspects in data modeling is defining (also said 'extracting') your classes. We will review two ways for doing this. The first is a consequence of your activities during the different phases of database design, which we will describe shortly. The second is a formal method called Normalization. Normalization is a formal method because it does not analyze the content of your tables (it does not care if the classes are books or authors or if the attributes are names or dates, etc.) but focuses on certain mechanical connections of the primary key. There are several normal forms and, in this tutorial, we will review three: first, second and third normal form. In your work you will be doing both, content analysis and formal analysis.

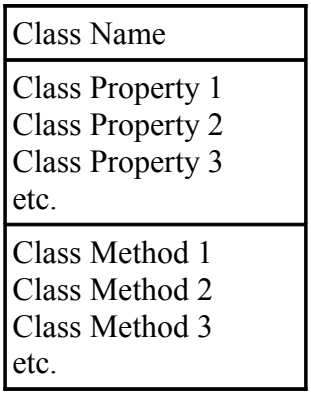
There is no one way to design a relational database. Your design will reflect your level of preparation and experience. The techniques we will review here are only tools to help you make better decisions. In general, however, you will want your database to avoid repeating information while still recording all the data you need; that your database can grow, as your collection of data grows, without a breakdown in functionality, and that your database has some flexibility to handle entries that are not common but possible. Principally, you want your database to provide the information you need and that such information is valid for the sample from which it was extracted.

In order to ensure this, modeling experts talk about Entity Integrity, Referential Integrity and Data Integrity. Entity Integrity means that each row of a table represents only one entity in the database. Therefore, each entity can be assigned a unique (and only one) primary key. Referential integrity means that all foreign keys in all tables must be matched by a row in another table (obviously, a primary key). Finally, Data Integrity means that all data in the database correctly represents the fact that it aims to collect. This is achieved basically by minimizing data entry error -you know- misspelling of names, repetition of characters, transposition of numerical digits, wrong or impossible dates and a long etcetera. Throughout, we will be reviewing ways to achieve these goals of proper design.

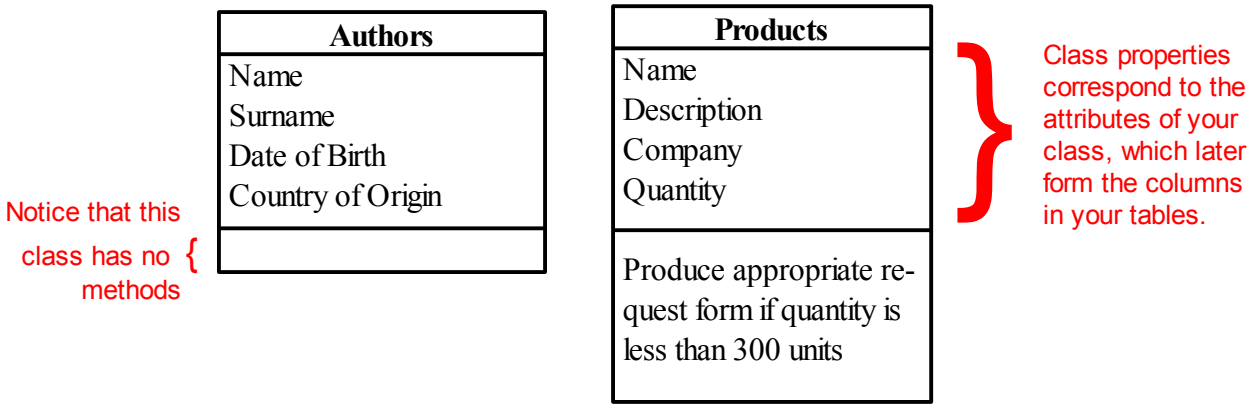
Visual vocabulary.

Database design is aided by a diagram protocol called UML, acronym for Unified Modeling Language. UML provides a standardized way to signify components or aspects of the Database. Let's see this:

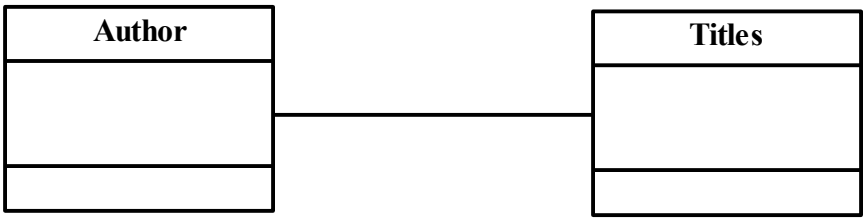
Classes are often notated this way:



See the examples:



The relationships are commonly notated by a line or arrow between the classes:



Because in the design of a database you are mostly concerned with the classes and their relationships, these two elements will suffice for now to describe, think about and communicate with others the structure we give to our data (That is, the structure of our classes -which attributes we will want to record- and the structure of connections -what table connects with what table and with what cardinality). UML offers other elements but we will not review them here.

You can imagine that, in the process of fleshing out the design of your database, you are prone to spend a lot of time drawing boxes, connecting them with lines, deciding on cardinalities and deciding which attributes to include in each class. It's funny to notice that this very important phase of database design is best aided by a simple pencil and paper. However, this is very powerful. We have already hinted at the fact that you will be working with several tables and the more thought you invest in the design, the more flexible, reliable and functional it might become. Changes in design now are just a matter of erasing a line and drawing a new one or deleting or adding attributes instead of trying to undo part of your work in front of the computer while trying to retain other decisions at the same time. With the level of abstraction offered by UML, you can focus on deciding the general structure of your database and deal with the details later.

Let me note that Draw offers a flexible set of tools for drawing boxes and connecting them with arrows. Draw can even let you easily adjust the arrows, for example, if you have several boxes and need to surf the connections between them. You can also find other freeware for doing this, like Dia (a GNU license project) that comes with a complete set of UML tools.

Later, when we discuss the database we will work with in part II of this tutorial, you will see how our design decisions are represented in the diagrams, how this simplifies the communication about the design and how this translates into the development of the application.

A more formal rendition of your design should include the names of your attributes in the corresponding classes and the identification of the primary and foreign keys. The relationships should indicate the cardinalities and the arrows should go from the primary key to the foreign keys (figure 5).

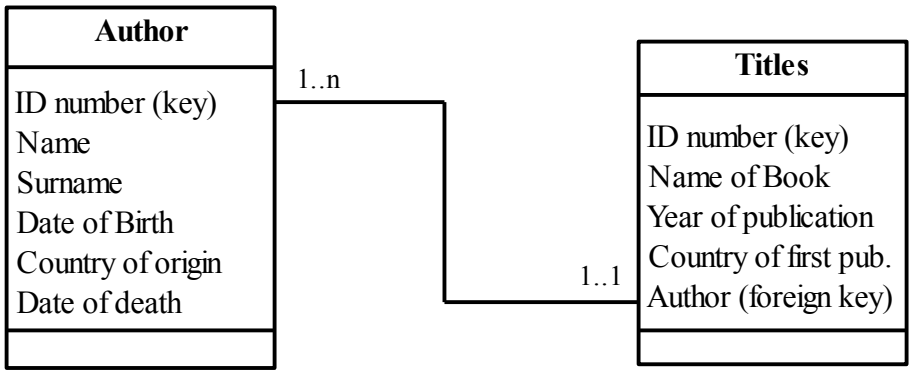


Figure 5: UML diagram for the Author and Titles database

Phases in database design.

In general, the effort to design a database will start with someone finding a problem in the real world. It can be a dentist trying to organize his clients and lab work or a company trying to keep inventory of on-line sales. This person will need help from someone with knowledge of database design and in implementing it with Base. Let's call the person a 'client' and the expert the 'developer'.

Of course, the client and the developer can be the same person -and if you are reading this, then this is most probably the case. But I make the distinction because a dialog between the client and the developer is always necessary, even if it only takes place inside someone's head. Usually, the client has a general idea of what he wants and the developer will ask questions in order to be able to translate the need into a well developed database application.

Broadly, the phases of database development can be described in the following way:

1. Presenting problem is identified
2. Possible solution is defined
3. Data modeling
4. Model testing
5. Database running and maintenance

During the first phase, the client tries to explain what his needs are -which are often multiple- and the expert tries to help him narrow it down to the kind of tasks a database can really perform. In the second phase, the expert develops a formal statement of the goals and scope of the database application. This is important because a small change in a goal can produce important changes in the design of the database. Besides you will discover yourself becoming more ambitious with what you want your database to do. If you don't set a line somewhere, you can find yourself making your model progressively more and more complex and never actually develop your application or, worse, have it perform erratically as you keep modifying data structure.

Now that, as the expert, you have defined the scope and goals of the database, you are in the privileged position to decide which information is relevant and which not. You can make better decisions that can help define classes and attributes, primary keys and relationships with zero or one cardinalities. This is the extraction of classes we earlier identified with **content analysis**. You can also sketch and consider the forms and reports that your application will need and how they integrate with the design. Of course, you also have to test your logic and grammar by asking friends to find holes in your design and meaning in your reports and forms. These two phases do not have a clear boundary because the testing will impose changes in your design and the new design will need to be tested. Don't mind spending some time here and test, test test!

Now that you feel confident of your database design (class structure and structure of connections) and the forms and reports you want to use, it is time to code them with Base. Now you are sitting in front of the computer for the first time since the project started! Of course, if you are in the process of learning then I strongly encourage you to sit and play around with Base, the more the better, so that you can become acquainted with its different screens, dialog boxes and options. But if you are in the task of designing a database, this is the first time that you really need to interact with the computer.

During maintenance, you test that your database remains consistent and able to handle the growing number of data it receives. Defragmenting your database and making backups that could get you up and running should disaster ensue seems appropriate. Now is when you can also try to introduce the features that you didn't think of before and that can make your application run faster or more efficiently. Of course, these changes need to be permitted by the structure of your database. When your structure can no longer accommodate the new functionality that you want, then maybe it's time to go for a new design.

The more that you, as the expert, question the client and become familiar with the business process, the better your decisions for class extraction and table structure. Usually, the actors involved in the productive process (customers, providers of service, suppliers, investors, etc.) become classes. Different phases in the production processes (recorded as schedules or logs), places (e.g. the different stores of the chain) and even events (e.g. rendition of services) can also become classes. Some data can be better recorded as tables; some other times you will find that the same data is better recorded as attributes. The difference is set by the purpose (functionality) of the database. Here is where imagination and experience come in very handy. Try out your ideas and see what happens. Read about database design and don't fear to ask other people (especially if they know something about database design!).

The other tool for deciding on class extraction is Normalization, which deserves a chapter of its own.

Chapter 2

Getting into normal form.

Normalization is a formal way to ensure that our tables have a good structure, that is, that each table has the appropriate attributes.

Typical problems avoided with proper normalization include:

- ➔ Unable to create a record without information from a foreign record.
- ➔ Deleting one record will result in other data being lost.
- ➔ Data being repeated in other fields (or tables) resulting in possible inconsistent records of data.

Many of these problems have to do with deciding if certain data should be organized by itself -in a table of its own- or as fields in a host table.

Because this is a formal method, we will not deal with the actual content of our tables (which is what we are supposed to do the rest of the time anyway) but will analyze certain mechanical connections of the primary key.

A key concept of normalization is that of 'Functional Dependency'.This concept means that there is one field (the primary key) or more fields (for a compound primary key) of a record that is/are enough to help me identify any other field (attribute) of that record, that is to say that knowing the value of any field **depends** on knowing the value of the primary key.

Clearly, true primary keys help me identify -or determine- all other attributes in a record (row).

This also means that, if we have a compound key and one of the fields in it turns out to be superfluous or redundant, then this primary key is not a true primary key, it's just a key. Therefore, a primary key has no subset of fields that is also a primary key.

Many times the vastness and complexity of the data in our database obscure this leanness, potentially creating problems like those described above. By analyzing functional dependency -at several levels- we can identify if our data structure is compromised or not and avoid those problems. There are several normal forms and in this tutorial we will cover three: first, second and third normal form.

First normal form.

According to this rule, *a table must not try to keep multiple values for a piece of information, whether in one field or in multiple columns*.

We would have this situation if we tried to include in the 'Authors' table all the books written by them. Let's say that we create a column called 'Books' and then we cram together 'The Count of Monte Cristo', 'The Three Musketeers', 'The Man in the Iron Mask' and etc., all separated by commas, for the record of A. Dumas. Even if we use several columns (Book1, Book2, Book3, etc.) we are still trying to cram multiple values (the name of the different titles) for one type of information (Books written by author).

We already saw other reasons why this is a bad idea. First normal form helps us identify this problem. It also offers a generic solution: If a table is not in first normal form, remove the multivalued information from the table and create a new table with that information. Include the primary key of the original table as a foreign key.

In our example, this would result in the creation of the 'Titles' table, with info on the books, and a relationship to the 'Authors' table.

Although not immediately evident, first normal form solves a problem of functional dependency, as the authors' primary key would have not helped to identify a particular value for the multivalued field. If I ask you to provide a concrete title for a particular author, you have no way for selecting it.

Thinking in terms of extracting multivalued pieces of information is very simple and completely equivalent to the analysis we did in the earlier chapter with this same example.

Second normal form

Second normal form requires that a table be in first normal form AND that *all fields in the table (that are not primary keys) can be identified only by the primary key and not by a subset of it*.

By definition, this problem can arise only when we have a compound primary key to identify a record. To explain this, let's analyze the following table:

Teacher ID	Project Name	Department	Contact	Hours
mackormac032	Science Fair	Science Dept	344-2713	10
Phillipi72	Soccer coach	Athletic Dept	225-5902	18
mackormac032	Art Contest	Art Department	639-6798	7
Phillipi72	Science Fair	Science Dept	344-2713	15

Is this table in first normal form? Yes, If I tell you the primary key, you can identify unique values for the other fields.

In this case, the primary key is a compound key that requires the Teacher ID and the Project Name. I need both bits of information to know, for example, how many hours have been devoted by a certain teacher to a certain project.

But it is also true that I only need the Project name if I want to know the involved department or the contact information. Oops!

So, while I am thinking that Teacher ID and Project Name form the compound primary key, it turns out that a subset of it -the Project Name- is a primary key for certain fields in this table. To solve this, we need to extract the info related to the subset primary key (Department and Contact) and form a new table with them, including the subset key they depend on (Project Name).

This process is called 'Decomposition'. Decomposing the table of the example we get:

Teacher ID	Project Name	Hours

and

Project Name	Department	Contact

Note that the original table retains the entire compound key (Teacher ID and Project Name) and the info that functionally depends on it (Hours).

Lets restate this in formal parlance:

> *A table is in second normal form if it is in first normal form and we need all the fields in the key to determine the value of non-key fields.*
>
> *If a table is not in second normal form, remove the non-key fields that do not depend on the whole of the compound primary key, then create a table with these fields and the part of the primary key they do depend on.*

Third normal form

Third normal form requires a table to be in second normal form AND that you can not use regular fields (that is, that are not key fields) *to determine other regular fields.*

In other words, you need to make sure that only key fields determine non-key fields.

To make sense of this, let's analyze to following example:

Teacher ID	Surname	Name	Department ID	Department Name
fsmith089	Smith	Fred	17	Science
mling619	Ling	Mei	17	Science
syork233	York	Susan	18	History

If each teacher works for only one department, then this table is in first and second normal form. Let us analyze this:

1. If I know the teacher's primary key (the Teacher ID) I can tell you a unique value for all the other fields.
2. The primary key is not compound, so I have no possibility of subset conflicts.

However, notice that there is information about the department that is repeated (department ID and Department Name) and could become inconsistent (e.g. It seems that the Science Dept. has been assigned the ID number 17. If we later find a record that assigns the number 23 instead, we would have an inconsistency). Further, it is possible to determine the Department Name by knowing the Department ID, which is not a primary key, and *vice-versa*.

In this case, we *remove the regular fields that are dependent on the non-key field* (In this case, we remove the Department Name) *and create a table with it, in which we also include the field they depend on* (the Department ID), *left as their primary key.*

This way we would have:

Teacher ID	Surname	Name	Department ID

and

Department ID	Department Name

In formal parlance we have:

> *A table is in third normal form if it is in second normal form and we cannot use non-key fields to determine other non-key fields.*
>
> *If a table is not in third normal form, remove the regular fields that are dependent on the non-key field and create a table with them, including the field they depend on as their primary key.*

Aiming for simplicity

An analysis of functional dependency can easily identify decisions in table structure that will create problems sooner or later. Notice that some of the decisions made with Normalization had already been implemented with content analysis. This shows that both tools have an area of overlap. This is fine. Normalization can help us simplify structure when we are very in love with the table designs we have done and it hurts us to chop them further. And believe me: simple is better!

However, in the process of designing your database, you will be using both tools: Normalization and content analysis i. e., the extraction of classes by understanding the production processes, the key roles involved, the required reports and the overall purpose of the database.

Chapter 3

Recording attributes: Are you my variable?

The data that you will store in your tables will actually be stored as variables in your computer. Computers have different ways of storing data. In general, they trade memory or speed for accuracy: Computations that require more accuracy tend to be slower and use more memory. When you are building your tables with Base, you are asked what kind of variables you want to store and are presented with a drop-down menu of options. The decisions you make will affect the performance of your database. To better understand what these options mean and how they affect the way Base handles your variables, it is necessary to review the way computers handle data.

In general, computers handle numbers and characters differently. Some people are confused by the fact that numbers can be treated as characters also. e. g. '7' is a sign that represents the concept of seven. In that sense, it is stored in the same way as the signs '@' or '#' or the letter 'A'. As a character, the sign can not be operated mathematically, just as it would not make sense to calculate the sum of '@' and '#'. Phone numbers are good candidates for being stored as characters. Of course, any information that uses text is also stored as characters. Addresses need both, numbers and text, but we store them all as characters. Whether they are letters or numbers, when we are only storing the signs (as opposed to the value for a number) we call them 'Alphanumeric characters'.

Computers have different ways of storing alphanumeric characters. For example there is the ASCII code, that needs only one byte to store a character. Unfortunately, this limits the number of possible characters that you can use to only 256. Although enough for many applications, it falls short if you want to have access to expanded character sets: Chinese or Arabic characters, accented vowels (diacritics) and things like that. Protocols that allow for lager numbers of characters have been developed, like Unicode, that use more bytes per character.

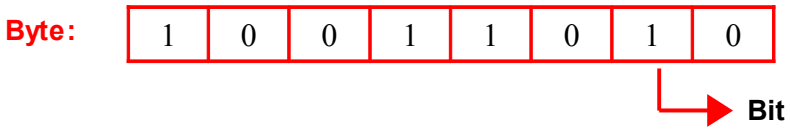
Base will store alphanumeric characters using UTF-8, which is a code that is compatible with both ASCII and Unicode. Base will use one or more bytes for each character according to internal calculations. When Base asks you the length for a particular record, e.g. Surname of Author, it is not asking the number of Bytes you want to allocate but for the number of characters you want to receive. How many bytes are actually used is fixed by the software.

This is not the case when you store the value for a number. Different ways of storing numbers will require more or less bytes. Let's see why:

Computers store information in binary form. The unit is called a 'bit' and can have one of two values (thus binary): either zero or one (or On and Off, or 3.5 volts and 5 Volts, or

any of two possible states). Bits are organized in larger groups -usually eight bits- to form a Byte.

In this conceptual representation a byte is drawn as a box with eight spaces. Each space can hold a zero or a one:



Since this is a binary numeral system, each box represents a power of two, in the same way our decimal number system assigns successive powers of ten from right to left:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Which is to say:

128	64	32	16	8	4	2	1

To represent the number five, you would need to 'activate' or indicate the numbers for four and one (because four plus one is five). So your byte representing five would look like this:

0	0	0	0	0	1	0	1

= 5

With a little bit of math you can figure out that you can represent all numbers from 0 to 255 (that is 256 numbers in total) using one byte.

If you need to store numbers bigger than 255 then you need to add extra bytes. Base offers different amounts of byte aggregate options to store numbers, depending on how potentially big the values that you need to record are. Check the chart below to get and idea. Now, what if you need to store negative numbers, say -7?

The solution for this has been to use the last bit to the left of your byte to represent the sign and use the other seven bits to represent the number. Because the last bit no longer represents the value 128 but instead works as a flag to indicate if the number is positive or negative, we can represent numbers form -128 to 127 with this arrangement[4].

This type of bytes are called 'signed' bytes, while the bytes that only store positive numbers are called 'unsigned bytes'. The chart below indicates which options are signed or unsigned and helps you know the range of values you can store with them.

4 The binary negative numbers are deduced by a technique called *two's complement* where 10000000= -128 and 10000001= -127... 11111110= -2, 11111111= -1, 00000000=0, 00000001=1, 00000010=2... 011111111=127 and around again. This way the machine can subtract numbers by adding a number with the two's complement of the other number and not have to attempt really complicated alternative ways of subtracting.

Again, if you need to store values beyond these boundaries, you need to add more bytes, which uses more memory per record. In any event, only the leftmost bit, also called 'Most Significant bit' (MSb) will act as a flag, that is, if you are using two bytes, the MSb will act as a flag and the other 15 bits will store numbers. With this information, you can calculate the range of such a variable.

In general, **Numeric data variables** are described by the number of bytes they use and whether they are signed or unsigned. These two factors determine the range of possible values they can hold. Base offers several types of numeric data variables, both signed and unsigned and using different amount of bytes.

At the least memory consuming side of number storage we have the Boolean numbers. A Boolean number is in fact just a bit, and we use it to store YES/NO type of data, like and answer to the question 'have you ever been to Hawaii? At the other end there are variables called 'floating point numbers' (just 'Float' to family and friends) that allow us to store numbers that have decimal places like 1.618033... They are the most memory consuming numbers but the only ones that can perform divisions with good accuracy.

Table 1. Numeric Type Variables: Used for storing numeric values.

Name	Data type	No. of Bytes	Signed	Range
Boolean	yes/no	1 Bit	----	0 - 1
Tinyint	Tiny Integer	1 Byte	No	0 – 255
Smallint	Small Integer	2 Bytes	Yes	-32768 to 32768
Integer	Integer	4 Bytes	Yes	-2.14×10^{9} to 2.14×10^{9}
Bigint	Big integer	8 Bytes	Yes	-2.3×10^{18} to 2.3×10^{18}
Numeric	Number	No limit	Yes	Unlimited
Decimal	Decimal	No limit	Yes	Unlimited
Real	Real	4 Bytes	Yes	$5 \times 10^{(-324)}$ to 1.79×10^{308}
Float	Float	4 Bytes	Yes	$5 \times 10^{(-324)}$ to 1.79×10^{308}
Double	Double	4 Bytes	Yes	$5 \times 10^{(-324)}$ to 1.79×10^{308}

You might have noticed that many numeric variables in Base using HSQL have the same parameters. This is because those variables might behave differently when Base acts as a front end for other database systems. When you use the HSQL embedded engine -like in this tutorial-, you can consider these variables as interchangeable.

To be more precise, Base using HSQL understands NUMERIC and DECIMAL as one set of interchangeable types of numeric data. Their characteristic is that they can hold very large numbers. On the other hand, REAL, FLOAT and DOUBLE, also interchangeable, handle better the divisions. For example, if you store 8 and 10 as NUMERIC and then ask for 10/8, Base will return 1.2. If you store them as REAL, it will return: 1.25.

Don't be afraid to use all variable types at your disposal. Just make sure you assign the data type that best describes the values you need to store. If your database records the number of children or previous spouses of a person, an unsigned byte should be enough and a signed double would be a waste. Not only does this planning save memory but it also ensures that the application will run as fast as it can and will be less prone to breakdowns.

The same care should be exercised with **Alphanumeric characters**. Although the number of bytes per character might depend on the code system used (e. g. ASCII or Unicode), Base allows you to limit the maximum number of characters it will accept for an entry. Here you have some options: Lets say that you have the attribute 'Surname'. You assign it a CHAR (fixed) data type and give it the value of 50. This means that Base will assign a space of **exactly** 50 characters to store each 'Surname' entry. What if you enter the name 'Verne', that only uses 5 characters? The computer will store 'Verne' and 45 blank characters. Instead, if you assign the variable type VAR CHAR (Var short for 'Variable') and the value 50 then the computer will store **up to** 50 characters and not more BUT would only store 5 characters for the entry 'Verne', saving you memory. If you try to store more than 50 characters, Base will only record the first 50 characters and ignore the rest.

You will have to do some research before assigning length values. 50 characters could seem like a waste for surnames, but five or ten could be dangerously low. Also check address formats and the length of street names that could show up. You will notice in the chart below that the different alphanumeric data types can store, as of the time of this writing, up to 2 gigabytes of characters, which is more than plenty. You will also note that the parameters are the same for several data types. Again, in HSQL they can be considered equivalent.

Table 2. Alphanumeric Type Variables: Used for storing alphanumeric characters.

Name	Data type	Max length	Description
Memo	Long Var Char	2 GB for 32 bit OS	Stores up to the max length or number indicated by user. It accepts any UTF 8 Character
Text (fix)	Char	2GB for 32 bit OS	Stores exactly the length specified by user. Pads with trailing spaces for shorter strings. Accepts any UTF 8 Character.
Text	Var Char	2GB for 32 bit OS	Stores up to the specified length. No padding (Same as long var char)
Text	Var Char Ignore Case	2GB for 32 bit OS	Stores up the the specified length. Comparisons are not case sensitive but stores capitals as you type them.

Another important type of variable is the **Date** type. They are used to store calendar information like year, month, day, hour, minute, second and fraction of a second. There are several types, designed to be the most efficient in storing some or all of this information. **Date** allows you to store year, month and day as it is stored in your computer (yes, your computer keeps track of this. Other database systems use the time stored in servers in the Internet -which could be more precise, but then they require an Internet connection). The same is true for the **Time** type variable, which stores the time of the day: hour, minute and second. Be sure to understand the format in which this information is given. The USA uses the month before the day. Other countries use the day before the month. Some countries use the am/pm format while others use the military format (e.g 19:30 hrs. for 7:30 in the evening). These preferences are set globally when you assign the language for the OpenOffice.org suite. Finally, some procedures might need you to record both the time and day of an event. **Timestamp** has been designed for this, recording all information at once.

Table 3. Calendar Type Variables: used for storing dates and hours.

Name	**Description**	**Format**
Date	Stores month, day and year information	1/1/99 or 1/1/9999
Time	Stores hour, minute and second info	Seconds since 1/1/1970
Timestamp	Stores date and time information	

The **Binary type variables** allow you to store any information that comes as a long string of zeros and ones. Digitized images use this format. Digitized sound uses this format too. In general, anything digitized is stored as a long sequence of zeros and ones. They are told apart by the computer because the first zeros and ones identify the kind of file they represent (a JPEG image or an MP3 file, etc.). However, Base will make no attempt to identify the kind of file you have stored. This is to say that it won't care if the file is an MP3 or a TIFF and it will happily store it, regardless. The only limitation is the amount of memory the file uses. At the time of the writing of this tutorial, the maximum amount of memory Base will use to store Binary variables is 2 gigabytes.

This in effect means that you could use a Base database to store, for example, photos of the members of a project or the staff , or sound snippets or voice messages. However, be warned that images and sounds tend to use a lot of memory and limit the functionality of your database.

Again, the different Binary types can be assumed as interchangeable when using the embedded HSQL database engine.

Table 4. Binary Type variables: Used for storing files like JPEGs, Mp3s, etc.:

Name	Data type	Max length	Description
Image	Long Var Binary	2GB for 32 bit OS	Stores any array of bytes (images, sounds, etc). No validation required.
Binary	Var Binary	2GB for 32 bit OS	Stores any array of bytes. No validation required.
Binary (fix)	Binary	2GB for 32 bit OS	Stores any array of bytes. No validation required.

Finally, there are two variable types offered by Base called OTHER and OBJECT. They are used to store small programs that really advanced developers of databases can use in their applications. Essentially they store binary data just like Binary but this time Base pays attention to the beginning of the code so it knows what kind of program it is and how to use it. We will not be using this type of variables in this tutorial.

Table 5. Other Variable types: For storing small computer programs in the Java language.

Name	Description
Other	Stores serialized Java objects – user application must supply serialization routines
Object	Same

Why is all this information relevant?

Base, working with the embedded HSQL database engine, requires that all your database (data, data structure and forms and reports) be in RAM memory while you work with it. This means that the number of records that you can keep will be affected by the size of your variables and the amount of RAM in your computer. The number of your reports and the speed in which they are calculated will also be affected by this.

In theory, HSQL limits your tables to 2 gigabytes (which is a lot of memory) and the overall size of your database to 8 gigabytes (remember, this includes the forms and reports -but still it's plenty). If you had a table with an image that is roughly 2 gigabytes, you would not be able to record more information in that table. You would have one table with only one record in it.

In practice, however, and at the time of this writing, computers tend to have 512 megabytes, 1 gigabyte or 2 gigabytes of memory, reducing the resources available to your Base database. Although this is still plenty for a functional application, you can maximize your resources by spending some time assigning variable types and memory allocations commensurate to the variables you need to record. This will also help your application run faster. Now you know!

On logical Names and Physical Names

Throughout this tutorial I try to use the words “Class”, “Table”, “Attributes” and “Columns” as if they belong to different logical levels. We defined a class as a collection of objects that share the same attributes. A class translates to a table and the attributes conform columns, which is why we are tempted to use them interchangeably; but in fact they denote different things. A class is a logical relationship, formed when the abstractive powers of our mind are able to establish patterns of commonalities among the entities we are working with. A table is the physical expression of such a class, as embodied in the way the database software and hardware actually store, index and organize data.

When thinking about the name for our variables it can be useful to differentiate between a logical name for the variable and the physical name used in the application. For example, “First Name” is the name of the variable that stores the first name of our objects, like “John” or “Chloe”. “First Name” describes a data entity we are working with. It does not matter if “First Name” is alternatively written “FIRST NAME” or “firstName” or “first_name” or even “F_N” as long as we conceptually understand that it references the first name. When we think about conceptual or logical relationships, a clear and descriptive name is all we need. This is the kind of name we would use in our UML diagrams, which is an example of conceptual or logical data model. Unfortunately this does not translate so simply into the names that a database software will allow us to use. For example, Base working with HSQL will allow us to use “First Name” but other database applications would reject it and ask us to use “first_name” instead. This has to do with the way a particular database software has been designed and the conventions accepted then. The name actually used in the internal structure of our tables is called a “physical” name, as opposed to the logical name discussed above. This physical name will have to conform to the conventions imposed by the software that we are using.

To be clear, Base using the HSQL engine can accept spaces and other special characters as valid characters for names and can also act as caps sensitive as long as you enclose the names within double quotes. Because of this we will be using physical names in this tutorial that are identical, for the most part, to their logical names, which will simplify our work when we start creating forms with Base. We will see this in action in part III of the tutorial.

If this is so, why do I care about how other database programs handle variable names?

You will discover how much thinking goes into designing the tables and their connections. Compared to such effort, actually building them can seem almost trivial. Once you have the logical structure, you can easily transport it to any database application. What if the code itself, with which you create the database in Base using the HSQL database engine, could be transported to other database software as easily? This is not always possible because of different physical name conventions.

Furthermore, Base can act as a front end for several different database engines (known as Relational Database Management Software, or RDBMS). Later on you might want to start using databases that offer you more features, speed or stability and, therefore, mi-

grate from the HSQL embedded engine to some other database software. If you are planning to use Base in this way then you might want to make sure that the software that you will migrate to accepts the variable names you have chosen and properly understands the variable types that you are using.

To facilitate porting your design to other database software, you can conform to the following practices when naming variables:

1. Start all variable names with a letter.
2. For subsequent characters use either letters, numbers or the underscore character.
3. Don't use a space between words. Instead separate them with the underscore.
4. Don't use special characters except for the underscore.
5. Use abbreviations, if needed, to help keep the length of variables names short.

What short means exactly will be determined by experience and the context of the problem. You will see how our example uses some abbreviations.

In any event, In this tutorial I will focus on the HSQL embedded database engine, which allows me to use spaces, special characters and capitalization. As you will see soon, this will facilitate the creation of the forms I need for data entry. Let's get going!

Part II

Things you need to do before you code a database with Base.

Contents:

Chapter 4

Let's get real!

In this section we will translate the ideas analyzed in part I into a concrete project that should help illustrate how to ***apply*** such concepts. This example will be of a general nature -in order to facilitate the application of the operations described here into projects of your own- and of a medium level of complexity. This should make our work both challenging and useful.

Case Example: Now we need a problem...

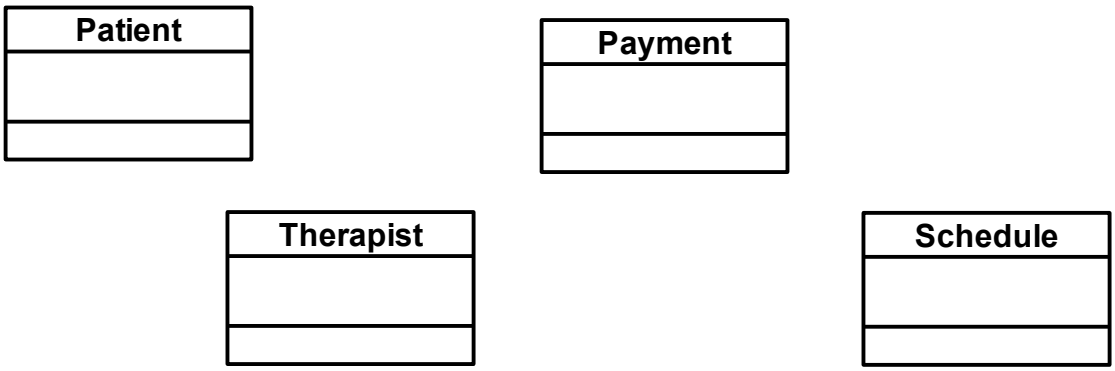
Let's apply the phases of database design to a concrete and possible problem. We will consider the needs of a small psychotherapy center. The director will be the client and you and I will be the experts.

Our first step as experts will be to try and identify the presenting problem. Because this is a stylized version of the work, the most important questions -and answers- will appear quite quickly. In real life this can easily take several meetings or a good chunk of time devoted to thinking about the operations or business processes we will be aiding.

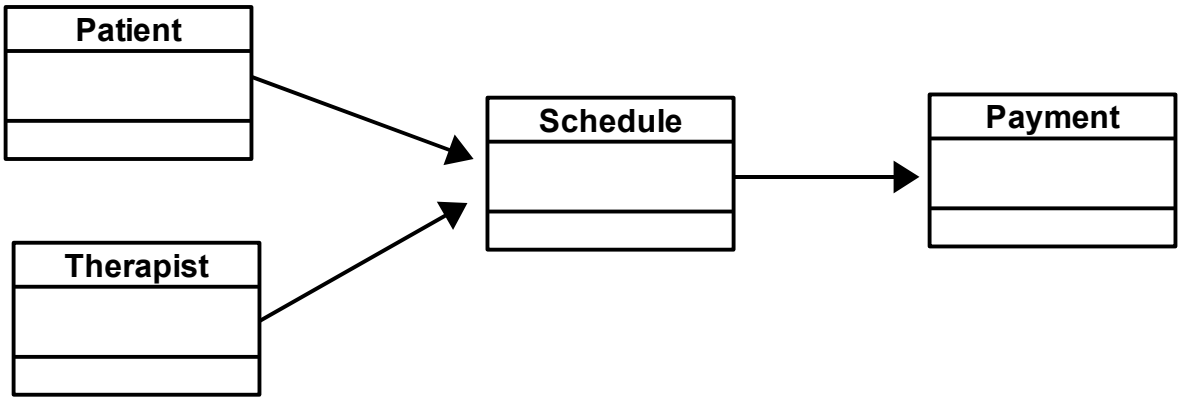
In our example we start by asking the Client and learning what is the business about and who the principal players are. The clinic receives people looking for psychotherapy services. They are usually referred by their medical doctor but not always. Once evaluated and admitted, they are assigned to a therapist according to relevant training and time availability. Then they show up for a number of sessions they, or their insurances, need to pay for. The director will then pay the therapists according to the number of patients they saw in a month.

Now that we have a broad understanding of the process, we want to know what the needs of our Client are: The Client needs to record data from his patients and decide if they should be admitted to the clinic. He needs to organize the admitted patients, matching them to a therapist and finding an available time slot. He needs to keep track of the performed sessions, by therapists, so he can pay them, and by patient so he can charge them. He also needs to keep info about the therapists, their training and other data for tax purposes.

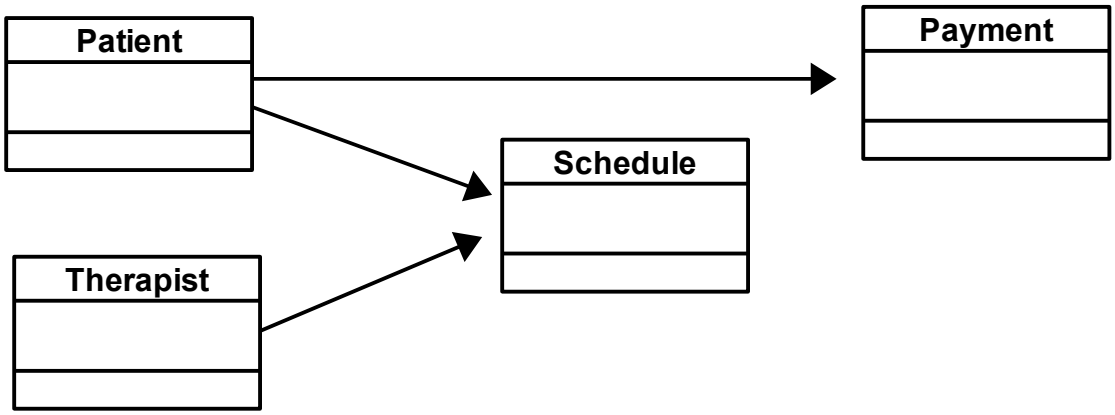
So we need to record info about the patients, the therapists, about their meetings and about the payments. After our first (and very stylized and lean) approach we can start to identify some classes we are going to need: The Patient class, the Therapists class, the Schedule class and the Payment class.



We can imagine that a patient and a therapist will be assigned to a schedule and that a payment will be associated with a performed session, as recorded in the schedule. So in very general terms we can imagine a structure of connections more or less like the following:



Now here comes the first problem with assumptions: They can be wrong! So it is always better to ask about what we are thinking instead of taking it for granted: When we show this diagram to the Client, he tells us that, although our first assumption -that patients and therapists will be assigned to a schedule- is correct, our second assumption -that payments are associated to a performed session- is not. First, patients will have to pay for a session they didn't show up for if they do not cancel it at least 24 hours prior, so payment is not necessarily associated with performed sessions. Second, patients might request a payment plan, allowing them to make -say- weekly payments of a fraction of the cost of the session. This way, they will make small payments until the debt is complete even if they are no longer receiving therapy. All right, good we asked! Now we know that the schedule will have to record if the session took place or not and if it was properly canceled in order to keep a tally of money owed; and that the payments will only be associated with the client and not to the Schedule. We can modify our model to reflect this in the following way:



It might seem like a small modification but, at this very early stage in the process, this change has taken our design in a different direction.

We go back to the Client for more information: We heard that some patients are referred by their medical doctors. Do we need to collect this information? The Client says yes; furthermore, every patient needs to have one medical doctor on record. Are there other professionals we need to register for every patient? The Client informs us that some patients might also need services by a psychiatrist and that this data, along with the medication they are assigned, needs to be collected. Do all patients need a psychiatrist? The Client informs us that no, only some patients could need it.

We know by now that the Client needs to collect info about the potential patient but that he also needs to perform an evaluation of several factors and later make a decision to admit him/her or not. Now we can see that there is contact info, evaluation info and resolution info. The dialog between Client and Expert continues: At what moment is the patient registered into the database? When he calls seeking services or when it is resolved that he be admitted? Will all the information requested be available at the time that the record is entered? If not, what information is essential to decide to register a record? (remember the NOT NULL attribute of columns?).

By asking the former, we the experts learn that the patients will be registered into the database only after they have been admitted, which means that the evaluation has already been performed and a diagnosis been made; this way there is no need to collect all the information generated by the evaluation or the resolution. Being registered in the database is a very important milestone as it marks the moment when the patient is incorporated to the clinic and becomes a responsibility of it. For this reason, it is also important to record the date at which the patient is registered into the database. However, at this point in time, there could still be some information that has not been properly collected from the patients. For example, some could be taking medication and, although know the name of it, not be sure of the dosage. Others could have forgotten the phone number of their medical doctor and promise to phone later with it. This shows us that some information is essential while other is not... The Client gives us a list of the information he wants to collect

about the patients but also informs us that the name, address and the diagnosis -proof of the evaluation- would be enough to make an entry. This essential information will later become NOT NULL records.

Let's continue the dialog between the Client and the Expert: What data do we need from the therapists? Of course, name and contact information, their degrees and specialties and tax information so they can be properly paid. And what data is relevant to collect in the payments class? The Client wants to know who made the payment and how much he paid. At this point we can use our imagination and identify data that the Client has not thought of but that sounds important to us: For example, would the Client like to record the date when the payment was done? Yes, he answers. Would he like to record the date a therapists is hired or stops working at the clinic? At first the Client seems unsure about this info, that it would not be commonly used, but later realizes that it would be proper administration and decides to include that data.

What about the schedule class? The Client states that he will want to record the time slots the patients are assigned and the therapists they are assigned to, the date when the patient is assigned and the date the case is closed. As we discovered earlier, he also wants to record whether the patients came or not to a scheduled session and, if not, if they properly canceled the session. We, the experts, ask him what happens if it is the therapist who has to cancel a session. In that case, he answers, the patient should not be billed for that session and therefore, who cancels the session (the therapist or the patient) should also be recorded.

After reviewing the paper forms the clinic has been using up to now, we discover other elements of interest: Some patients have a home phone number and live happily with that. Other patients have a home number, a cell or mobile number, a job number and maybe even a second job phone number. We ask why and discover that being able to get in touch with the patients is very important for the clinic, particularly if an emergency arises. This also leads us to the possibility that the Client could need extra contact information to reach the medical doctor and the psychiatrist in case of an emergency. We ask the Client about this but he says that the staff of those professionals will know how to reach them if needed, so we only need to record their office numbers.

What reports does the Client need to make? At first, the Client can think of three principal ones: Clinical reports, financial reports and schedule reports. The clinical reports should contain contact information about the patients, diagnosis, assigned therapist and medical doctor, and psychiatrist and medication info if any. Financial reports need to provide information in three different ways: The overall number of billable sessions performed in a set period of time, patient information that includes the total number of sessions performed and the total amount of money payed by him and, lastly, the assigned sessions to be payed to every therapist. The schedule report should indicate what therapist has been assigned to what patient and in what time slot. After thinking about this for a while, the Client also tells us that he would like to have a summary of all the sessions to be performed next day with the name and contact data of the patients so they can be

called and confirmed. Hum! Challenging. Let's see if we can accommodate these requests!

Now, just in case this has been going too fast, let's make a summary of what has happened so far. A client, the director of a mental health clinic, has approached us, the experts, to develop a database to organize his clients, therapists and to keep track of payments. We, the experts, have inquired about the way this clinic handles business, previsualized the reports required by the client and analyzed the paper forms he currently uses. We also stressed our own imaginations and offered some ideas, all of which helps in fleshing out the classes and attributes we could need. With this work we have identified four main classes: Patients, therapists, schedules and payments.

So far, so good.

Possible solution.

Now that we have relevant information, we should sit down and write a formal statement about what the goals or purpose of our database will be. By this I mean the functionality that, we as experts, believe that is possible to provide for the situation and needs revealed in the previous research.

So after thinking about it for a while we write down that we will develop a database that stores contact and clinical information about patients, that stores and maintains information about services provided, including dates of sessions and assigned therapist, that stores and maintains information about payments received by them and that stores information about therapists and their qualifications. The database will be able to provide the following reports: Clinical summary of patients, summary of services rendered to individual patients, payments made by individual patients, sessions performed by individual therapists, summary of all services rendered by the clinic for a set period of time and a log of next-day sessions with patient contact info.

You can see that this statement is quite short and representative of the needs of the client and the information that needs to be collected. It provides concise goals and boils down the needs of the client into concrete procedures and final reports. It also sets a standard of completion: When our Base application can do what is stated in the previous paragraph then this project has been completed (and a new one can be started). Professional developers usually present this to their Clients and make them sign a copy. When clients later complain that the application does not, for example, provide a summary of the most common mental health illnesses, they answer: “Well, that was not in the initial specification. If you want that we can develop it for you and it will cost you $$$ extra”. Pure genius!

The other important thing is that this statement will help us make decisions about data modeling. As you might know by now, there is more than one way to organize the same data. If one of those options makes producing one of the required reports difficult or complicates the collection of information then that is an option we will surely drop in favor of

an option that makes the tasks easier. This should become self evident the more you practice your data modeling skills.

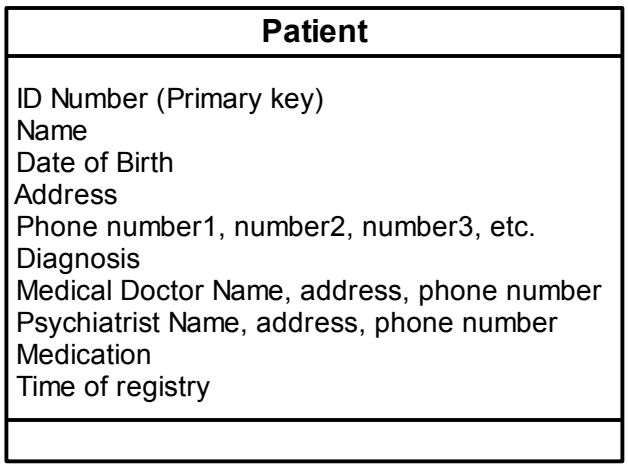
With the elements collected so far we can start drafting the forms we might need. For patients we will want to record: name, date of birth, address, one or more phone numbers, diagnosis, medical doctor, psychiatrist and medication -if present- and the date of registry. For therapists, we will want to record name, address, phone numbers, tax id number, degree, specialization and date hired. For the Payment class we will record the patient, the amount and the date of the payment. For the schedule class we will collect the patient, the therapists, the assignment date, the time slot and whether the session took place or not.

We are now ready to start modeling data. However, and to be truly honest, we have not cleared all possible questions relating our data. As we try to conform the classes, their attributes and connections, new questions will pop up that will force us to go back to the Client, rethink our designs, include new data or even exclude some of the data we now find necessary. Let's not fear this and consider it part of the job. However, our goal statement provides a direction and, whatever changes we do, they must all aid in achieving the stated goals.

Data Modeling.

Intuitively, the Patient class and the Schedule class seem the most complex. Let's star by analyzing and fleshing them out.

In our initial draft, the Patient class was connected to the Assignment and the Payment classes. This means that we will need a primary key. We will let Base produce a numeric key for each record. With the information collected so far, we can propose the following attributes for this class:



Any problems with this? Several.

Maybe you have already spotted the multivalued information for phone numbers. This is because this table is not in first normal form. Normalization here requires us to take the phone numbers and produce a new table with the multivalued item and the primary key as a foreign key. If we didn't do this, we would have an arbitrary number of columns (Phone1, Phone2, Phone3, for example) that sometimes are populated and oftentimes not. We would also be unable to record a fourth phone number if the patient has one. All this is solved by extracting the phone numbers and making them their own class.

Do we have problems with second normal form? No, because this class does not have a compound primary key, so there is no way a subset of it could also be a primary key.

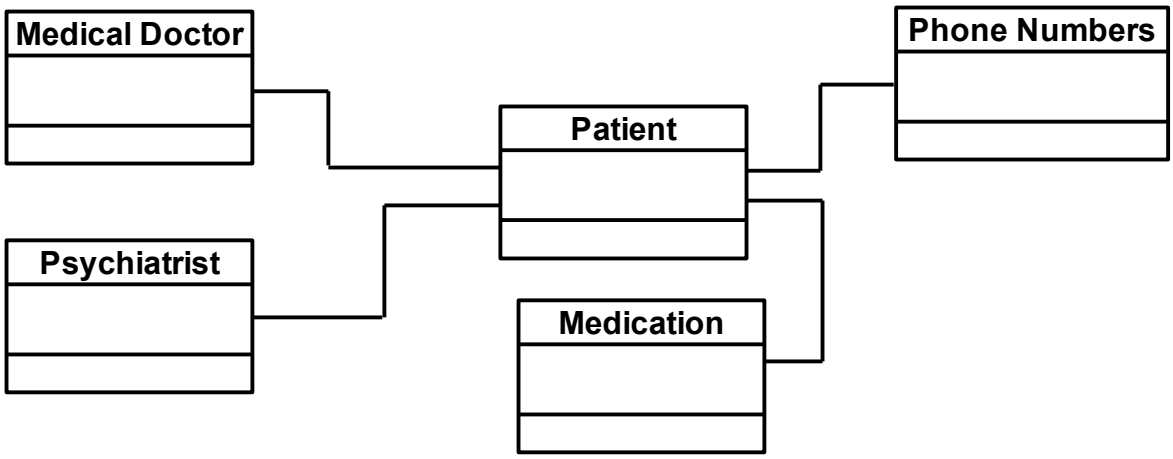
But there is something fishy about the data for the medical doctor. Notice this. I do need the primary key to know what medical doctor a particular client has. But I don't need the key to know the medical doctor's address or phone number, all I need is the medical doctor's name and I can tell you the other data. This could be better visualized if you imagine this class as a table with several records that have the same doctor. Only the information particular to each patient is unique, but the medical doctor's information would be repeated again and again and you could identify it by just knowing the medical doctor's name. Aha, we are in violation of third normal form here! We need to extract the data 'Medical Doctor' and make a new table with it.

The same thinking applies to the information about the psychiatrist. I only need the psychiatrist's name and I can tell you the psychiatrist's address or phone number. This info also requires a table of its own. Not all patients have psychiatrists, which means using optionality zero, and a patient will have at most one psychiatrist. We can already anticipate that we are going to need a cardinality of: 0..1.

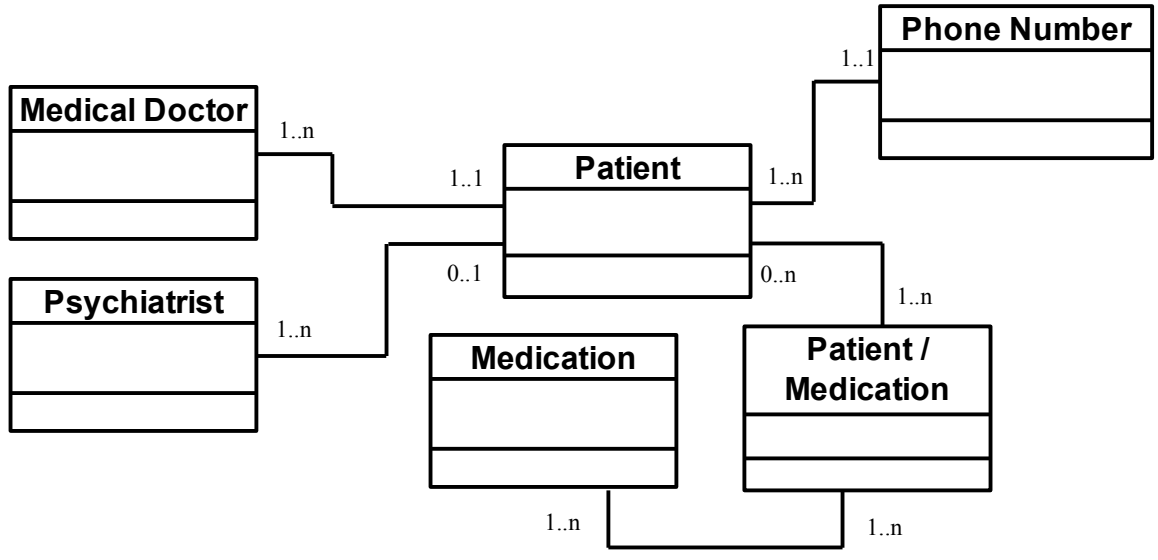
The other way to look at this situation is to recall the first principle in class formation that states that a class must be about one topic and one topic only. Our first draft for the Patient class does not follow this rule and includes other topics like medical doctors and psychiatrists.

Something similar happens with medication. Not all patients need medication. This is, again, a set-subset situation and medication needs to be made its own class. Would normalization spot that medication needs to be in its own class? Yes, but by a different route: people could need one but also two, three or more different medications, reviving the problem of multiple values for one item of information alluded by first normal form. Therefore, medication would have been made a table of its own. Both ways of thinking reach the same conclusion and are, in consequence, equivalent.

After applying normalization to the Patient class we have developed four new classes that relate to it. Using UML we can describe their relationships as follows:



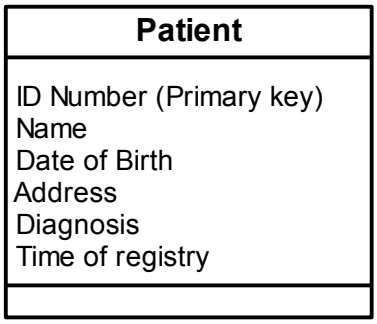
Let's now think about the cardinalities: By asking our Client, and using our imagination, we can realize that one medical doctor could be responsible for several Patients (1..n) but that each patient will have exactly one head medical doctor (1..1). Similarly, one psychiatrist could be seeing several patients from this clinic (1..n) but Patients will have either exactly one psychiatrist or none (0..1). Phone numbers are not a big mystery: one patient can have one or more phone numbers (1..n) while phone numbers can be *thought as* belonging to exactly one patient (1..1, although family members could have the same number). What is the situation with the medication? We already know that one patient would be using either none, exactly one or even several medications (0..n) while several medications could be being used by several patients at the same time (n..m). Aha! n..m relationships force us to use an intermediate table between Client and Medication! We will call this table Client/Medication. Let's add all these elements to the diagram:



What seemed like a simple Patient class was decomposed to form five new classes. However, making this decomposition was not difficult at all as it only took applying the rules of Normalization and understanding well the needs of our client and the conditions under

which business operations take place. Intuition also suggests that this new design is far more flexible and reliable than the first one-table draft for a Patient class

Consequently, the new Patient Class will be structured in the following way:



Any problem with this? Again, yes.

To start with, the table records the name of the patient as only one attribute. If we later wanted to send mail to the patients we could only use the obviously computer-generated “Dear John Smith:” -for example- unless we embark in complicated string manipulation. If we record the name in smaller categories, like first name and surname, we could have the option to produce different tones in our letters, like a formal: “Dear Mr. Smith:” or a more friendly “Dear John:”.

Generalizing this, it will give us more flexibility if we tend to record the smallest unity of usable information. What is the definition of usable? That will depend on the goals of your database. For example, there is no use for me to distinguish between the area code and the phone number and I would record them both as one attribute. Yet, for someone who is analyzing patterns in phone numbers maybe storing each individual number of the sequence in its own column would make plenty of sense.

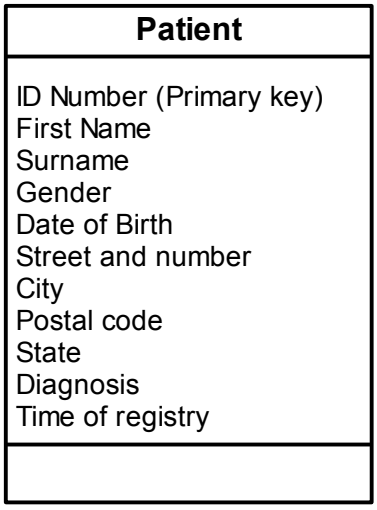
In the same line, it is useful to me if I atomize the address information. Where I live, addresses include a street name and a number, a city, a state and a postal zone number. By storing each in its own attribute we could later, during maintenance, develop reports that produce statistics that show what cities or postal zone codes patients tend to come from, for example. Sure, this is not part of our initial goal statement but that was because we were very inexperienced then and didn't know about the power and flexibility of atomizing information categories. If the client later asks us for such statistics, we will be ready to deliver and not discover with dread that our data structure does not support this.

Another helpful thing is to include gender information. I really dislike mail that reads: “Dear Sir/Madam John Smith”. This happens because the database was not able to capture the gender of the person. I might not be the only one complaining because I have seen attempts to solve this. Some forms will ask for a prefix: Mr., Mrs., Miss, etc. You might know that OO.o Writer allows you to create letter templates with conditional text

that will be printed only if certain conditions are met. For this reason I would recommend to include a variable to record the gender of the patient, for example:a Boolean variable where 0=male and 1=female (this assignment is arbitrary, of course) or a CHAR to hold 'male' or 'female' and then use this as a condition to tailor the text. Additionally, we can later compile statistics that include gender as a factor.

Summarizing, what we have done is atomize information (e. g. name ➔ first name + surname) and added descriptors (e.g. Gender info). These same considerations will be applied to the other classes: Medical Doctor, Psychiatrists, Therapists, etc. Even a class like Medication can benefit from a descriptor, recording, for example, what the medications do or what illnesses they treat.

With these elements, we have modified the structure of the Patient class to the following form:



Anything missing? Well, it is time to assign foreign keys.

The phone number table is quite straightforward, as we only have to follow the instructions for first normal form. This means that it will be this table that will receive the patient's primary key as a foreign key. The intermediary table “Patient/Medication”, by definition, will receive the primary keys from the Patient table and also from the Medication table.

What about the medical doctor's table? If we made this table receive the patient's primary key, this would mean that we would be repeating the medical doctor's info for every patient that had him as a head doctor (visualize this by imagining a populated table that repeats all data for a doctor except for the foreign key that connects it to a particular patient), and this would not make any sense. In fact, it would be creating the kind of redundancy that we want to avoid because it consumes unnecessary memory and opens the door for inconsistencies. Instead, we will want the patient's table to receive, as a foreign key, the primary key of the medical doctor's table. This same logic can be applied to the psychiatrist's table.

This way we know that the medical doctor's table and the psychiatrist's table will both require primary keys and that the patient table will need to store them as foreign keys. After this analysis, we can finally complete the Patient class, which takes the following form:

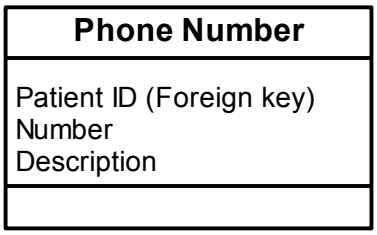
Patient
ID Number (Primary key) First Name Surname Gender Date of Birth Street and number City Postal code State Diagnosis Medical Doctor (foreign key) Psychiatrist (foreign key) Time of registry

Applying the same principles alluded so far, the related tables will have to following structure:

Medical Doctor
ID Number (Primary key) First Name Surname Gender Street and number City Postal code State Phone number

Psychiatrist
ID Number (Primary key) First Name Surname Gender Street and number City Postal code State Phone number

OK, we can see the primary key that Base will generate for us, we can see that atomization of name and address information. We can also see the gender descriptors. But wait a minute!! How come these tables have a phone number attribute? How come Phone Number is a table in relation to the Patient class but is an attribute in these two tables? This is dictated by the needs of the Client. He needs to record as many numbers from the patients as possible, in order to handle emergencies; but only needs one number from the professionals as their staff will know how to reach them if they are not immediately available. The defining factor is that in one case we need an undetermined number of phone numbers and in the second case we know that we need exactly one.

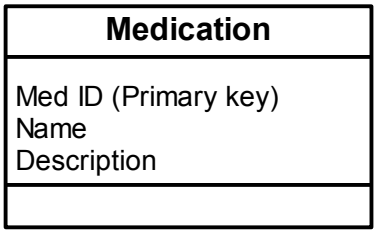


OK, we can see the foreign key connecting the phone number with the proper patient. We can also see a descriptor with the very creative name of “description”. We will use this to record if the number is the home number or the mobile or the office or the second job, etc. If it were an important piece of information, you could also add the descriptor that records the best time to call, but this didn't come up in our research. It would be up to you to recommend this or not.

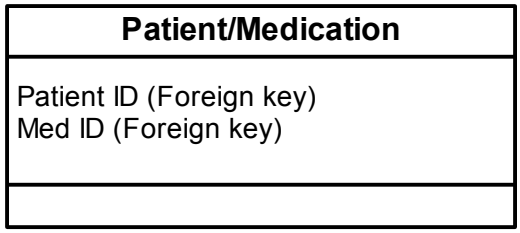
Now wait! Shouldn't there be a primary key attribute here? Strictly speaking, we don't need it, as all the information is relevant only in relationship to the Patient table. Our queries relating phone number will all have the form: “locate phone number where patient id is such”. However, please note that when you are coding your tables in Base using the GUI (that is, the graphical interface as opposed to using SQL command lines) Base will automatically generate a numeric primary key in every table, just to keep things neat. Additionally, in our example all our tables will have a primary key, even if theoretically this is not required, to aid some automatic processes handled by Base.

Another thing: isn't this a rather small table? Shouldn't tables be bigger? Isn't it a waste to create a table to just record two or three attributes? Well, we have seen that the power of relational databases comes from their ability to connect data, not from the size of its tables. It is not uncommon that databases are made of many, many rather small tables, but all really very interconnected. On the other hand, I am not sure if this table is small. It records only three attributes but, if every patient has exactly two numbers (home and office, for example; and about everybody also uses a mobile phone these days), this table will always hold -at least- twice as many records as the patient's table. Who's small, again?

With this in mind, the medication table should not surprise you:



Nor the intermediate Patient/Medication table:

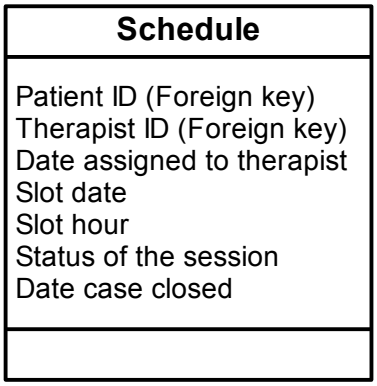


This last table records only two attributes and doesn't even need a primary key, yet is allows us to find all the medications one particular patient is using or all the patients using a particular medication. Isn't this nice?[5]

Could this table use a descriptor? Maybe we could have recorded the date this medication was assigned and the current dosage. And if we capture the date the patient started using this medication, we could also record the date he stopped using it, and keep this as an historical record for the patient (and we will be doing this in our final version). However, the need for this information did not arise in the goal statement and seems more relevant in a database for the psychiatrists, who is the one who is really monitoring the medication. We could ask the Client if he really needs this information or not. However, it works here to show that a table with only two attributes is not only possible but also very important and useful.

At this point in the game, you should be able to organize all these tables in a nice UML diagram describing the structure of the classes, connecting primary to foreign keys and indicating cardinalities. You should also be able to model the Therapist's table with what you have learned so far!

Let's focus now on the Schedule class. According to our research and the goal statement we did, this class should have the following attributes:



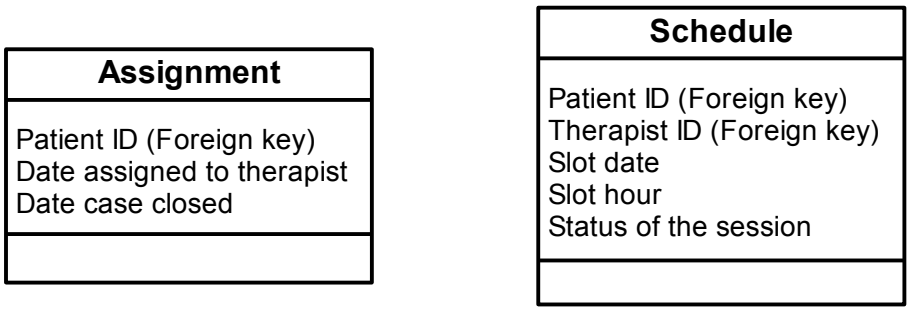
How are we feeling about this? We can see that in order to identify a slot hour, for example, we need to know both the patient id and the therapists' id. So this table uses a compound key to identify each cell. What is the rule of normalization for compound keys? Oh, right! Second normal form. That rule states that no subset of the primary key can also be a primary key. Is there anything like this here?

For the most part, I really need both elements for the key. After all, a patient needs to be assigned to a therapist from which he can have his case closed later on. Yet there is this strange feeling that the date of assignment and the space for storing the date the case is closed will be repeated unnecessarily every time a patient schedules a session. On closer inspection note that the attributes Date Assigned and Date Closed depend on Patient ID

5 Let me stress that when we actually build them, all our tables will have a primary key in order to aid Base with some automatic functions.

only and not on the compound patient-therapist ids. This table is not in second normal form!

Following the rule then, we will decompose this table into one table that only has the Patient ID, the date he is assigned and the date the case is closed (which seems proper to call the 'Assignment' table) and have another table that records the patient id, the therapists id and the slot date and time, like this:



This is much better. We can see that in the schedule table, slot date, hour and status of the session (whether the patient came or not or if it was canceled by the therapists or the patient) depend exclusively on the compound key made by patient id and therapists id.

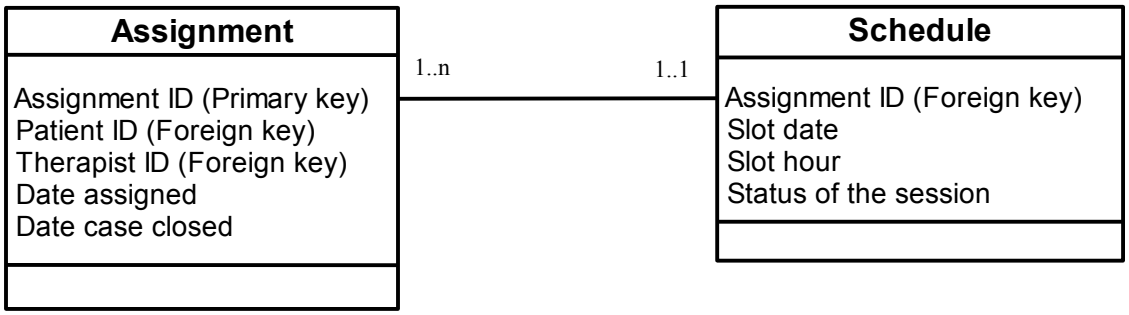
The assignment table, in turn, records the date a patient is assigned to a therapist and the date his case is closed, data that is functionally dependent only on the patient id key.

However, there seems to be a piece of missing data. I don't know you but I feel uncomfortable by the fact that the assignment table does not record what therapist was assigned. Of course, I could simply add the therapist's id to the table. This would be correct and not affect the structure of the tables.

But if I do so, the assignment and schedule tables will both have the same two attributes of patient id and therapist id. This opens the door for possible inconsistencies that, of course, we want to avoid. How do we solve this?

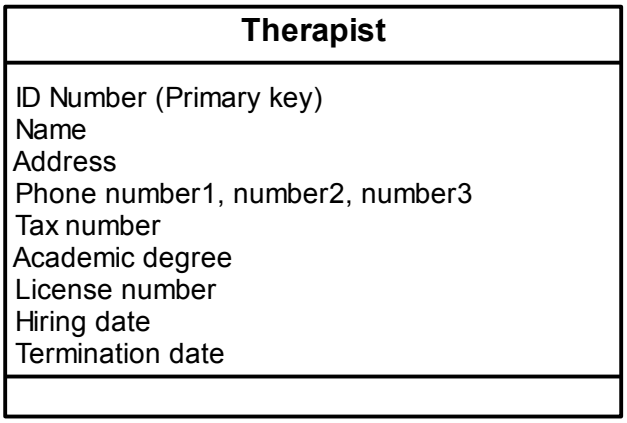
Remember that I said that there is no one way to design a database? This problem will serve as an example of that. There is no one rule that helps us here and, instead, we are going to have to use our imagination to solve this one. Different imaginations will come up with different solutions, and in all honesty, some time devoted to thinking about this could provide several options.

The solution that I use includes thinking of each assignment as its own entity. So I will record in the assignment table the patient id and the therapist id along with the dates of assignment and closing. I also provide each assignment with its own number primary key. Then in the schedule table, I use the assignment primary key as a foreign key instead of the patient's and therapist's id, and leave the slots time and date and the status of the session, like this:

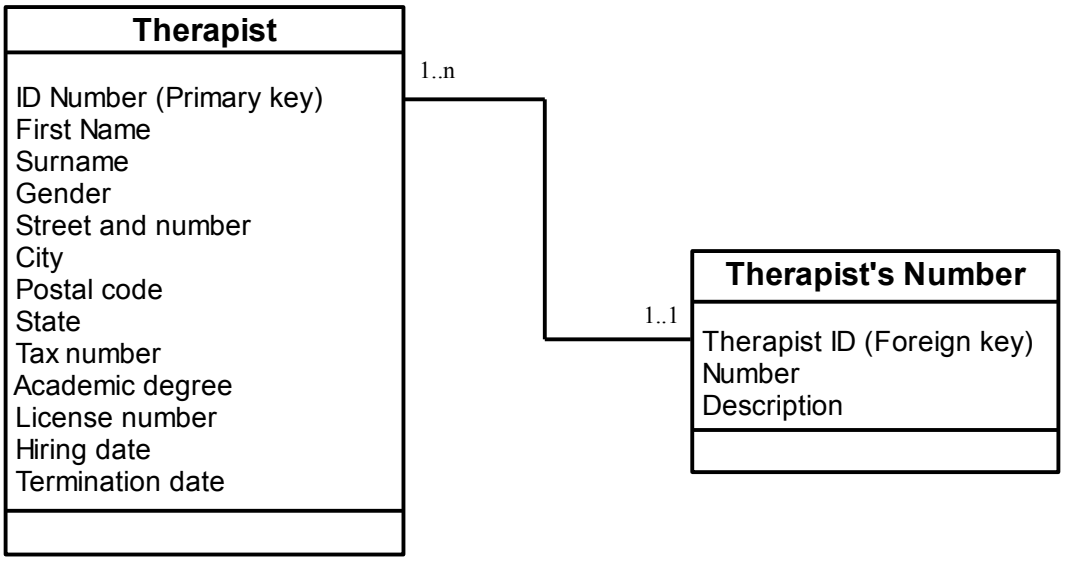


So far we have given examples of data modeling by using normalization, by using our knowledge of the needs of the client and the way he handles business (coded in our goal statement) and now, by stretching our imaginations and creativity. Expect to do all these in your own projects!

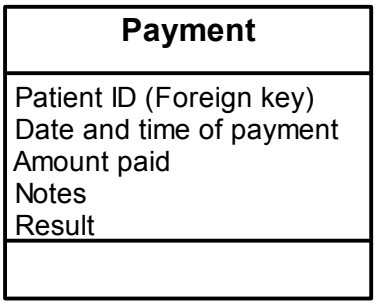
Let's finish this exercise in data modeling by conforming the Therapist table and the Payment table. For the therapist's table we will start with the necessary data identified by our research and later apply the principles of normalization, atomization of categories and applying gender descriptors. I will only render an initial and a final version. By now you should be able to describe the reasoning in the changes:



Transforms into:



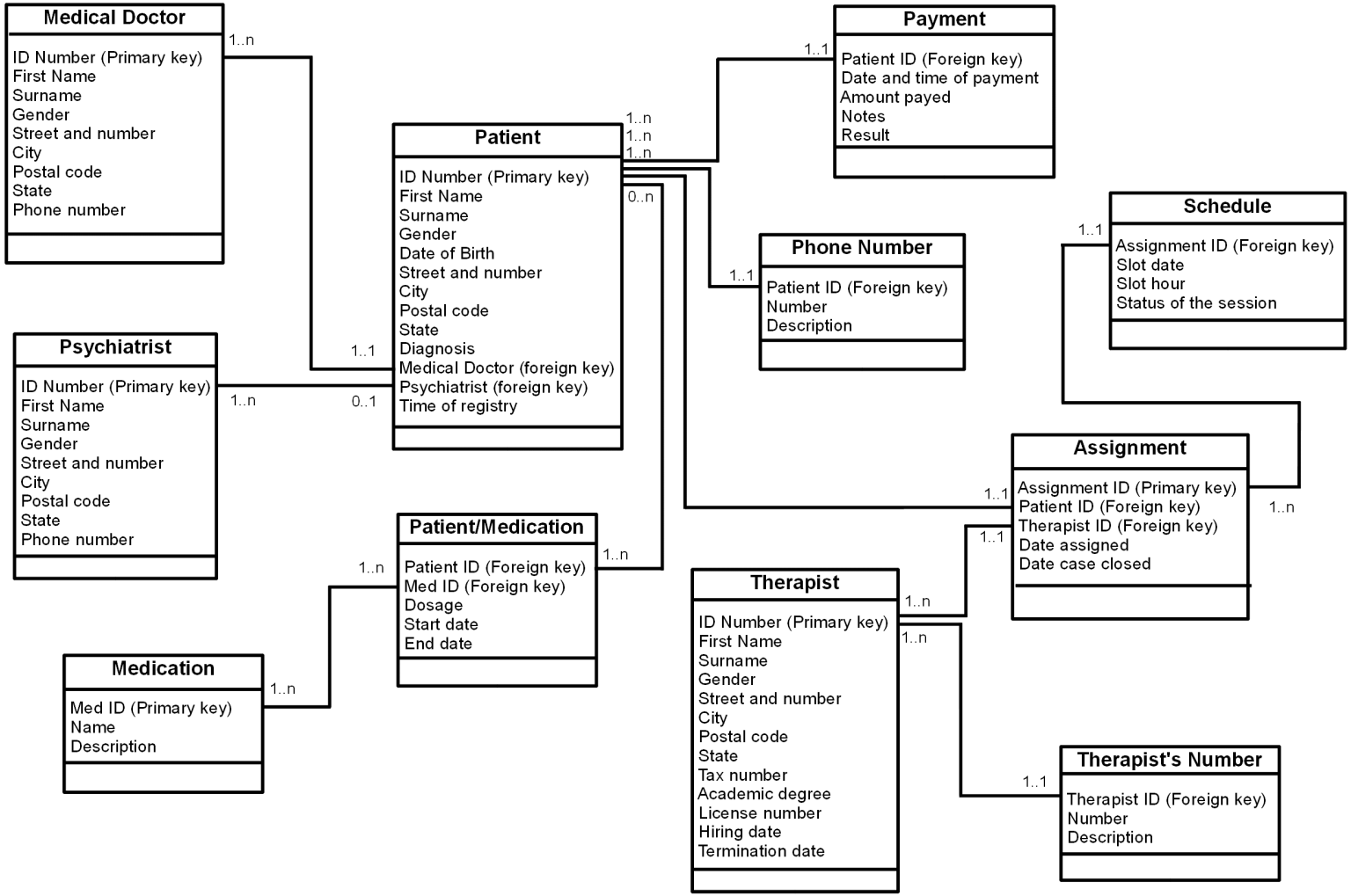
Finally, we have the data required by the payments' table:



Date and time appear as one attribute because I intend to use the timestamp data type, that records both the date and the hour the record is created. I also created a 'Notes' attribute to record special data about the payment. For instance I can record if the payment was done with cash or check. I also use 'Result' to record if the check was paid or not and enter a negative number in the amount attribute to reflect that there is still money owed. This way, the design can handle (unpleasant) exceptions to payments. I should also make a note to myself and make sure that I assign a signed numeric variable to 'Amount payed' (see Variables Definition Lists later in this chapter). Finally, and true to data integrity, I should also uniform or restrict the kind of results I can write, so that similar circumstances receive the same entry (for example, write 'Credit' for every successful payment or 'Debit' for any unsuccessful attempt at payment). As we will see later, it will be easier for the application to simply count the number of 'Credit' or 'Debit' outcomes recorded in 'Result' than to count outcomes that have several different ways to state if the payment was successful or not.

At this point we have finished the modeling of our data. We have expanded our initial draft that included only four boxes and made the necessary class extractions, we have determined the structure of each class and their structure of connections. By now you should have a UML diagram like the one in the next page.

UML Design for Mental Health Clinic Database



A little bit more: Duplicity of roles and super classes.

The example that we have developed in this tutorial belongs to the very sophisticated field of medical records and medical billing, one obvious area for database applications. This example pales when compared to the functionality provided by professional applications in this field. For example, we have not recorded what insurance company is in charge of the patient's payment, most of which have different procedures and time lines. A pro design would be able to handle all that. Also, what happens if the patient is a minor or for any other reason has a legal guardian assigned to him? All these kinds of exceptions -and more- can be handled by professional designs. Why hasn't this example gone further?

First of all, medical billing is a complex subject in itself, that is taught at the college level and is based on many years of trial and error by many bright minds tampering with it. In contrast, this tutorial aims to give you tools to develop more general types of databases, not only medical billing. However, most of those extra functionalities can be implemented using the same principles described so far: By understanding the data that your clients need to record and retrieve, by harnessing the power of interconnected tables, by not defaulting normal form, by thinking through your designs carefully, by challenging them and by testing and testing and testing them.

However, there is a characteristic of the design we have arrived to in this tutorial that could become a problem in a different context. Maybe you have even spotted this potential problem already.

If you remember, one of the goals of proper database design is to minimize redundancy. Our design does this quite well as long as a therapist of the clinic is also not a patient of it.

If you review the data structure for the therapist's and the patient's tables, you will notice that both start by asking name, gender, address and phone numbers. If a therapist were also a patient, we would have to repeat this data in the patient table, opening the chance for inconsistent data and, in any event, duplicating records.

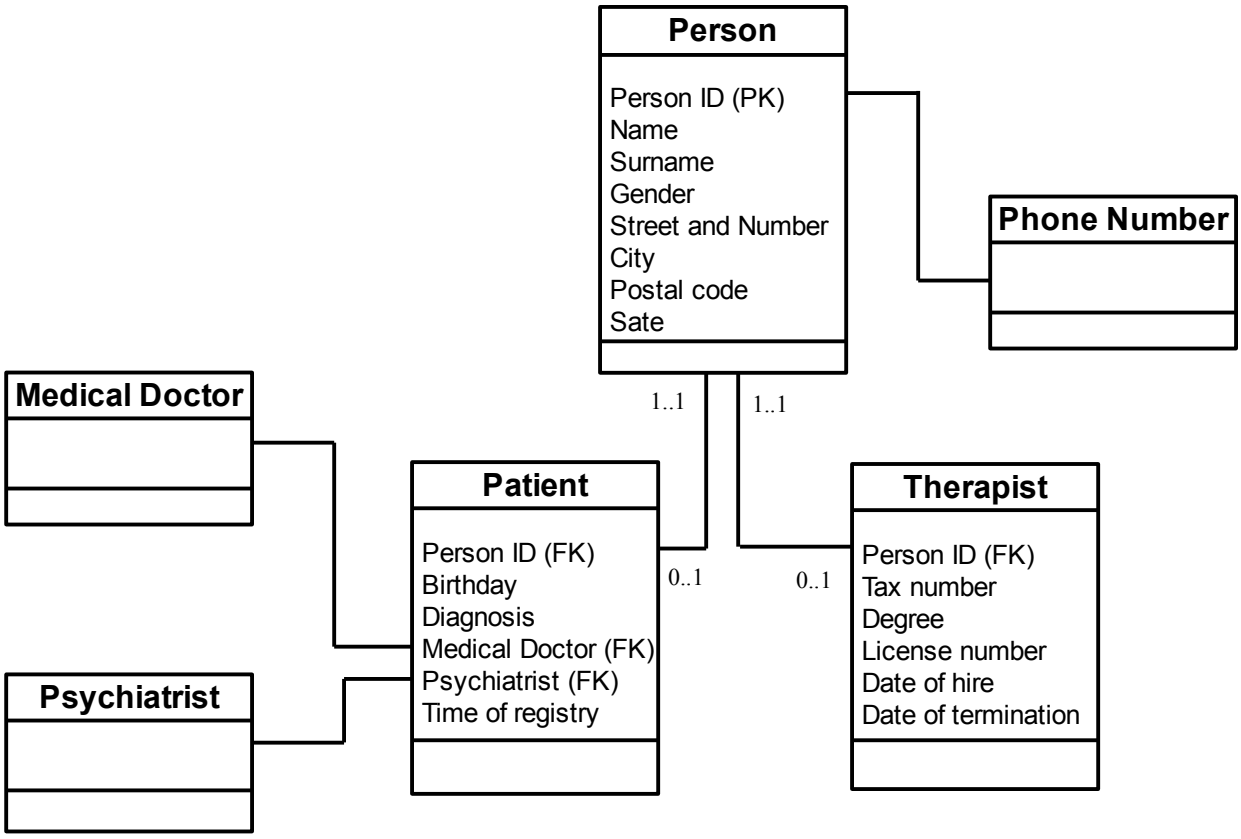
Now maybe this design makes sense in the context of a small mental health clinic where, for ethical reasons, it is not a good idea that a therapist also becomes a patient of it. But there are many other examples where we could expect this duplicity of roles and should be ready for it. For example, in a film school we have students and professors. But it could also happen that a director of photography professor enrolls in a producing course. It would also be possible that some third year students make some extra money by assisting the audio recording professor in the sound lab. In these cases, we have professors that are also students and students that are also teachers. Teachers that also study need to be enrolled, pay for the courses and receive the same benefits other students do, data that is probably not stored in the professor's tables. The same way, students that teach need to be paid and taxes deducted, and require other data that will not appear in the regular student's tables.

If we record such kind of person twice, once as a teacher and once as a student, we are duplicating the common records and opening the chance for inconsistent data.

In cases where a duplicity of roles arises, you might want to think in terms of a super class, a class that stores all the attributes common to the different roles and later relates to them as sub-classes. For example we can create the class 'Person', which will have the attributes common to students and professors (name, address, etc). We then connect this table to the student and professor tables which will record the attributes specific to each role. This way, one person can be either a student, a teacher or both.

Let's see this in the context of our own example. We have the patient and therapists tables. When we analyze their data structure, we discover that both tables require: id number, name, address, gender and phone numbers. We can extract this data to form the 'Person' table. The patient table will be left with the following attributes: a person id foreign key, birthday, diagnosis, a medical doctor foreign key, a psychiatrist foreign key and a time of enrollment. In turn, the therapist table would be left with: person id foreign key, tax number, degree, license number, date of hire and date of termination.

This structural change is summarily reflected in the following UML diagram:



Unless you have a good reason to keep the subclasses separate, like the ethical consideration in the clinic we are deriving our example from, the notion of a super class connected

to related subclasses is the solution you most often will want to apply when you find duplicity of roles.

Attributes/Variable definition lists.

Our tables will store the data with variables of which we already know that there are several types. These variables will need names, lengths or memory allocations, maybe an initial value.... To make a long story short, each variable has a set of characteristics we need to specify. In order to make sure that we assign the most economic options and that we are consistent in assigning these characteristics, we will write them down in a Variables Definition List. This takes the form of a table where we include the variable in question, the name we will give it, the data type it belongs to, maybe an initial value if it needs one and any other remarks you might want to remember about them. This is a table that we -and not the computer- will be using, but that will make coding our application with Base much easier.

We will see later that it's possible to give our variables initial values that will be automatically populated by Base when a new record is created. This is useful when you want to store a value by default which will remain true until it is modified. It is also possible to create conditions that Base will look for before accepting data. For example, you can ask that Base check that the variable “StartDate” is always older than “EndDate”. Or that the variable “Gender” only accept the values 'Male' and 'Female'. Lastly, it could be possible that we use arbitrary code conventions in the code (like 0=male, 1=female for Boolean variables). These kinds of elements should be made explicit in this list. Let's write down the variables definition list for the Patient class:

Patient Class Variable Definition Lists				
No.	**Description and Name**	**Variable Type**	**Initial value**	**Remarks**
1	ID Number	Integer	1	Unique, auto increment
2	First Name	Var Char	Empty	Length = 25 chars Not null
3	Last Name	Var Char	Empty	Length = 25 chars Not null
4	Gender	Char	Empty	Accept only Female/ Male
5	Date of Birth	Date	Empty	Format: mm/dd/ yyyy
6	Street and No.	Var Char	Empty	Length = 50 chars
7	City	Var Char	Empty	Length = 25 chars
8	Postal Code	Var Char	00000	Length = 5
8	State	Var Char	NY	Length = 2
10	Diagnosis	Var Char	Empty	Length = 60 Not Null
11	Medical Doctor ID	Integer	Empty	Foreign Key
12	Psychiatrist ID	Integer	Empty	Foreign Key
13	Time of registry	Timestamp	Current date	mm/dd/yyyy : hh/mm/ss

By now you should be able to understand all the information in this table. We have used an Integer data type for the patient id number, which is a signed 4 byte number with a range wide enough to store many patients. We are also writing the remark that this attribute should be unique to each record and should increment automatically for every new record, something that Base will do by itself. However, you need to code this -Base will not guess what your intentions are- so it is good practice to write this down so we don't forget about it. For Name and Surname we use Var Char. Var Char means that this attribute will store up to the 25 characters we have specified, or less depending on how many are really typed. Also notice the attribute of Not Null, making this a required field in order to make an entry, as requested by our Client. The gender has been assigned a Char data type that will only accept 'Male' or 'Female' as valid entries. Date of birth has been assigned a Date data type and we make note of the format used to make sure we don't get confused later. The postal code has been assigned a Var Char data type. Could have I assigned this variable a Small Integer data type -that uses only two bytes per record- instead of Var Char that will use at least 5? Yes, you could have. Just bear in mind that if you do so, the general address information and postal code will belong to different data types so you don't mix them in ways that the application might not understand, Also, remember your choice so you don't attempt string manipulation on a variable that is of the numeric type. I have also assigned an initial value for the State attribute that the Client would most commonly find in the geographical area where the clinic conducts business, so they don't have to type it every time they create a new record. Finally, I made sure to make explicit that the values for medical doctor and psychiatrist are foreign keys. This reminds me that they have to be made of the same data type that they will be assigned in their own tables. In general, all auto-generated primary keys will be an Integer type.

Notice that the second column in the table is called “Description and Name”. That is because -in this case- the conceptual and physical names coincide. Remember that in other cases this would not be possible or desirable. For example if you are planning to port the database structure to a software that does not allow spaces in names then for variable number 11 the *Variable Description* could be something like: “Medical doctor ID number (foreign key)” while the *Variable Name* would have to be something like “md_id”.

You will see that the SQL commands that we will use with Base to actually create the tables translates almost directly from this Class Variable Definition List. As an exercise, you could now write the variable definition lists for the other tables!

Now that we have the design of our database, let's check the forms and reports that we will want to use with it.

Chapter 5

The forms

Let's now analyze the forms for data entry the way we envision them for the database design in this example. It is possible that the final forms will not look exactly like these ones (although with some work it can be done) but they need to have the same functionality. After you have completed this tutorial you will be much better in guessing the finally layout of the forms for your own applications. Let's start with the form for patient information (figure 1).

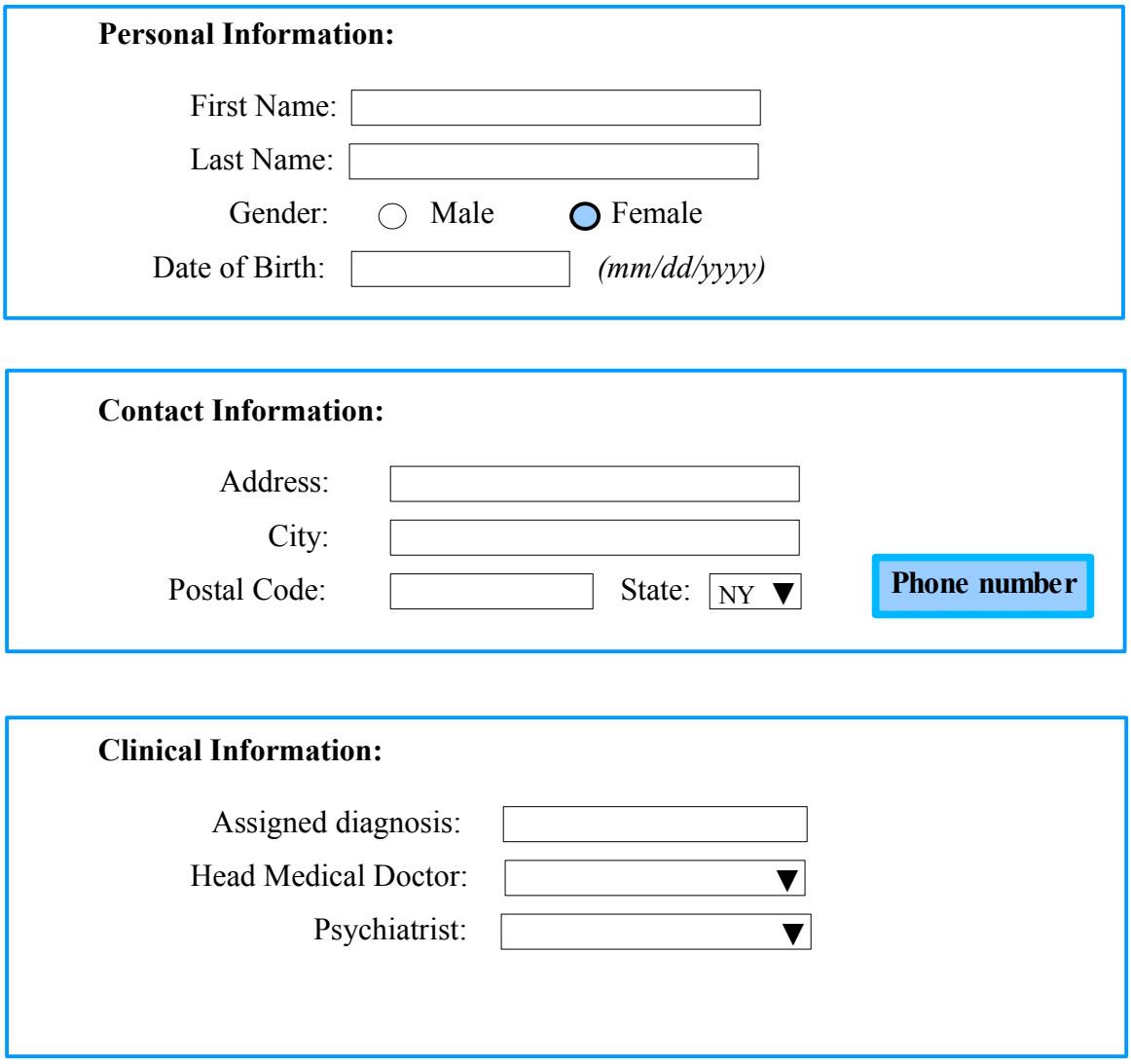


Figure 1: Patient information entry data form.

I have divided this form into 3 components to aid our analysis. The first box, Personal Information, will receive the first name, surname and date of birth with text boxes. Note that for entering the gender I do not need to write anything at all: not “male” or “female” nor “M” or “F”, nothing. All I have to do is click on the appropriate radio button. Of course this means that we are going to do some extra work when we actually develop this form. But that extra work will simplify our data entry and will make sure we only enter values within a set expected by our design.

The second box, Contact Information, has an interesting feature: If you remember, our design calls for an undefined number of phone numbers for patients, for which we have a dedicated table. The form we use will have to allow us access to both, the table where we store patient information and the table where we store phone numbers. We also are going to need to program this form so that it automatically associates the phone number to the primary ID number for the patient being recorded. This automatic registry will not be noticed by the user (saving him time), but is fundamental for the design to work. For now, I will suggest this connection between the two tables with a button that calls the phone number entry form for each number we want to enter. But this solution is not completely satisfactory and we will learn of better ways Base has to handle this.

The third box, Clinical Information, also has some interesting features: Notice that the assignment of a MD or a psychiatrist is done from a drop-down list. Like Radio buttons, Drop-down lists allow us to make sure that the data entered belongs to a predefined set of options; but unlike 'Male' or 'Female' we will not know before hand what these options will be. The Drop-down lists will be populated from the entries made to the MD and Psychiatrist tables. Because the form will have to read from the database to identify all available MDs or psychiatrists, this is some kind of dynamic form that is created based on previous entries. The extra work to accomplish this ensures that this information is entered without errors that could make it difficult later to identify the data properly (different or wrong spelling, for instance). What happens if the MD of the patient has not been entered yet? It will not be available as an option from the drop down menu so make sure that you enter the professional's data first. Also notice that the form will provide the name and surname of the MD but, once we identify and chose one, what Base needs to record is actually the primary key of the MD as a foreign key in the patient's table. We will see how all this is done!

Of course we, the developers of these forms, will have to program these functions, which will take an extra effort, but they will pay with better performance.

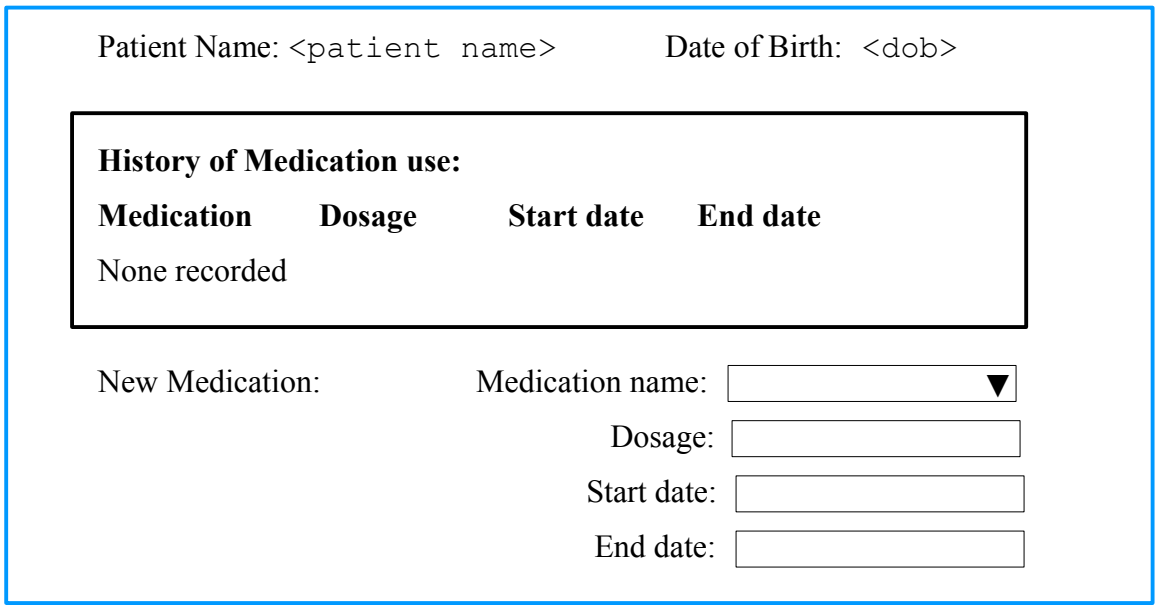


Figure 2: Medication Assignment form

The form for medication assignment also needs to be populated with data from previous entries -like medications entered in their own table- and be associated with a particular patient (figure 2). Note that the name of the medication will be picked from a drop-down list that will be populated from records in the medication table. This form assumes that the database design will record dosage and the date the medication stared and ended.

The Assignment Form (figure 3) is one of my favorites because it seems so simple, but many things are happening under the hood aside from being crucial for the working of the database. First, notice that the form just looks to pair a patient with a therapist. However, the form will provide options with drop-down lists populated from records in the database. This will ensure that only recorded patients be assigned, and because this is a point and click operation, we eliminate errors in spelling. If you remember, it is also very important that we record the date of this assignment. This will be done automatically by the form (with lots of help form our programming), so the user does not need to concern himself with remembering what date is today. If the final user does not have access to the code or to modifying this record, then this date registry could also have the authority to establish a time line. Quite cool!

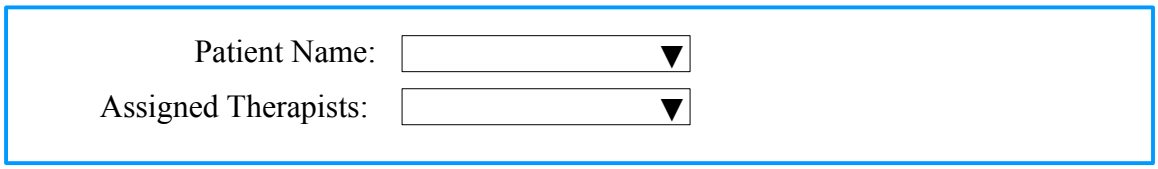


Figure 3: Assignment form

For closing a case we could use the form in figure 4. Note that all data about the file is populated by the form. For entering the closing date we will use a special gadget that allows us to enter dates with the ease of point and click and thus minimize errors in format and even in content.

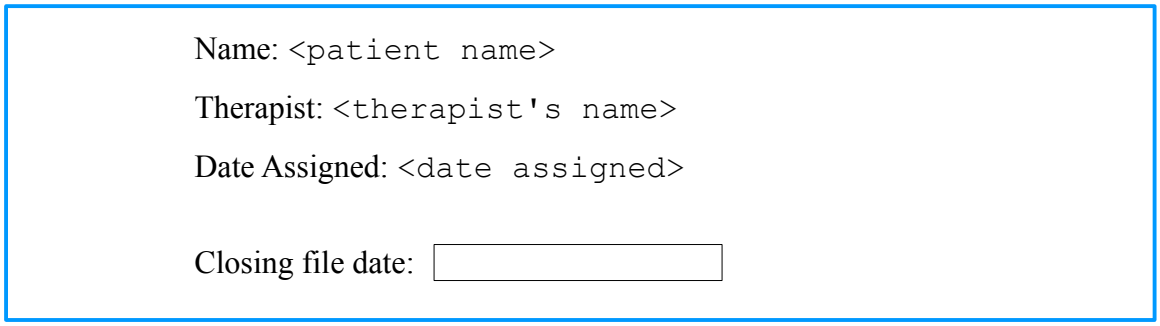


Figure 4: File Closing form

The Scheduling Form (figure 5) requires elements that we have already discussed. The purpose of this form is to set the date for a future appointment. The form will provide the name of the patient and the name of the therapist, which have been assigned previously. The user will input date and time of next session. We will be using very cool tools provided by Base to enter date and time information.

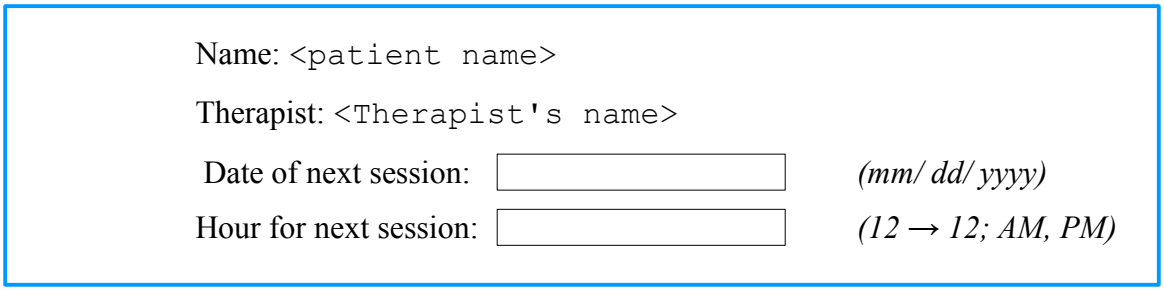


Figure 5: Scheduling form

The last relevant form is the Follow-Up Registry (figure 6, next page). Here is where the user records whether the patient came or not or if the session was canceled and by whom. The form will indicate the name of the patient, the therapist assigned and the date and time when the session was supposed to take place. Below, the form offers just 4 radio buttons of which the user must chose only one. That will be a simple click but will allow complex calculations that have to do with generating bills and payments.

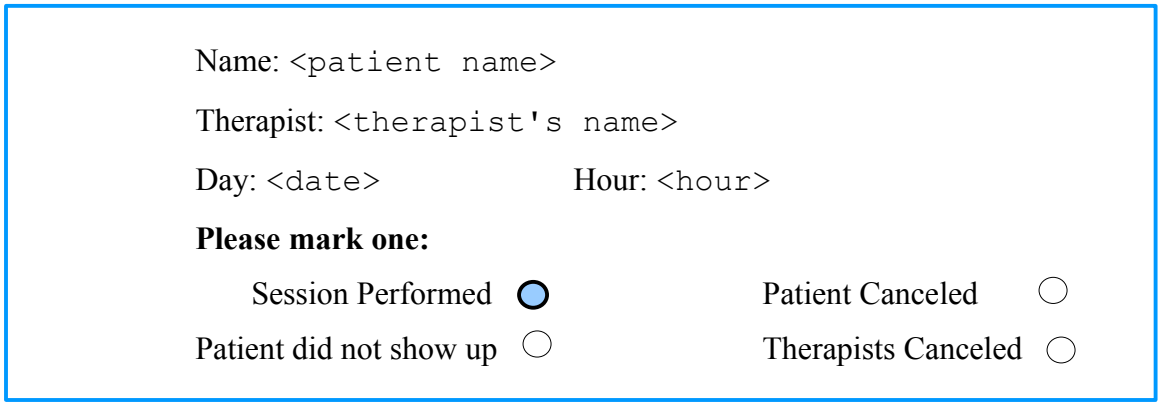


Figure 6: Follow-Up Registry

To sum this up, the forms that we need to create will receive and understand input from radio buttons, drop-down lists and other automated gadgets; and most of these drop-down lists will be dynamically generated based on data previously recorded in the database. This way we can not only minimize data entry errors but will also make these forms easier to use for the user.

Please note that other required forms (like the forms for the MDs or the psychiatrists, for example) are not described here because they require features that have already been identified.

Let's now take a look at the reports we would want our database to be able to produce.

Chapter 6

The reports

In the specification made earlier we identified at least five reports that the database needs to generate:

1. Clinical summary, which displays the patient information and the clinical information for each patient;
2. History of services and payment summary, which displays the history of sessions (whether they happened or were canceled) and the history of payments performed in a specified period of time;
3. Summary of all sessions performed by a particular therapist in a specified period of time;
4. Summary of all sessions performed by the Clinic and all payments received in a specified period of time; and
5. Next day Schedule confirmation table, which lists all patients that have scheduled a session for a specified day, with contact data so someone can call and confirm the appointment.

Let's review them in more detail:

1. Clinical Summary:

Patient Information:

Date: `<today's date>`

Name:	`<Name Surname>`		
Date of birth:	`<dob>`	Gender:	`<gender>`
Address:	`<address1>`		
City	`<city>`	Sate:	`<state>`
Postal Code	`<zip>`		
Phone Numbers:			
	(xxx) xxx-xxxx	Home number	
	(yyy) yyy-yyyy	Work Number	
	(zzz) zzz-zzzz	Cell Number	

Clinical Information:

Date of admission: <date of admission> Date of termination: active

Therapist: <Therapist>

Diagnosis: <diagnosis>

Head Doctor: <medical Doctor> Tel.: <phone number>

Psychiatrist: <psychiatrist> Tel.: <phone number>

Medications:	Abilify 20 mgrs./day	From: 09/12/2012	until: Today
	Prozac 0.5 mgrs/day	From: 05/23/2012	Until: 09/12/2012

End of Clinical Summary

This report links a patient ID with all contact information and the corresponding phone numbers, MD, psychiatrist, medication history and assigned diagnosis. The most recent medications are listed first. Notice that if there is no date of termination, the report will write “Active”. In the same line, if a medication has no termination date, the form will write: “Today”. Also, notice that this report states today's date in the upper right hand corner, so we know that the report is valid up to that day. Finally, notice that the report ends with a line under which we write : “End of Clinical Summary”. The purpose of this is that, if this report had more than one page, you can easily find where it ends. Picture yourself producing several of these reports and you can see why a clear end like this could be useful.

2. Patient history of services and payment summary:

Patient History of Services Summary
rendered between <date1> and <date2>

Date: <today's date>

Name: <Name Surname>

Date of services:

Date:	Status:	Value:
08/12/2015	No Show	$35
08/19/2015	Patient Canceled	$0.0
08/26/2015	Performed	$35
09/03/2015	Performed	$35
09/10/2015	Performed	$35
09/17/2015	Therapist Canceled	$0.0
09/24/2015	Performed	$35
09/30/2015	Performed	$35
	Total:	$210

Payment History

Name: `<Name Surname>`

as of `<today's date>`

Payments Received on:	Amount:
08/26/2015-13:30:22	$35
09/03/2015-13:34:53	$35
09/07/2015-11:12:33	$35
09/10/2015-13:32:41	$35
09/24/2015-13:37:12	$10
09/30/2015-13:29:34	$55

Total Payments: $205

Balance Summary

total services amount: $210 - total payments: $205 = -$5

End of Patient History of Services Summary

The report renders all sessions scheduled in the top half and all payments made in the bottom between a specified time window (that is, between `<date1>` and `<date2>`). The amounts are added. Then a balance is calculated. Patient and date information are repeated to simplify lecture (maybe this gets to be a long list). We also display the date the query is run and terminate the report with a clear notice.

3. Therapist's summary:

Therapist's Services Summary:

Date: `<today's date>`

Therapist: `<Therapist>`

rendered between `<date1>` and `<date2>`

Date	Patient	Status	Value
07/01/2015	`<surname>`	No Show	$20
07/01/2015	`<other>`	Performed	$20
07/01/2015	`<andother>`	Performed	$20
07/01/2015	`<yetother>`	Canceled	$0.0
and so on...			
07/30/2015	`<lastone>`	Performed	$20

Total Services: $1207

End of Therapist's Services Summary

This report is built with all scheduled sessions for a particular Therapist's ID and a time range. The list is chronologically ordered and, within the same date, alphabetically ordered by the surname of patients. The report would never write “and so on...” this is just something I wrote so that you get the idea. The ellipsis represents a continuation of the list of events. Date and end of report are also clearly identified.

4. Clinic's summary of services rendered and payments received:

Clinic's Summary of Services

rendered between `<date1>` and `<date2>`

Date	Therapist	Status	Value
09/15/2015	`<Atherapists>`	No Show	$35
09/15/2015	`<Atherapist>`	Performed	$35
...			
09/15/2015	`<Btherapist>`	Performed	$35
09/15/2015	`<Btherapist>`	Canceled	$0.0
...			

Total Services: $5700

Summary of Payments:

rendered between `<date1>` and `<date2>`

Payment Received On:	By Patient	Value:
09/15/2015-13:30:22	`<Apatient>`	$35
09/22/2015-13:34:53	`<Apatient>`	$35
etc. for `<Apatient>`		
09/16/2015-11:12:33	`<Bpatient>`	$35
09/23/2015-13:32:41	`<Bpatient>`	$35
09/24/2015-13:37:12	`<Bpatient>`	$10
etc. for `<Bpatient>`		
09/30/2015-13:29:34	`<Cpatient>`	$55
etc. for `<Cpatient>`		

Total Payments: $5670

End of Clinic's Summary of Services

This report has two components: First it states all sessions performed by all therapists (in alphabetical order) in a selected time range. The form includes the status of the session (performed, canceled, etc) and the value charged to the client. The 'Value' column has a total sum at the bottom. The second component lists all payments made for the same time

range, ordered by patient's surname and chronology of payment; and includes the timeestamp and the actual values payed by them. A sum of all payments is also done at the end of the 'Value' column.

5. Next day Schedule confirmation table

Future Sessions Confirmation Table

For Date: `<selected day>`

Name of patient: `<name surname>` **Tel.:** `<phone number>, [<phone number>]`

Therapist:`<therapist>`

Time slot: `<hour>` ❑: Confirmed

❑: Re-scheduled for: _____ ❑: Canceled

Name of patient: `<name surname>` **Tel.:** `<phone number>`

Therapist:`<therapist>`

Time slot: `<hour>` ❑: Confirmed

❑: Re-scheduled for: _____ ❑: Canceled

Name of patient: `<name surname>` **Tel.:** `<phone number>, [<phone number>]`

Therapist:`<therapist>`

Time slot: `<hour>` ❑: Confirmed

❑: Re-scheduled for: _____ ❑: Canceled

End of Confirmation Table

The boxes will have no other functionality than letting the user of a printed version of this report check the result of the confirmation. This report will list every patient scheduled for the specified day.

The final layout of the actual reports we will build with Base might not be identical to these drafts but will need to include all the information and the logical elements described here. In the same line, although this tutorial will not produce all of these reports, it will provide you with all the tools you need for making them. It would be a good exercise if you do!

Turn on the computers

We are finally ready to sit in front of the computer and create our application with Base. We have a database design, we know the forms that we need and we know the reports that we want. We have also built variable definition lists for all tables. All the thinking about our particular database has been done so now we can focus on the complexities of coding it with Base.

Our first step will be to actually build the tables. You can do this with the graphical user interface of design mode, with the help of the wizard or by coding them with SQL commands. There are excellent tutorials that explain how to build your tables using the first two methods and I truly hope that you study them. However in this tutorial we are going to create our tables using SQL commands. This might seem less glamorous than creating the tables by using the graphical interface (GUI), but you will quickly notice how it is as easy and avoids some limitations of the GUI.

We will then build the forms for this database with all the complex functionality our design calls for like radio buttons, drop-down lists, forms that need to access two tables and link records, etc.

Finally, we will produce the reports that, as you have seen, also focus on advanced functionalities. This is because these reports not only retrieve particular records that depend on extra information only available when you request the report (like the name of a particular client or a time frame of performed or paid sessions) but also offer summary information.

Well, we have promised to cover a lot. Let's get started!

Part III

Things you need to do while coding a database with Base.

Contents:

Chapter 7

Creating tables in Base with SQL

At this point we have a UML diagram of our database, complete with tables and connections; we have the corresponding table definitions lists, we have a design for our forms and a design for our reports. Now it'-s time to turn on the computer. Go ahead! In fact, most of the text in this part will not make much sense unless you are following the “click on this, then click on that” instructions that we provide. Take your time and study the screens and dialog boxes with the aid of the descriptions made here.

General Overview:

The first thing that you will notice when you open Base is that the Database Wizard pops up. The interface has two columns. The small one to the right describes the steps the wizard will walk you through. The bigger column offers the different options available to you. First the wizard asks you if you want to create a new database, open a previous one or connect to an existing database. By selecting “new database”, Base will work with the embedded HSQL database engine.

In the second step (Save and proceed), Base asks us first if we want to register the database with OpenOffice.org. This doesn't mean that your information might be accessible to other folks using or developing the OpenOffice.org suite. What it does mean is that other applications of OpenOffice.org, like Writer or Calc, in your own computer, will be aware of the existence of this new database and, therefore, will be able to interact with it -for example, to produce and print labels. With all confidence, mark “yes”. We will not be using the tables wizard, so leave the default and hit “finish”. For the last step, chose a name for your database and save.

Now the database interface opens. Below the menu bar and the tool bar you will find the screen divided in 3 columns. The first one to the left is very narrow and has the heading: “Database”. You will see 4 icons with the following text, correspondingly: Tables, Queries, Forms and Reports. The first one gives you access to the tables in your database and the tasks that you can perform on them. Click the “Tables” icon and you will see the available tasks appear in the second column. If you position your cursor on top of any of these tasks, a description of what they do appears in the third column. Pretty nice, eh?

One third down the page you can see a horizontal gray bar with the text “Table”. There is where the name of each table you build will be stored, so that you can access any of them by clicking on it. Because we have not built any tables so far this section is now empty.

If you now click on “Queries” in the first column, you are taken to the queries page, and so on for Forms and Reports. Each page offers you the available tasks, a brief description of the task under the mouse cursor, and the bottom section that stores your work.

You will see that there is always more way than one to do the same task in Base. For example, “Create Table in Design View” and “Use Wizard to Create Table” (in the Tables page) both allow you to create tables for your database. In this case, the difference is that the wizard will make the task easier while Design View gives you more flexibility. You still have a third option for doing this: SQL commands, which gives you the most flexibility and control.

Creating tables and relationships with SQL commands.

We will start by creating the tables. In this tutorial we are going to use SQL commands to build them. Base accepts SQL commands in two distinctive windows: The most common application of SQL is for creating queries, done in the “Queries” page, which is where you should be now. Let's analyze this for a moment. You can see that the third task is “Create Query in SQL View...” next to an icon that reads “SQL”. We will see later that queries start with the SQL instruction “SELECT”. This instruction does not modify (or create or delete) your data or data structure in any way, it just displays the available data in the fashion you specify with the “SELECT” command. For this reason, Base will report an error for any SQL query that does not start with “SELECT” made in the Queries page. Let's now move to the “Tables” page.

For creating tables we will need another window for SQL instructions. To reach it click “Tools” at the menu bar and then click on the option “SQL...”. The “Execute SQL Statement” window opens. You will find a cursor blinking at the “Command to Execute” box. This is where we will type the instructions to create the tables. SQL commands in this window do modify your data and do not need to start with the instruction SELECT.

You will have to type the complete following set of instructions. Pay attention to quote signs, capitalization and syntax. You can also copy and paste these instructions from the link:

http://documentation.openoffice.org/servlets/ProjectDocumentList?folderID=778

When you enter the instructions, the “Execute” box highlights. When you are done, click on it. This will run your commands and create the tables. After a few seconds, the window will inform you that the instructions have been executed. Other than that there will be no visible signs on your screen. If you now go to the “View” menu and click on “Refresh Tables”, a complete list of the tables you have created will appear in the lower section of your screen. (You can also try exiting and re-entering the “Tables” view). Try this now.

Before analyzing the SQL instructions that have just created your tables, I want to show you that the relationships between them have also been created. Click on “Tools” and then on “Relationships...”. You will see a long line of boxes and some lines connecting them. Each box is a table and their different attributes (columns) are listed inside. Some boxes have scroll bars. This is because they have more attributes than the number that can appear in the default size for the boxes. You can readjust their sizes (width and height). You can also reposition the boxes so that the tangle of lines connecting them becomes

easier to read. You will also notice that these lines connect the primary keys (indicated with a small yellow key symbol) with the foreign keys and display the corresponding cardinalities (although simplified), just like a UML diagram. Isn't this fine? In fact, by the time that you finish tidying the relationship view, it should appear just like the UML diagram in our example. If you right-click on these lines and select 'Edit', a dialog box pops up where you can set Delete options as they were explained in page 10.

Let's analyze the SQL instructions that make all this possible. Take the time to read through this and see if you can derive any sense from it. We will analyze them together afterward:

```
DROP TABLE "Patient Medication" IF EXISTS;
DROP TABLE "Medication" IF EXISTS;
DROP TABLE "Payment" IF EXISTS;
DROP TABLE "Schedule" IF EXISTS;
DROP TABLE "Assignment" IF EXISTS;
DROP TABLE "Therapists Number" IF EXISTS;
DROP TABLE "Phone Number" IF EXISTS;
DROP TABLE "Patient" IF EXISTS;
DROP TABLE "Psychiatrist" IF EXISTS;
DROP TABLE "Medical Doctor" IF EXISTS;
DROP TABLE "Therapist" IF EXISTS;

CREATE TABLE "Psychiatrist" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(25) NOT NULL,
"Surname" VARCHAR(25) NOT NULL,
"Gender" CHAR(6),
"Street and number" VARCHAR(50),
"City" VARCHAR(25),
"Postal code" CHAR(5),
"State" CHAR(2),
"Phone Number" VARCHAR(10)
);

CREATE TABLE "Medical Doctor" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(25) NOT NULL,
"Surname" VARCHAR(25) NOT NULL,
"Gender" CHAR(6),
"Street and number" VARCHAR(50),
"City" VARCHAR(25),
"Postal code" VARCHAR(5),
"State" CHAR(2),
"Phone Number" VARCHAR(10)
);

CREATE TABLE "Patient" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(25) NOT NULL,
"Surname" VARCHAR(25) NOT NULL,
"Gender" CHAR(6),
"Date of Birth" DATE,
```

```
"Street and number" VARCHAR(50),
"City" VARCHAR(25),
"Postal code" VARCHAR(5),
"State" CHAR(2),
"Diagnosis" VARCHAR(60),
"Medical Doctor ID" INTEGER,
"Psychiatrist ID" INTEGER,
"Time of registry" TIMESTAMP,
CONSTRAINT "CK_PAT_GNDR" CHECK( "Gender" in ( 'Male', 'Female' ) ),
CONSTRAINT FK_PAT_PSY FOREIGN KEY ("Psychiatrist ID") REFERENCES
"Psychiatrist" ("ID Number"),
CONSTRAINT FK_PAT_DOC FOREIGN KEY ("Medical Doctor ID") REFERENCES
"Medical Doctor" ("ID Number")
);

CREATE TABLE "Phone Number" (
"Phone ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT NULL
PRIMARY KEY,
"Patient ID" INTEGER NOT NULL,
"Number" VARCHAR(10),
"Description" VARCHAR(10),
CONSTRAINT FK_PAT_PHN FOREIGN KEY ("Patient ID") REFERENCES "Patient" ("ID
Number")
);

CREATE TABLE "Therapist" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(25) NOT NULL,
"Surname" VARCHAR(25) NOT NULL,
"Gender" CHAR(6),
"Street and number" VARCHAR(50),
"City" VARCHAR(25),
"Postal code" VARCHAR(5),
"State" CHAR(2),
"Tax number" VARCHAR(20),
"Academic degree" VARCHAR(25),
"License number" VARCHAR(15),
"Hiring date" DATE NOT NULL,
"Termination date" DATE,
CONSTRAINT "CK_THP_GNDR" CHECK( "Gender" in ( 'Male', 'Female' ) ),
CONSTRAINT "CK_TERM_DT" CHECK( "Termination date" > "Hiring date" )
);

CREATE TABLE "Therapists Number" (
"Phone ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT NULL
PRIMARY KEY,
"Therapist ID" INTEGER,
"Number" VARCHAR(10),
"Description" VARCHAR(10),
CONSTRAINT FK_THP_PHN FOREIGN KEY ("Therapist ID") REFERENCES "Therapist"
("ID Number")
);

CREATE TABLE "Assignment" (
"Assignment ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Patient ID" INTEGER NOT NULL,
```

```
"Therapist ID" INTEGER NOT NULL,
"Date assigned" DATE DEFAULT CURRENT_DATE NOT NULL,
"Date case closed" DATE,
CONSTRAINT FK_PAT_ASMT FOREIGN KEY ("Patient ID") REFERENCES "Patient"
("ID Number"),
CONSTRAINT FK_THP_ASMT FOREIGN KEY ("Therapist ID") REFERENCES "Therapist"
("ID Number"),
CONSTRAINT "CK_CLOSE_DT" CHECK( "Date case closed" >= "Date assigned" )
);

CREATE TABLE "Schedule" (
"Schedule ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Assignment ID" INTEGER,
"Slot date" DATE NOT NULL,
"Slot hour" TIME NOT NULL,
"Status of the session" VARCHAR(20),
CONSTRAINT FK_SCH_ASMT FOREIGN KEY ("Assignment ID") REFERENCES
"Assignment" ("Assignment ID")
);

CREATE TABLE "Payment" (
"Payment ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Patient ID" INTEGER NOT NULL,
"Date and time of Payment" TIMESTAMP(6) DEFAULT CURRENT_TIMESTAMP,
"Amount" DECIMAL (10, 2) NOT NULL,
"Notes" VARCHAR(100),
"Result" CHAR(6) NOT NULL,
CONSTRAINT FK_PAT_PYMNT FOREIGN KEY ("Patient ID") REFERENCES "Patient"
("ID Number"),
CONSTRAINT CK_DBT CHECK("Result" IN ('DEBIT', 'CREDIT'))
);

CREATE TABLE "Medication" (
"Medication ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Name" VARCHAR(30) NOT NULL,
"Description" VARCHAR(256)
);

CREATE TABLE "Patient Medication" (
"Patient ID" INTEGER NOT NULL,
"Medication ID" INTEGER NOT NULL,
"Dosage" VARCHAR(50),
"Start date" DATE DEFAULT CURRENT_DATE,
"End date" DATE,
CONSTRAINT PK_PAT_MED PRIMARY KEY ("Patient ID", "Medication ID" ),
CONSTRAINT FK_MED_PAT FOREIGN KEY ("Medication ID") REFERENCES
"Medication" ("Medication ID"),
CONSTRAINT FK_PAT_MED FOREIGN KEY ("Patient ID") REFERENCES "Patient" ("ID
Number"),
CONSTRAINT CK_END_DT CHECK( "End date" >= "Start date" )
);
```

The first thing one can notice is that this set has two types of instructions:

DROP TABLE

and

CREATE TABLE

in that order. The reason for this is that if you find the need to modify your database structure in any way (add a table, omit attributes, change connections, etc) and you run the modified SQL instructions, the DROP TABLE command will start by erasing the previous tables, in effect allowing you to start from scratch. The down side of this is that you will lose any data you might have entered into the tables.

Notice that the instructions use capitalized letters and that the name of the tables use caps and smalls and are enclosed within quotes. Without the quotes, Base will consider all letters as caps. This feature relieves the programmer from jumping from lower to upper case too often. Unfortunately, it also creates potential problems like lost tables or columns impossible to find. We explain this in more detail later (see What SQL window are you, anyway?).

Also note that the imperative tone: “DROP TABLE” is toned down by the inclusion of a conditional statement “IF EXISTS”. This way, if the table didn't exist to begin with (like when you run these commands for the first time) then the application will not prompt an error message. For the same reason, be aware that it will also not prompt an error message if you mistype the name of the table, making you believe that you have erased a table that you really haven't, so type carefully or check this issue if things are not running the way you want.

Finally, note that all the instructions end with a semicolon: “;”. This is the SQL equivalent of a period in the English language and its omission is bound to create confusion.

The CREATE TABLE instruction starts by providing the name of the table to be created, also within quotes and using caps and small letters, and then describes the particular characteristics this table will have: column names, type of variables used, relationships, the way to handle deletions, etc. These instructions are written within parentheses. It then finishes with a semicolon, more or less like this:

```
CREATE TABLE "Name of the Table" ("Name of Column" COLUMN ATTRIBUTES,
etc.);
```

Inside the parentheses you specify first the name of each column, also within quote signs, and then the column attributes, like the kind of variable you are using (e.g.: integer, varchar, date variable types, etc.) and attributes like not null or auto increment, etc. Note that the instructions inside the parentheses are separated with a comma: “,” and consequently the last instruction does not have one.

At this time, please note the close resemblance of the SQL instructions with the Variables Definition Lists we made earlier and how the former can be translated to the later with very little difficulty (so now you know why we insisted so much on them). In fact, SQL

can be read very much like a formalized English. Go back and read some instructions creating tables and see if you can identify the components that we talk about here.

Maybe you are asking: Do I now need to learn SQL? To be exact, you should; and it can help you greatly in building good Base databases providing even greater flexibility than the wizard or design views. On the other hand, when you build tables, all your instructions will conform, more or less, to the same structures that appear in this example. Only the name of the tables or their attributes are bound to change (although there will be plenty of recurrences) and you will be selecting within a known set of column attributes, most of which have been described so far. This means that if you understand the instruction set we use in this example, you should be able to handle most definitions for your own databases without requiring a college degree. So take some time and analyze the way the different tables have been built, and start developing a sense of how SQL works and the syntax and grammar it uses.

Possibly the more puzzling instruction is CONSTRAINT found inside the CREATE TABLE command. This is a very useful instruction that allows you to, for example, set the connections between the tables, indicating which column is the primary key and which are the foreign keys. It can also allow you to set restrictions, for example, making sure that only M or F be accepted as gender entries, rejecting other input, or making sure that values in one column are always smaller, greater or equal than values in other columns, rejecting entries that do not conform to those rules. This is very useful to ensure data integrity.

The CONSTRAINT instruction is followed by the name of the constraint. This name is chosen by us, just like we gave a name to the tables or to their columns. In this case, all names are given in caps, with underscores separating components of the names. This way we don't need to use quote signs or worry about capitalization. For example, find the following name in the last table of the instruction set:

`FK_PAT_MED` that appears in:

```
CONSTRAINT FK_PAT_MED FOREIGN KEY ("Patient ID") REFERENCES "Patient"
("ID Number")
```

We came up with this name **`FK_PAT_MED`** to, somehow, mean that this constraint creates a foreign key in the patient-medication table. After the name comes the instruction that, in this case, defines that the “Patient ID” column in this table is a foreign key and holds the value of the “ID Number” column found in the “Patient” table. Now, that wasn't so difficult. All your foreign keys will be defined with this same pattern, just changing the names of your columns and the tables they reference. Of course, you will need to think of unique names for each declaration.

The CONSTRAINT names we chose can look a bit cryptic, but made sense to us because we wanted to capture a rather complex relationship with few characters. Of course you can define your own names, even using quotation marks and small and cap characters if you want to. For example, you could have written:

```
CONSTRAINT "Patient Foreign Key in Medication Table" FOREIGN KEY
("Patient ID") REFERENCES "Patient" ("ID Number")
```

At this point you could be asking: Why do we need to name CONSTRAINTS just like we name Tables and Columns? Because in the future, as you optimize your database or adapt it to a new business practice, you might want to change the behavior of a particular constraint, maybe even drop it altogether (but without erasing the data you have entered in the tables so far). This would be impossible if you don't have a way to reference the constraint that you want to modify. The easiest way for doing this is to give a name to the constraint when you create it.

Instructions that shape the type of CONSTRAINT you are building include: PRIMARY KEY, FOREIGN KEY, REFERENCES and CHECK.

Let's see other examples (go and find the **`CREATE TABLE`** instructions that harbor these constraints):

- `CONSTRAINT CK_END_DT CHECK( "End date" >= "Start date" )`

This instruction checks that the “End date” column in this table always stores a value that is greater (newer) or at least equal to the value in the “Start date” column. This constraint received the name **`CK_END_DT`** which, in our mind, stands for “check end date”.

- `CONSTRAINT PK_PAT_MED PRIMARY KEY ("Patient ID", "Medication ID" )`

This instruction establishes that the primary key of this table is formed by “Patient ID” and “Medication ID”, which is to say that this table uses a compound key formed by exactly these two attributes (columns).

- `CONSTRAINT CK_DBT CHECK("Result" IN ('DEBIT', 'CREDIT'))`

This constraint is a bit more tricky to read. First we need to know that when using Base with the embedded HSQL engine, the names of tables, columns or constraints that are being referenced need to be enclosed in double quotation marks. Meanwhile, a string of characters (any string of characters) will be indicated by single quotation marks. With this in mind we can understand that the CK_DBT constraint checks that the column “Result” can only receive the values 'DEBIT' or 'CREDIT' and will reject any other value (sequence of characters) that the user attempts to enter.

With this in mind now explain to me what does this mean:

```
CONSTRAINT CK_PAT_GNDR CHECK( "Gender" IN ( 'Male', 'Female' ) )
```

Don't forget to identify in what table these instructions appear in.

- `CONSTRAINT IDX_PAT UNIQUE ( "Patient ID" )`

This constraint, that we named IDX_PAT, makes sure that the “Patient ID” stores unique numbers. Should we want to eliminate this restriction in the future, we could go to the “Execute SQL Statement” window (found in the Tools menu under “SQL...”) and run the command: **`DROP IDX_PAT.`** However (and to avoid confusion) I have eliminated this instruction from the code set already and you won't find it in any table.

If you happen to declare a constraint instruction that references a table that you have not yet created, Base will prompt an error message. This is relevant for constraints that reference other tables (than the one in which is being declared) like Foreign Keys. Review the SQL code again and note how the order on which tables are created avoids this problem, In short, when referencing other tables, you need to make sure that they already exist.

I recommend that you go back to the instruction set and read the CONSTRAINT instructions again with this new information. Don't they make more sense now?

Let's focus now on the way to define the columns inside the CREATE TABLE command. Let's review the following examples:

- **`"First Name" VARCHAR(25)`**

This instruction creates a column called “First Name” (note the use of double quotation marks for the name) which will accept **up to** 25 alphanumeric characters. Aha! Now is when that long and boring analysis of variable types we made before starts to make sense. Can you tell what happens if I make an entry to “First Name” that only uses 12 characters?

- **`"Gender" CHAR(6) NOT NULL`**

This instruction creates a column called “Gender” that accepts **exactly** 6 alphanumeric characters. It then adds the instruction “NOT NULL”, which means that Base will make sure that you enter a value here. Can you describe the other available conditions we can set our columns to?

- **`"Patient ID" INTEGER NOT NULL`**

This instruction creates a column called “Patient ID”, assigned to receive integer numbers. Can you tell me what is the biggest number I can enter here? Could I enter a negative number? Later I add the instruction “NOT NULL” to make sure that there is always a “Patient ID” value.

- **`"Amount" DECIMAL(10,2) NOT NULL,`**

This instruction could strike you as strange. You can easily read that we are creating a column called “Amount” that is to be not null and that is of the decimal type; but what are those numbers within parentheses? In alphanumeric variables you specify the number of characters you want inside parentheses but Decimal is a number type of variable, not alphanumeric. Well, the answer is that you can specify the size of the column for any type

of variable, be it a number or alphanumeric. You can also define the precision, if you need to, with the number after the comma inside the parentheses. In this case, we have defined that the Decimal variable type (which can store very, very large numbers) used in "Amount" be organized in a column 10 digits wide and that it can store numbers with a precision of one hundredth of a point, that is, decimal places that range from 00 to 99 (which are two decimal places, hence the number two). In other words, "Amount" displays in a column of 10 numbers, where the last two numbers represents 1/100 numbers. If you were to enter the number 12.037, Base will store 12.04. This definition makes a lot of sense when you use a column to store money amounts.

- **`"Medication ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT NULL PRIMARY KEY`**

This is an important pattern as it defines primary keys. This instruction generates the "Medication ID" column and assigns it an integer variable type. It then adds the instruction that it should be generated automatically, the first entry being number zero, it should be NOT NULL and is the Primary Key of this table. Although not explicit, this pattern also ensures that this column has unique numbers.

At this point you could be asking: Do I now need to create a CONSTRAINT that identifies "Medication ID" as a Primary Key? No you don't. This has already been done in the definition of this column. Again, note that there is more than one way to do things with Base.

If you review the SQL code now you will find that all tables have a primary key (one of them even has a compound primary key, can you tell which?). You might remember that we said in Part II that, in theory, not all tables required one. So why? Later on, when we build forms, we will simplify our work greatly with the use of wizards and they might not work properly if they don't have these keys as reference. In consequence it is better and some times required to include them.

- **`"Start date" DATE DEFAULT CURRENT_DATE`**

This is an interesting declaration and we will review it again later when talking about forms and reports. The first part is very easy to understand: create a column called "Start date" that will store a DATE variable. The DEFAULT setting makes sure that this variable is populated upon creation. But, populated with what? With the content of the CURRENT_DATE function. CURRENT_DATE provides date information kept by your computer's internal clock. When we later create a new record (using forms), the DEFAULT instruction will retrieve the value stored by CURRENT_DATE and store it in "Start Date" automatically.

- **`"Date and Time of Payment" TIMESTAMP(6) DEFAULT CURRENT_TIME`**

Here we are creating a column called "Date and Time of Payment" that will be populated automatically by the contents at CURRENT_TIME upon creation of the record. I want to focus on the number six within parentheses. That is an index of precision. **`TIMESTAMP`** ac-

cepts a precision value of 0 or 6. 0 means that the variable will hold no sub-second data. If you don't indicate precision, the default is 6.

Let's wrap-up all what we have seen so far in a more formal definition of the **CREATE TABLE** instruction.

First, a CREATE TABLE instruction starts with **CREATE TABLE** followed by a name for the table. Then, within parentheses, you specify one or more column definitions and one or more constraint definitions, all separated by commas. Finally you indicate the end of the instruction with a semicolon.

If you were looking in the Wiki or other source for information on the **CREATE TABLE** command, you would come across a formal representation of the previous definition, which would look like this:

```
CREATE TABLE <name> ( <columnDefinition> [, ...][,
<constraintDefinition>...]);
```

The first thing you will notice are the angle brackets ("<" and ">"). They indicate that "name" or "columndefinition", etc., stand for names, or values, that you are going to chose, whatever those names or values will be. Then we have the square brackets ("[" and "]"). They enclose instructions that are not mandatory, that is, that we can chose to include or not. So for instance, this formal definition implies that we need at least one column definition but having constraint definitions is optional. Lastly, the ellipsis (those three points together, "...") mean that the last instruction is repeated, that is, that we can make more column definitions. Because it is enclosed between square brackets, in this case it also means that having more than one column definition is optional. Note the comma before the ellipsis. This reminds you that you must separate column definitions with commas.

Why are we going through this? Because after you finish this tutorial, you might want to know more about Base and the SQL instructions it uses. When you review the Wiki and other sources, this is the syntax that you are going to find.

Now, "**columnDefinition**" and "**constraintDefinition**" also have their formal representations. Let's see:

Column Definition:

```
<columnname> Datatype [(columnSize[,precision])]
    [{DEFAULT <defaultValue> |
    GENERATED BY DEFAULT AS IDENTITY
    (START WITH <n>[, INCREMENT BY <m>])}] |
    [[NOT] NULL] [IDENTITY] [PRIMARY KEY]
```

You should be able to read that after the column name, the only required parameter is the data type (integer, varchar, etc.). You can define column size and precision if you want to

(and even precise column size without defining precision). What does that vertical line (also called pipe, "|") stand for? That means that you have two or more options for a statement. For example, in defining a column you can write: **GENERATED BY DEFAULT AS IDENTITY (START WITH 0)** or you can write: **NOT NULL IDENTITY PRIMARY KEY.** Of course, if you wanted the primary key incremented by a value other than 1, then you would want to use the first option and include the increment that you want. So, in short, the pipe stands for a logical OR.

Let's now check the very powerful constraint definitions:

```
[CONSTRAINT <name>]
    UNIQUE ( <column> [,<column>...] ) |
    PRIMARY KEY ( <column> [,<column>...] ) |
    FOREIGN KEY ( <column> [,<column>...] )
    REFERENCES <refTable> ( <column> [,<column>...])
    [ON {DELETE | UPDATE} {CASCADE | SET DEFAULT | SET NULL}]
    CHECK(<search condition>)
```

Notice how constraints allow you to set column properties, search for conditions to match and set delete and update options like those available through the Relationships view. I hope that this review motivates you to go and find out more about Base, the HSQL engine and SQL.

What SQL window are you, anyway?

So far we have described two instances in Base where we can input SQL commands: the "Queries" page and the Command box. Throughout this chapter we have described some differences between them that I want to summarize here:

First, the "Queries" page is reached by clicking on "Queries" in the narrow column to the left under the heading "Database" while the Command Box is found in the "Execute SQL Statement" window called by the SQL icon in the Tools Menu.

Second, the instructions in the "Queries" page do not transform your data or data structure. This window only receives instructions on how to display the information that you already have in your database. It checks to see if your SQL statements start with the command "SELECT" and will prompt an error message if they don't. Meanwhile, the instructions through the Command box can transform your data or data structure, that is, you can create, delete or modify your tables, columns and constraints and, in the process, delete data. You can also input instructions that create, delete or modify records. The Command box also accepts "SELECT" commands.

The third important difference has to do with the way they handle names. As you know by now, tables, columns and constraints receive names that you assign in order to reference them later. The proper syntax requests that those names be enclosed in double quotes, particularly if you are using mixed case names and spaces (like "Patient ID"). Now, mixed case names defined in the command box that do not use quotes are silently

converted to all uppercase but mixed case names used in the "Queries" page that are not enclosed in quotes are passed as they are typed, without a conversion.

This can create some confusion and you better be aware. Let's say that you define the following table through the Command box:

```
CREATE TABLE TransactionType (
Tran_Type_ID INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
Tran_Description VARCHAR(50),
Debit CHAR(6) NOT NULL,
CONSTRAINT CK_DBT CHECK(Debit IN ('DEBIT', 'CREDIT'))
);
```

and you later, through the same Command box, issue the following instruction:

```
SELECT * FROM TransactionType;
```

Both sets of instructions should execute without error. However, when you are later creating a report in the "Queries" page and issue the same instruction:

```
SELECT * FROM TransactionType;
```

You would receive the error message: "Table TransactionType does not exist". What happened here?!

If you notice, the **CREATE TABLE** command in this example does not use quotes when assigning a name to your table, so although you named your table "TransactionType" (without the quotes), Base really assigned it the name "TRANSACTIONTYPE" (all upper case). When you issue the **SELECT** command through the Command box, although you again used mixed case, Base silently converted the name to all caps and didn't have trouble finding the table. However, in the "Queries" page you repeated the name as you wrote it but this time Base did not convert it to all upper case, despite the fact that you didn't use quotes to encase the name. So now the application is really looking for the table "TransactionType" and not finding it, because you have really created the table "TRANSACTIONTYPE". The same thing is bound to happen with the column names "Tran_Type_ID", "Tran_Description" and "Debit", which have also been defined without their double quotes.

Notice that the **CONSTRAINT** defined in the example uses a name with all uppercase and no quote signs. The silent conversion that will take place here is irrelevant because it leaves the name the same.

You can chose to name all your objects (tables, columns, constraints) with uppercase and forget about quotes but it is considered better practice to use quotes. This practice also gives you a bigger set of possible names. In this tutorial, constraints are the only objects that we will name using only all caps.

Recapitulating and to be more precise, any string in the Command box that is not enclosed in quotes will be transformed to upper case. For example:

```
DROP TABLE "Patient" if exists
```

will be converted by Base to:

```
DROP TABLE "Patient" IF EXISTS
```

which gives you a good hint on why the Command box behaves this way.

Chapter 8

Producing our forms

At this moment we have a sound database design, which took a lot of thinking, and that we have encoded using SQL language to create the tables and their relationships with Base. Now we are going to focus on creating forms that will help us populate these tables swiftly while minimizing errors in data entry.

The first thing will be to remember the considerations mentioned in the first part of this tutorial regarding the need to make forms that are unambiguous on what data are we supposed to provide for each field and to minimize errors in the input by offering pre-made options that are easy to indicate. In this tutorial we will focus on the following ways to simplify or automate data entry that are very useful: radio buttons, drop down lists, sub forms, default values and some tricks for time entry. We will further describe these options as we work with them.

Let's get to work.

On the Base document where you have the tables for our psychotherapy clinic, click on the icon to the left over the word “Forms”. You will see the following options appear in the *Task* panel (upper panel to the right): *Create forms in design view* and *Use wizard to create forms*. However there is still a third way not made explicit yet: Defining forms using SQL. At this moment it seems necessary to explain something:

Using SQL for creating tables or entering data requires that we know SQL. In contrast, using the Wizard just requires that we understand what are we being asked and chose the appropriate options. The result using SQL will be as good as the underlying engine that we are using, in this case the HSQL embedded database system. In all merit, HSQL is a very good implementation of the SQL standard. Using the wizard or design view, on the other hand, will only be as flexible as the software mediating our options and the resulting SQL code can be. In all honesty, the coders at OpenOffice.org have done a terrific job developing this easy to use graphic interface, but for its very nature, graphical interfaces can not always accommodate the complexities and intricacies that raw code can denote. Also, there can always be a bug here or there that could impair the ability of the software to achieve a particular goal (which is why it is so important to report bugs and help achieve better code). So we will be forced to use SQL code when the Graphical Interface does not provide us with an option that we need or when we suspect that there could be a bug affecting us. This means, again, that in order to develop robust databases, learning SQL is necessary. In short, SQL code gives us flexibility and power, design view and the wizard give us ease of use. It will not come as a surprise to you that in this tutorial we will be using these three options.

In general, we will use the following approach to building forms: First we create forms using the wizard, which will help us get up and going quite quickly. Later we use Design View to customize them, clean them and leave them easy to read and understand. Finally we add any SQL instructions required for more complex functionalities, if necessary.

The Form Wizard.

I want to start with the form for entering the information on the psychiatrists. First, because it is simple enough not to distract us from the process of building and cleaning the form, something we will cover here and later just assume that you are repeating with the other forms, and second because it allows us to explore how to use radio buttons for data entry.

So, let's recap: click on the icon over *Forms* in the left most column in your display, the one under the heading *Database.* Now choose *Use Wizard to Create Form...*

The first thing that will happen is that a Writer document pops up, with some extra controls on the bottom, and -over it- the Form Wizard.

So this is the big surprise, a form in Base is really a Writer document, the one you use for editing and printing documents. Of course, the elements that we will include in this document, like text boxes and radio buttons, need to somehow be connected to the database that we are working on. Don't worry about this as the Form Wizard will take care of this connection. But the fact that this is a writer document means that we can enjoy all the flexibility usually available for organizing the layout of our written documents. If you remember, the draft forms I made for part II in this tutorial were worked with Writer -although at that moment they were not functional and were made only to illustrate the layout- which means that, if we take the time we can replicate them completely if we want to, now fully functional and all.

The Form Wizard is organized in two columns: the one to the left names the different steps the wizard will walk us through. The panel to the right will ask us the questions that each step needs to know about.

In the first step we need to indicate where the fields that we will use are coming from, that is, what columns from what table will be associated to this form. In the drop down box you will find a list of all the tables available. Find the option “table: psychiatrist” and select it now. Note that on top of the drop down box the caption reads: *Table or Queries.* This is because the result of a Query is also composed of columns and you can use them with the wizard as if it were another table (we will be using this later).

Upon selecting the psychiatrist table you will see that a box in the bottom to the left populates with all the column names that belong to the table. We are supposed to select all or some of these for which we want a field in our form. For the sake of simplicity let's just pick them all. You do this by clicking on the button with the sign “>>”. Do this now.

Now all the names for the columns have shifted to the box on the right, meaning that there will be an input field for each column.

If you had wanted input fields for only some of the columns, you would have had to select them with the mouse and then click the button with the “>” sign. Let's say that you see no reason for a field for the ID number -because it will be generated by Base automatically, because the user does not need to know this value and because you don't want to leave it exposed to an involuntary (and fatal) change. In this case you would need to click every other name and then click on the “>” button, every time for each field. If this were the only attribute for which you don't want a field, you could also move all the column names to the right (with the “>>” button), select the ID number and then click on the “<” button, saving you some time.

However, for the time being, select all column names and then click on “Next”

The second step ask us if we want to set up a sub form. We will deal with sub forms later and the Psychiatrist form does not need one, so just click on “Next” without changing the content here.

Now the wizard jumps to step five and ask us what layout we want for the fields. The default uses a table configuration (third icon from the left), with the name of the columns in the header (called *Data sheet*). We will use option number one (called *Columnar -Labels Left*). Experiment clicking on the other options to get a feel of the alternatives.

Step number six creates restrictions about the use of the data by the form, either allowing the view and modification of all data or not allowing the data to be viewed, modified or some combination of these. The options are self explanatory and we will not be using this yet, so we will advance to the next step leaving the default: *This form is to display all data,* with no restrictions.

Step number seven allows you to chose the background color for the form and the style used by the text boxes that conform the entry fields. When you choose a color for the background, Base chooses colors for the fonts automatically. Your selection here will have no bearing on the functionality of the form other than make it easy to read. I believe that the offered combinations are well chosen and I like dark backgrounds with white characters. But that is me, you choose whatever you want.

The last step will ask us for a name for this form and what we want to do with it. The wizard will offer, as a default, the name of the table, in this case “Psychiatrist”. Many times this is enough but I would recommend to use a more descriptive name. Some people would add the letters “frm” before the name, to make explicit the fact that this is a form. I invite you that, this time, you change the name of the form to: “Psychiatrist Data Entry Form” just to see what happens. Later, you can use your own naming conventions. Finally, Base wants to know if it should offer the table for data entry or if it should open it in Design View for further editing. We will take the second option. Click on it and then click on “Finish”.

At this moment, the wizard closes and the form opens again, this time with text boxes for data entry and corresponding labels. The form has opened in Design View and is ready for our editing. You can also notice dotted lines forming a grid pattern. This is to make it easier to design and position elements on the page.

Design View and the Properties dialog box.

If you click on any of the text boxes or their accompanying labels, you will see small green boxes appear defining a rectangular limit. This limit includes both the label and the text box. If you try to move one, the other will drag along; if you try to modify one, chances are that you will somehow modify the other too.

At this moment I want you to double click with the left button of your mouse. A dialog box appears with the name *Properties: Multiselection.* This properties box allow us to modify the characteristics of the elements we place on our form, and will become very handy just now. Multiselection means that there are two or more elements alluded by the properties box because they have been grouped together. This is why the label and the text box are traveling together and changes on one can change the other too. If you now click elsewhere on the form, the group becomes un-selected and the Properties dialog box becomes empty, stating that no control has been selected. You can now click on another element and its properties will become displayed by the dialog box. If you close the box, you can call it again by left-double clicking the mouse while on top of an element.

Close the dialog box for now and then click (once) over an element with the mouse button to the right. Now we see the green squares again, indicating the active element, while the more familiar contextual menu appears. Near the bottom of the menu you will find the item “Group” and within it the options ungroup and edit group. Choose ungroup.

If you click on an ungrouped element now, you will see that they display the green boxes but only in relation to themselves, not extending to include the other elements. If you call the properties dialog box, you will see that it offers options that are particular to the selected element, in this case, either a label or a text box.

Let us now analyze what we see on the screen: there are nine pairs of labels and text boxes. Most of the text boxes are of the same length except for the one after *ID Number.* If you analyze the SQL code that created this table you will see that all the boxes with similar length are for some kind of text entry while *ID Number* is an integer variable. So maybe Base chose the length of the boxes depending on the type of Variable they are. We might want to change these lengths in order to make the form less intimidating and easier to read.

The other thing you can notice is that the labels describing each box give a quite accurate description of the kind of data entry that we expect. How did Base know? If you look at the SQL code that created the table, again, you will see that Base actually copied the name of the column as given by us. Because we took the time to use double quotes, gave descriptive names and used spaces between words, now these descriptive names appear

to the left of the boxes. If instead we had used more cryptic names for the columns, like PK_FST_NM, then the labels would be displaying just that. This would not be a problem however because we can easily change the caption in the label through the Properties dialog box.

Select the pair with “Number ID” and ungroup them. Now select the text box only. You can position the cursor over it and, by dragging the borders, make it wider or higher. You can also reposition the box anywhere on the form by dragging it while the cursor looks like a cross with arrow heads. Nice! With these skills you can, for example, reduce the *Postal code* and *State* fields -they don't need more than five and two characters respectively- and place them side by side so the person confronted with this form finds it more familiar and less of a chore.

Now select the label element and call it's properties dialog box. This time, the box has two tabs: *General* and *Events*. Through the *Events* tab you can program behavior in response to the events named in the dialog box. This requires some macro programming and we will not be using this feature in this tutorial. With the *General* tab we can change other properties. The Name property is automatically provided by Base and it is used to refer it with internal code (connecting the form with the database). The second property corresponds to the caption that we actually read. Change “Number ID” for “Psychiatrist Primary Key” and then hit return. The cursor goes down to the next property in the dialog box and the value is immediately changed in the label. This also happens if you take focus from this field. Try it.

I feel that the default size for the font in the labels is somewhat small. Let's change this. Find the property *Font* in the dialog box and click on the “...” button. This will call a dedicated Fonts dialog box. The first tab, “Font” allows us to change the font, the typeface and the size of the text in the label. The next tab allows us to include other effects, including changing the color of the font. Feel free to experiment with this for a while.

Chose a font size of 12 and see what happens. The text in the label is now bigger but does not display completely. This is because the boundaries set by the green squares did not change after we increased the size. Drag the squares until you see the complete text of the label. As an exercise, change the font size of every label in this form to 12 and adjust the length of the text boxes to something commensurate with the amount of data they will receive.

Now scroll to the bottom of the properties dialog box. The second to last property reads “Help Text”. By writing here you activate and populate the tool-tip. This is a yellow rectangle with helpful information that appears when you place the cursor over an element on a form. You find many examples of this in professional software or the Internet. Base automatically does this for us; we only need to insert the useful information. That is what you type here. For example, in the box for “Name” you could write: “Enter the first name of the psychiatrist”. You can now save the form edit, close it and call the form. Place the cursor over the box and see the tool-tip appear.

One important way to make forms more gentle to the eyes of the user is to limit the amount of fields to the ones he will actually have to populate. As mentioned before, there are good reasons to eliminate the Number ID field and have this form less crowded. But wait! Would not this interfere with the ability of Base to include the auto-generated primary key number? It will not. Base will continue to generate unique primary keys and include them into the records. What we are doing here is to make explicit the fields that the user is shown or not. The automatic generation of primary key numbers was coded with the creation of the table itself and is not interfered by this.

Now that we are getting a hang on the usefulness of the Properties dialog box, lets use it to program our radio buttons.

Radio Buttons.

Radio buttons offer the user a set of mutually exclusive options. The user can chose only one of them. What we want to achieve is that the elected option is entered to the corresponding column in our table. Now, pay attention: The options for radio buttons better cover all possible alternatives. If not, then at least the user should find the option to leave the field unanswered. Gender is a good candidate for using radio buttons. Granted, male and female are not all the possible options (hermaphrodites, among other less common options) but will correspond to a quite high percentage of the users. In another context however (a database for gender research, for example) this could be insufficient. So plan your radio buttons carefully. Another place where radio buttons can be used to an advantage in this example is in the table to register if the session took place or not (see the form layouts in part II).

The first thing that we need to do is actually obtain the radio buttons. You will find them in the Form Controls Menu. This menu usually pops up automatically when you open a form in Design View. If not, go to “View” menu option in the writer document where we are editing our from and select “Toolbars” (third from the top) and click on “Form Controls”.

Now a floating menu appears, with the name “Form ...”. This is a small menu that offers several icons. The icon of a circle with a black spot is our radio button. If you leave the cursor on top of it, a message appears (a tool-tip, actually!) with the name: “option button” which is how Base calls the radio buttons.

If you left-click on this icon, the cursor changes to a cross with a small box. This allows you to place the radio button somewhere in the form page. You can actually place it wherever you like. Just for clarity with this example, try to place it next to the text box with the “Gender” label, by dragging the cursor and releasing the left mouse button.

If things happen like in my computer, you should see a radio button with the caption “option button”. If you chose a dark background color (like I did), the caption in black will be difficult to read. To fix this, call the Properties dialog box for the radio button. In the general tab, change the *Label* to “Male”. Then click the *Font* button, select “bold” and

change the font size to 12. In the “Font Effects” tab, change the font color to white. That should be much better.

Let's now program the behavior that we need. The Properties dialog box for a radio button offers a tab that we have not seen yet: The *Data* tab. If you select it you will see three fields: Data field, Reference value (on) and Reference value (off). Data field indicates what column this control will reference. Because the wizard built this form for the psychiatrist table, you will find the name of all the columns for such table. Click on the list box and see them appear. Of course, select “Gender”. The Reference value (on) holds the value that will be passed to the Gender column if this radio button is selected. Write “Male” there.

Beware!: if you check the SQL code that built the Psychiatrist table you will find a **`CONSTRAINT`** that only accepts “Male” and “Female” as possible entry strings for the Gender attribute. Note that the initials are capitalized. If you write “male” (without the capitalized initial) in the *Reference value (on)*, Base will report an invalid entry error every time you select this radio button.

Now, on your own, create a second radio button. Change the label to “Female” and make it readable, relate the radio button to the “Gender” field and write “Female” in the *Reference value (on)* field.

Having created the second radio button, align both buttons so that they are at the same level (use the grid to guide you) and chose a harmonic spacing between them. Making it pretty makes data entry a bit more enjoyable.

At this point we no longer need the text box for entering the gender: the radio buttons will take care of this. But we are going to leave it for a while so that you can see the radio buttons in action. First, ungroup the label and the text box and make the box smaller (enough for 6 or 8 characters). Now take the box out of the way and place the radio buttons in its place (you can group them and move them as one element). For reference, leave the text box near by.

If you don't know how to group elements, here is how: Select one element by right clicking once on it. Next, press the *Shift* key and select another element. Now both elements appear under the same area defined by the green boxes. Repeat this to include as many elements as you need in the group. Right click the mouse while the cursor is a cross with arrow heads and in the context menu that appears, click on “Group...” and select “Group”. Now your elements are grouped.

This would be a good moment to save your work (lest a crash dissolve it into ether). Click the save button but then also close the form. When you go back to the Base interface we will find this document under “Forms”. Double click on it to use it.

Now the form appears again, but without the grid lines. This form is active and any data or alterations will end in the corresponding columns. You are actually in the position to

start populating the Psychiatrist table! If you click on either radio button, you will see the corresponding text appear in the text box. This means that this record has accepted your gender input! If you change your mind and click the other radio button, the form automatically changes the value and also erases the dot in the radio button previously selected, placing it in the new option. This happens because both radio buttons have “Gender” in their *Data field* and radio buttons are programed to be mutually exclusive.

Now, how do we make one of the radio buttons be selected by default, say, the “Female” option? For this to happen, you need to ungroup the radio buttons and call the Properties dialog box for the radio button with the female option. In the General tab you will find the *Default status* property. Check “Selected”. Immediately, a small black circle appears, indicating that this options is the current one.

As we said, we do not need the gender field box, and you should erase it. But before you do I want yo to study its Properties dialog box. Don't worry about the “Events” tab. Focus on the “General” and the “Data” tabs and study and play with the options. This teaches plenty.

Let's finish editing our Psychiatrist Form. By now you should have eliminated the “ID number” and “Gender” text boxes, made the labels bigger, re-size the remaining text boxes to be more commensurate with the amount of data they will typically receive and organize them in a way that is pleasant and easy to read. In the Toolbar select font size 26, font color white and center text. Then write: “Psychiatrist Data Entry Form” at the top of this page. Well, this is looking very nice.

At this moment you could produce the Medical Doctor Form because we are going to need it next and because it is very similar to the one we just made. This should be good practice.

Tab Stops (and Radio Buttons).

Most people that do data entry like to use the keyboard to navigate the form. If the cursor is in the first field, you can travel to the next by pressing the Tab key. They enter the value for the current field, press the Tab key and move to the next field. Try this.

I am sure that you noticed that every thing works fine except for the radio buttons. The navigation of the form just skips them. Currently a radio button can only be accessed by the Tab key when it has been selected. A group of radio buttons where none is selected cannot be accessed by the keyboard. I have found that even if you leave a radio button selected by default, that particular radio button is integrated into the tab order, but the others are left out, which does not help to change a selection. Also, the button is not necessarily left in a consecutive order, which is not elegant. This last thing, at least, we can fix!

Let's go back to Edit View: Close the active form and go back to the Base interface. Right click on the Psychiatrist's form and chose edit. The edit view of the form, grid pattern and all, should reappear now.

If you call the Properties dialog box of any ungrouped element you will see in the General section two properties: *Tabstop* and *Taborder*. The first one indicates if you want this element to be part of the tab order. For instance, we are using the Gender text box placed by the wizard as a window to check the value assigned to Gender. We don't need the tab to stop here so we could, with all confidence, set the Tabstop property of this element to "No". The *Taborder* indicates the sequence of tab stops in the form. You give each *taborder* property of every Properties dialog box of every element in your form a consecutive number starting with zero. Each press of the tab key will move the cursor in such order.

There is another way to do this that take less time: In the bottom of your form in edit view you can see several buttons. If you are not seeing this go to, and click on, View>Toolbars>Form design. You will see seventh from the left a button that shows a check box connected to a radio button with a curved arrow. If you place the mouse cursor over it a tool-tip will appear displaying the text: "Activation Order". Click it now and a dialog box will appear with a list of all the objects that can receive focus by pressing the Tab key. *Taborder* will follow the up-to-down order of this list. If you want to change the tab order all you need to do is reposition the elements in this list, either by dragging and dropping them or by selecting one and then moving it up or down with the arrow buttons.

Forms with Sub Forms.

Let's now develop the Patient form. This form imposes two new challenges: First, the need to record an unspecified amount of phone numbers for each patient and, second, the proper assignment of medical doctor and psychiatrist. If we do not enter the data for medical doctor or psychiatrist exactly the same way we have them in their own tables, Base could never find the records and thus would provide wrong information when queried. To prevent this we are going to use a drop-down list that will automate the correct input.

The problem with the phone numbers for the patients is that each patient will have a different and unknown number of phone numbers that need to be recorded. True to first normal form we had decided to place these phone numbers in their own table, using the patient's primary key as a foreign key in order to connect each phone number with its proper owner. The cardinality for this would be: one patient, one or more phone numbers, or 1..n.

Later, when we want to retrieve the phone numbers for a particular patient, we have to make a query that, more or less, reads: retrieve all phone numbers that have *this* primary key (belonging to a patient) as a foreign key. For this to work, we need to make sure to include the primary key of the current patient while recording his/her phone numbers.

In the draft we made for the patient form, we had devised a button that would call another form where we would record the phone numbers. This form would need to handle the recording of these relationships. The truth is that Base provides a solution that is by far more simple. This solution allows us to include a second form inside the primary form for tables that hold a 1..n relationship, where the sub form is for the table at the n side; and will automatically record the primary key of the current record at the principal form, in

this case, the current patient. This is what sub forms do. Obviously, we are going to use this option.

Let's start by calling the form wizard and select the patient table. Next, select all column names to receive input fields.

In the second step we are asked if we want to set up a sub form. This time click the checkbox for “Add Subform”. Then select the option that reads: “Subform based on existing relation”. Base offers you the tables that have a defined connection with the patient table (as established by the SQL code creating the tables that defined relationships). In this case you are offered to choose between *Payment* and *Phone number*. Select *Phone number* and then click on “Next”.

Now the wizard offers you all the column names in the Phone Number table for you to decide which ones will appear on the form. In this case there is *Phone ID, Patient ID, Number* and *Description. Phone ID* is the primary key of the record and *Patient ID* is the foreign key we need to associate for this to work. It would be a bad idea to include them as fields and risk someone altering the numbers. We already know that not including them will not stop Base form recording the data properly, so my recommendation here is that we don't include them. We will just select Number and Description, which is the data the user really needs to work with. After you have passed them to the box in the right, click on “Next”.

Because Base is going to calculate the joins for itself, the wizard is going to skip step four and land us directly on step five. Here we will select the layout for the form and the sub-form. This time we are going to select the *Columnar -Labels left* for the principal form and *Data Sheet* for the sub-form (options one and three from the left, respectively).

The reason for choosing Data Sheet is that it allows us to see a good number of the recorded phone numbers for the particular patient without having to navigate the records. Anyway, feel free to experiment.

Leave the default in step six (*The form is to display all data* -with no restrictions) and choose the style of your liking in step seven. In step eight chose a name for the form (I used Patient Data Entry) and ask the form to open it for editing.

(At this moment you could customize the form repeating the steps described before: Make the labels readable and adjust the size and layout of the text boxes. You can also take the field for the primary key out of tab order and make it to be read only, so no one inadvertently changes it (find the option in the Properties dialog box). Then, use font size 26 to write “Patient Data Entry Form” for a title and, hitting the <Enter> key several times, place the cursor above the sub-form and type “Phone Numbers:”).

To begin with let's arrange the sub-form. Left-double click on it. The green squares will appear, indicating that it has been selected and its Properties dialog box shows up. Analyze the General tab to have an idea of the options offered.

The first thing we can do is diminish the width of the sub-forms to make it less intimidating. You can do this by dragging one of the green squares on the sides, just like with the text boxes or the labels. Of course, changing the height will change the number of phone numbers in simultaneous display. The next thing that you can do is adjust the width of the columns for *Number* and *Description*. Remember that the phone number has been defined for a maximum of 10 characters and that if you include spaces while making an input and exceed the 10 number limit, Base will will reject it. Calculate a width accordingly so it gives the user a hint.

Close the design view and open the form. You will see that the sub-forms is not active at first. You can notice this because it does not offer you the chance to input a record. This is because the data in the sub-forms relates to data in the principal form but the principal form has no records yet. The principal form has the fields of First Name and Surname as required (i. e. NOT NULL. You can see this by reviewing the SQL code that created it). After you introduce the required data, the sub-form activates. You will see a green arrowhead indicating a yellow flash. You can now enter the number and its description, and navigate the form with the Tab key.

If you later decide to erase the record for the patient, Base will act depending on how you have handled deletions. If you had chosen Cascade Delete, for example, deleting the patient would delete all his/her phone numbers along with the record. If you had not made any provisions, Base will prompt an error dialog box instead. In that case you will need to manually erase the phone records and, only after that, erase the patient record. Give it a try. In order to manually delete a record press the icon with a red 'X' next to a green arrow head. If you haven't defined how to handle deletions yet, go to the “relationships” view (Tools>Relationships...) and find the line that connects the “Patient” table with the “Phone” table. Double click on it (or right-click and then select 'Edit') and a menu will appear. Select the “Delete cascade” and confirm.

Let's go back to design view and tackle our second challenge.

Drop Down lists

We have hinted that most times a unique number would be better than a surname as a primary key. When we associate a psychiatrist to a patient, what we are really doing is storing the primary key number -that identifies a psychiatrist- in the foreign key column of the patient table where psychiatrists go. In other words, relating the appropriate psychiatrist requires recording its primary key in the field called *Psychiatrist ID* in the Patient Data Entry form. If you check the SQL code that created the Patient table you will see that the data type assigned to *Psychiatrist ID* is an INTEGER, which handles numbers. If you try to write “Dr. Jones” you will get an error. The form expects a value like “012”, which should be the primary key for the record of Dr. Jones.

This means that we would need to keep track of the primary keys associated to each psychiatrist recorded in order to enter the right information. Who remembers this? This is not

practical. What we really want is to have Base offer us the names of the psychiatrists but, once we have made our selection, really record only the key number. This is exactly what a drop down list can do.

At this point you have the Patient Data Entry Form in Design View. Select the Psychiatrist ID label and text box and un-group them. Now select the text box and delete it. Call the Form Controls menu and select the List Box icon, the one that looks like a rectangle with lines of text and *up* and *down* arrowheads to the right side. This icon is usually next to the radio button icon.

Place a List box to the right of the label. When you release the mouse button, the **List box wizard** appears. This wizard goes through three steps in order to identify the list box's source and destination values.

In the first step you are shown a list of all the tables in this database and are asked to select the source table, that is, the table from which the list box will gather its data. In this case you will need to choose the Psychiatrist's table. Then click next.

In the second step you are shown a list of the names of all the columns in the Psychiatrist's table (the chosen table) and are asked to indicate which one will provide the data to appear in the list box. In this case, chose the *Surname* attribute. This means that, when active, this list box will display all the *Surname* records found in the *Psychiatrist* table.

In the third step you indicate what data from the source table (the *Psychiatrist* table in this case) is going to what column in the destination table (the *Patient* table in this case). Please note that you are entitled to chose *any* column from the source table and any column from the destination table. They are all listed in their respective boxes. There is only one condition: the data type of the source must match the data type of the destination. This makes sense: it would be impossible to try to insert a VARVCHAR type of data into a TINYINT.

In this third step, the wizard shows you two boxes. Inside the boxes the wizard lists all the names of the columns in the corresponding tables. The box to the left, under the caption “Field from the Value Table” corresponds to the destination table (in this case, the *Patient* Table). When you chose a column here, this will be the receiving column. In this case you should select *Psychiatrist ID*.

The box to the right, under the caption “Field from the List Table” corresponds to the source table (in this case, the *Psychiatrist* table). The name you chose here will be the name of the column from which the values will be copied. In this case you should select *ID Number*.

After you have chosen a destination and a source, click on finish. This will close the wizard and leave you to edit your List box. Arrange position and size to your liking.

Call the List box's Property dialog box and study the attributes given to it by the wizard. The more important ones are in the *Data* tab. Particularly, *Type of lists contents* specifies the use of SQL and *List content* holds the SQL instruction to be executed[6].

For now, let's go and see how this table works. Close design view and call the psychiatrist form. Make sure to include some valid entries. Then, close this form and call the patient form.

When you click on the arrow head you will see the surnames of all the psychiatrists in your database. The beauty of this is that this list is prepared, by the SQL instruction that we saw, every time we use this control. This means that if you include more psychiatrists in your database, their names will also be included in the drop-down list. This list updates itself!

When you select a psychiatrist, it is the primary key number, not the surname, that is passed to the patient table. You can monitor this by creating a text box and associating it to the corresponding field. You will see it displaying the appropriate number.

However, you could express concern because the list box is displaying surnames only. With two psychiatrist with the same surname, which is the one I want to select? In order to decide, it could be good if the list box included the first name of the psychiatrist too. How do we do this?

Here is where we will need to use SQL again!

List Boxes with compound fields.

By now you know that the List Box wizard will ask you to chose a table and the name of one, and only one, column to populate the data. We have chosen “Surname” but now feel that it is not enough. We would like the List Box to display both, the surname AND the first name, in order to make the selection clearer to the user.

The thing is, how do we produce a table -or something similar- that will have the first name and the surname of the psychiatrist table as one column?

This is where a *View* comes in handy. A View is the result of a query. As you know, the results of queries are displayed in columns and, consequently, can be thought of as tables. In fact, Base stores Views with the tables. So we need to make a query that will produce the result we want, transform that query into a view and then use the resulting view with the List Box wizard.

Queries are built with SQL code. Although Base allows us to create simple queries with the ease of point and click, this particular query uses a syntax available to the embedded

6 Which in this case should read: `SELECT "Surname", "ID Number" FROM "Psychiatrist"`. You can also do the work of the wizard by hand as explained in pages 108-109. But for now, let the wizard take care of this.

HSQL database engine but not incorporated into versions of Base earlier than 3.2, so we are going to produce it manually. Let's start.

First, in the Base interface, select the icon over the name “Queries” This is the second icon from top to bottom in the column under the name “Database”. Three options will appear in the top panel under the name “Tasks”: “Create Query in Design View...”, “Use Wizard to Create Query” and “Create Query in SQL view”. Select this last one.

A screen will open with a blinking cursor. Here is where you will enter the SQL code. Type the following:

```
SELECT "Surname" || ', ' || "First Name" AS "Name", "ID Number" FROM
"Psychiatrist";
```

Actually, this code is quite simple to understand except for those “||” signs, called 'double pipes', between *surname* and *first name*. Let's analyze this statement:

This instruction commands the database to **select** *Surname*, *First Name* and *Number ID* from the *Psychiatrist* table. The double pipe is a concatenation instruction accepted by HSQL (but, as said, not understood by recent earlier versions of Base). In this case, a double pipe is connecting the contents of Surname with a comma and a space and a second double pipe is connecting the previous with the contents of First Name, and all of these is to be placed inside one column called “Name”. This is just what we needed.

Remember that with Base, single quotes are used for string literals, in this case, the characters for the comma and the space, and that double quotes are used for the names of tables, columns or constraints (unless they only use capitals and no spaces in those names, in which case the quotes are not needed).

If you run this query you will see a table with two columns: The first one would have the surname and first names of all the psychiatrists in your table, separated by a comma (and a space, very important), and the second column would have the primary key number.

The editor in which you have typed this query has two buttons for running it: One that looks like two pages with a green tick where they overlap (whose tool-tip reads: “Run query”) and another one that says SQL with a green tick over the letters (that displays “Run SQL command directly” when you place the cursor over it). The first button runs the instruction by Base first but if your version of Base does not understand the double pipes, it will throw an error message of improper syntax. In such case you must select the second button, which runs the SQL instruction directly. Press it and you will not see any changes in your screen except that the button remains depressed and the other run query button becomes deactivated. Now save this query. I used the name “qryPsychiatrist” to remember that this is a query.

You can double click on the query and a table will come up. In my computer, and after previously inserting three fictional psychiatrists through the Psychiatrist Data Entry form, I have:

Name	ID Number
Smith, John	2
Perez, Adelaide	3
Lecter, Hannibal	5

My ID Numbers are not consecutive because I had made some deletions and HSQL does not recuperate numbers, it just keeps incrementing its counter.

Another important thing to know is that if any of the fields that were joined by the double pipe query command, in this case surname and first name, were empty, then the entire record in the “Name” column would appear empty too. That is, If I had not entered the first name for Dr. Perez, for example, my little table would look like this:

Name	ID Number
Smith, John	2
	3
Lecter, Hannibal	5

We have two options here: we include conditional clauses in the SQL command, looking for empty strings and instructing to include the non empty element, or we make sure that both, Surname and First Name are defined as NOT NULL so that they can not be empty when I need to use them. I will chose the second solution for simplicity. Let's go back to producing our compound list box.

For the next step we cannot use the query. We need the table produced by the query, which is called a View. You can think of a View as a virtual table. It is generated by the query when it's called, at which moment it becomes a table you can use. The good news is that this virtual table will update every time it's called, incorporating new records or deletions to the psychiatrist table. The bad news is that this calculation will add some time to the production of the form, making it slower the more records it needs to process.

To produce a View, right click the query we just created and, in the context menu that appears, select the option “Create as View”. A little box will ask you for a name for this view. I used “vPsyList”. After you confirm this, the box disappears and not much more happens. If you now go to the Tables section, you will see our new addition, with a particular icon showing that it is not a regular table but a view.

Now you can produce your list box as normally. Go to the Form section and select the “Patient Data Entry” form in edit mode. Create a new list box. When the List Box wizard appears, it will include the “vPsyList” view along with your tables. Select it. Choose “Name” to populate the list box and chose “Number ID” as the record to be copied to “Psychiatrist ID” in the receiving table. TADAAAH!

Now, create a list box for the medical doctors that will also display both the surname and the first name. You will need to adapt the SQL code accordingly.

Default values.

The address component of the tables we have used in this tutorial all include a “State” attribute. One can expect that most of the patients for this psychotherapy clinic will reside in the same state. It is possible to have the forms display this state automatically and simplify the process of data entry for the user. Of course, if the user encounters a patient that actually lives in another state and comes in for therapy, then he can change the value in the form.

Default values can be set by the Properties dialog box. Select the text box for the “State” field. Find the property “Default Value” and enter the string that you want. For this example I will use “NY”, which is where I live at the time of writing this tutorial. Now, the forms will include the “NY” value in the “State” field for each new record.

Default values can also be made possible by editing the tables. The big difference is that default values through the Properties box only affect the fields in the form. This means that if you decide to include records writing directly to the tables, there will be no default values in the assigned fields. However, if you assign default values through the design of the table, these values will appear in both, the forms and while using the tables to enter data. Let's see how to do this:

First, click on the Tables icon and right click on the “Patient” table. In the context menu that appears, click on “Edit”. The table now opens in Design Mode. You can see a list of all column names and the data types each accept. In the lower part you can set some parameters for the variable, which are specific to each data type and will change when you change the kind of variable you select.

Find the “State” column name and select it. The bottom of the page shows the parameters specific to Text (Char) data types. The third parameter from top to bottom is *Default value*. Here we will type “NY”. Now save the table and we are ready. If you open the form you will see that the next record already has NY in the field for State. Of course, if the user needs to change the value to “NJ”, for example, all he needs to do is overwrite on the field.

Very nice. Let's now see how to do the same with DATE variables.

Entering time and date.

The patient table has two columns that store date information: the DATE for registering the birthday of the patient and a TIMESTAMP for registering the moment she/he is admitted as a patient. You might remember that DATE records year, month and day information while TIMESTAMP records year, month, day, hour, minute and second. To expand

this example let me also reference the “Schedule” table, which is where we record the next session for a patient. This table records both DATE and TIME information. TIME records hour, minute and second.

Our goal here is to either set some date information automatically or help the user minimize error at entry.

Let's work the first approach. You tell Base to automatically set date or time information when you are building your table (with SQL code): First you create the type of date variable you need and then you specify that, as a default, it should receive the current date. HSQL has the following built-in functions for doing this:

CURRENT_TIME fetches the time at the moment this function is being used,
CURRENT_DATE fetches the date and
CURRENT_TIMESTAMP fetches both date and time information.

So your code would look something like this:

```
"Moment of registry" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

In fact, if you analyze the SQL code that created the tables for this tutorial you will find some columns for date information that use this code.

Let's analyze this command: The string inside the double quotes is the name of the column. Then we assign a TIMESTAMP data type to it. Later, we specify that it will take a default value with the instruction “DEFAULT” and finally we call the “CURRENT_TIMESTAMP” function, which fetches date and time information from the internal clock of the computer. The final comma assumes that this is not the last instruction in the creation of the table. If it were, just omit it.

When you run the form wizard with a table that includes **TIMESTAMP,** it produces what seems like a box with a line inside separating it into two fields. The truth is that these are a time box and a date box which have been grouped.

While the user is filling the form, the **TIMESTAMP** box does not show any particular behavior. But when you come back to a record you have entered, you will see that the **TIMESTAMP** box displays date and time information. In theory, if you change any information in a form, this change should modify the timestamp to reflect the time that the record has been updated. In practice I have seen that the **TIMESTAMP** box does not change if I modify the records and achieving this would require some Java programming, which is beyond the scope of this tutorial. The user can do it manually, however.

Now read and interpret the following:

```
"Time of registry" TIME DEFAULT CURRENT_TIME,
"Day of registry" DATE DEFAULT CURRENT_DATE,
```

These instructions would create columns that would automatically insert time and date information, respectively, when a record is created. In my experience, these instruction also do not update if you modify a record and the user will need to manually adjust them, if required.

However, there are some times when you do not want the date and/or time records modified. This is particularly true when your database stores historical information. For example, at what moment was one patient assigned to a therapist is such kind of historical information. When the patient was transferred to another therapist or when he stopped being a client of the clinic are all historical information which you would want to preserve, for administrative reasons. Not having them modified automatically would be a good thing. What you need to do now is prevent the user from modifying them manually.

In order to accomplish this you have several options. First, you could set Date or Timestamp by default and not display the corresponding field in the form. Remember that this automatic assignment is accomplished through the instructions that created the table, and not displaying the field does not stop the process, it just prevents the user from modifying the data. Later, when you are producing a report, you can retrieve this information. Or you could display the field anyway, but change its property to “Read only”, and prevent anyone from changing the data.

To try this, build the form for the “Assignment” table. (While you are at this, make the labels bigger and easier to read, eliminate the “Assignment” primary key field, set the therapist and patient fields to drop-down lists that display both name and surname, provide some useful tool-tips and give this form a descriptive title at the head of it). Now, in the Properties dialog box set the “date assigned” text box to read only. With this form you can assign patients to a therapist with simple point and click and let Base automatically record the time of the assignment. This is really cool!

Date and Time boxes are customizable, allowing the display of widgets that aid the input of data . Lets see.

In the “Assignment” form, ungroup the Date box from it's label and call the Properties dialog box of the date box. Study the “General” tab to see the goodies that it offers. Find the *Spin* and *Repeat* properties (they are right next to each other) and set them to “yes”. Further down you will find a *Dropdown* property. Also set it to yes. By now you will discover that the date box has changed: it offers two buttons, one on top of the other, with arrows in opposing directions. It also offers a bigger box with an arrow head pointing down. Save and run the form.

If you now click on the big arrow head you will see a small calendar for the month appear. You can swiftly change months by clicking on the arrows to the sides of the title where the name of the current month is in display. You can also select any day within the month just by clicking on it. At the bottom you will find two buttons: One for selecting the current day and one for erasing any date selected previously. Feel free to play with this.

The nice thing about this small calendar is that the user can enter a date in the proximity of the current date checking with a simple glimpse if it falls on a weekend, in this week or the next, etcetera, minimizing errors while entering date information.

Some other circumstances in date entry could not need this widget: Imagine that you need to enter the birth day of a person who was born the 14th of December in 1936. How many times do I Have to click on the left arrowhead to reach such date? Well, about 12 times for each year separating us from 1936. For sure, just typing 1936/12/14 (or in the corresponding format) seems lightning fast by comparison. However, I could not think of a better tool for entering the “next session scheduled” information. So, again, the appropriateness of this tool depends on the context. Remember that simplicity and ease of use are our primary goals.

The smaller buttons with opposing arrow heads increment or decrement the unit of time selected. If you just press one of the buttons now, the cursor will appear in the first domain to the left, which holds the month information in the USA, and will increment or decrement it accordingly. If you move your cursor to the day or year section, the arrows will increment or decrement them, respectively.

You can do something similar for Time boxes, which you can try in the form for the “Schedule” table. If you set the *Spin* and *Repeat* properties to yes, the box will show the two opposing arrowheads, with which you can increment the time displayed by the box. Time boxes do not have a *Dropdown* property.

As said before, the TIMESTAMP box is really a grouped Date and Time boxes. If you ungroup them and call their respective Properties dialog boxes, you can activate these widgets.

To finish this exercise, set the drop-down calendar as a property of the "Date case closed" date box in the Assignment” form.

Forms with two or more sub forms.

We are now going to tackle a special but not uncommon case of forms: Where you need to link three or more tables. This usually happens when we confront intermediate tables that help with many to many relationships. Think about this: the intermediate table usually only holds combinations of primary keys from two different tables, sometimes some extra information. But these combinations make sense only when we can relate them to the tables they derive from. We need to handle information from -or for- three tables in one form.

Our example records the medication some patients are taking. The *Patient* table provides the data of a particular patient. The *Medication* table provides information about the medication. The intermediate *PatientMedication* table records one or more medications for each patient that is taking medication. The thing is: how can we have all this information in the same form? By now we know about sub-forms and the help the wizard provides in

setting them up. But the wizard only adds one sub-form per form, which allows us to enter data for a second table only. We need to add a third sub-form and we need to do it manually. This is what we will study next.

As we know, the forms that we have built with the wizard are really just writer (.odt) documents over which the wizard has added a form. The wizard has been kind enough to connect the form to a particular table in our database. In this sense, the .odt document is just like a canvas and we can add more forms to it. We usually record information from one table to another in the form through the List Box control. The wizard is also kind enough to make the proper connections by asking us which cells we want to connect. It will now be left to us to place the forms, connect them to the corresponding tables and later interconnect them using one or more list boxes.

Let's establish some generalities that can help us with this: Each form is associated with one table. In the case of forms with sub-forms, we have two forms in one .odt document: The original or principal form and the new or sub-form. The table for the principal form establishes a 1..n relationship with the table for the sub-form. While the principal form displays a particular record, the sub-forms displays all the records in the associated table that belong to the record in the principal form. So the principal form dictates what records the sub-form must display. This is why the sub-form is called “sub-form” or “Slave form”. The principal form is called the “Parent form” or the “Master form”.

The process of creating a .odt document with several forms is basically like this: We insert two or more forms in the writer document. Each form will be connected to a particular table. Then we will need to indicate which form will be the master form (or forms) and which will be the slave form (or forms). Each form will receive controls that display and allow the recording of information. Each control is usually connected to one particular field (column) except for list boxes that will connect two.

To follow the implementation we will do now, be sure to review the SQL code that created the *Patient*, *Medication* and *PatientMedication* tables. Pay attention to the variable data types of the columns. Then, make sure you study the way they interconnect by reviewing the UML diagram. Particularly, notice that the Patient table has a 1..n cardinality with the intermediate table and that the Medication table also has a 1..n relationship with it. The intermediate table will want to record the primary key of the patient and the primary key of the medication. This table also records data about the dosage assigned and the date such medication/dosage was started and ended. Because each patient could have more than one medication, it would be better to use a sub-form in the tabular format so we can read several entries at once, very much like the sub-form for telephones for the patient form. Finally, notice that the *Medication* table stores a description of what the medication does, info that would be useful to display in this form.

While following the coming instructions, take some time to study the properties dialog boxes for forms and controls, so you can readily identify the properties that we need to set. To learn more about them, don't forget to study the help files that describe them.

Let's start by using the form design wizard to create a new form. Select the Patient table but then only select the First Name and Surname columns. That's all the information we will need to display from this table. In the second step do not chose “Add a Subform” because we are going to create it manually. Finish the process by allowing the editing of all data, pick colors of your liking and open in edit mode.

By now we have about one third of our form. For the next two thirds our new best friend will be the **Form Navigator**. Don't confuse the Form Navigator with the Navigator. The Navigator is called by clicking on the button that resembles a compass located in the toolbar. The Form Navigator, by contrast, is located in the Form Design toolbar, which is usually found at the bottom of the form you are editing. The button for the Form Navigator looks like a compass over a form and is usually the fifth button from the left. If you can't see the Form Design toolbar, click on View> Toolbars> Form Design.

The Form Navigator displays all the forms associated with the current .odt document. If we examine the contents of the Form Navigator we will see a folder with the name “Forms” and then a folder with the name “Main Form”. This “Main Form” is associated with a label called “lblFirstName”, a text box called “txtFirstName”, a second label called “lblSurname” and a second box called “txtSurname”. You can see that this describes exactly our current form.

If you click once on “Main Form” you will see green handles surround the labels and text boxes in our form. The rest is just the writer (.odt) document.

Obviously, the content of this Form Navigator was written by the wizard after accepting our selections. Now, we are going to write directly to it.

Let's start by creating a sub-form. In the Form Navigator, right-click on “Main Form” and from the context menu that appears chose New>Form. Now a new form has appeared, with the name “Form” or “Standard” that is as a dependency of (that is, with a line connected to) “Main Form”. You can actually drag this form and place it as an independent form, appearing connected to “Forms” instead. Or you could have created a new form (with controls and all) and only later drag it to become dependent of “Main Form”. It works either way. By creating this dependency, you establish that the form called “Form” or “Standard” is a sub-form of the form called “Main Form”

Let's review the properties of this form. Right click on “Standard” and select “Properties”. Now, the Properties dialog box for this new form appears. You can see that a Form has three tabs: *General*, *Data* and *Events*. This last tab is reserved for Macro programming, so we will leave it unattended. The Data tab allows us to make all the connections that we need, so we will be spending some time here. The General tab is quite small and we won't need much of its functionality. However, you can change the name of the Form here. “Standard” is not very descriptive so change it now to “PatMed form”.

We have created a new form but this form is invisible to the user because it has no controls yet. Let's add the control “Table Control” which has the appearance of a grid. You

can find a panel to your left with the Form Control elements. If you don't see it, go to View>Toolbars>Form Controls. Click on the box for: *More Controls*[7] and then select "Table Control". Now find a good position in your form and place it by dragging and releasing the mouse button. At this moment a **Table Element** Wizard appears, requesting that we connect this table to a data source. We could do this trough the *Data* tab in the Properties box (and we can change selections there later) but doing it here is fine too. First, we must select a table. In this case we need to connect to the *PatientMedication* table. Select it and then click on next. Now we are requested to chose the columns that will be displayed by this Table Control. We don't need the Patient ID. Just select Medication ID, Start Date, Dosage and End Date. If you now click on finish, you will see the table with column names for the data requested. You can also inspect the Form Navigator. If things have gone as expected, you should see the Table control associated to our new sub-form.

Now pay attention to this as things can get a bit confusing: The Table Control has a set of properties. If the Properties dialog box is open, you can see them by selecting the Table Control. Each of the fields inside the table control (the columns) also have properties that are unique to them. You can see them in the Properties dialog box when you click on the header of the columns. Finally, the form itself also has its own properties, which you can call by clicking on the name of the form in the Form Navigator. Do practice now calling for these properties and study their options until you become familiar with them. It feels terrible and is quite easy to be looking for properties that belong to the Form when you have really selected the Table Control, for example. So make sure that you can access them at will and can tell them apart.

Now is time to make our assignments. We have wanted this new form to be a sub-form of the form with the patient data. That is why we have placed it as a dependency of it. Because of this, Base assigns special properties to this form that allow us to link the corresponding primary and foreign keys so that the sub-form displays appropriate data. Let's see this: Click on the sub-form name (that is "PatMed form") in the Form Navigator and call its properties dialog box. Select the *Data* tab.

You can read that the first option is "Content Type" and that it's assigned to Table. If you peek on the options you will discover that we can also chose a query or a SQL command. Table is selected here as a consequence of our earlier interaction with the wizard. For the same reason the *PatientMedication* table is selected in the second option: Content.

Further down we have the options: "Link master fields" and "Link slave fields". The *Link master fields* option selects the data field of the table associated to the parent form that is responsible for the synchronization between parent and sub-form. This is usually the primary key and in our example it's the "ID Number" field from the *Patient* table. The slave field is the field of the table associated to the sub-form that holds the foreign key. In this example it's the "Patient ID" field from the *PatientMedication* table. You can write down these column names or you can click on "..." to the right. A dialog box will appear that allows you to find and select both fields simultaneously. Notice that the field for the

7 Read the tool-tips. This box is usually number 3rd from the right. The button for 'Table Control' looks like a table.

sub-form is located to the right in the dialog box, under the name of the table it derives from; and that the field for the master from is offered to the left. Select them now.

How does the wizard know from which tables to offer options to link master and slave fields? Because we have them connected as shown by the Form Navigator. By the time you click OK, confirming your selections, “Main form” and “PatMed form” become form and sub-form, respectively. This means that when you select one particular patient, the sub-form “PatMed form” will show you any medications associated solely with the particular patient selected in “Main form”.

Let's now go to the third leg of this exercise: Including the “Medication” Table. What we want to achieve is that when we select a medication in the “PatMed form” we can read some information about that particular medication. This means that the “PatMed form” will be the master in relation to a third form we will have to include although it is the slave form in relation to “Main form”.

Right click on “PatMed form” in the form navigator. In the context menu that appears select New... and then click on “Form”. A new form appears in the form navigator, with the name “Standard” or “Form”. Call its properties dialog box. In the *General* tab, change the name to “Medication info”. Now go to the *Data* tab. For “Content type” select “Table” and for “Content” select the table *Medication*. Now skip to the part where we link master and slave fields and call the wizard (the “...” button). From the “Medication” option select “Medication ID”. Remember, this is going to be the slave field. From “PatientMedication” select “Medication ID”. This is going to be the master field. Don't get confused by the fact that both columns are called “Medication ID” and instead notice that what we are doing is joining foreign key with primary key. You might notice that it is the intermediate table, which provides the foreign key, the one linked to the master form and that the *Medication* table, which provides a primary key, is associated to the slave form. This is perfectly fine: It is not the primary key which defines who gets to be the master form but our need that the “Medication” table provide information selected in the “PatMed form”.

Our third form has no controls yet. Select it now and add a label and a text box. Make them big so that they balance the big Table control if you place them side by side like I did. In the properties dialog box name the label “Med Descrip” or something like this and make it display “Medication Description”. Also give it a background color that will make it stand out (I chose Grey 20%). In the *Data* tab of the Text box select “Description” for Data field. This way, every time you pick a medication in the PatMed form, a description of the chosen medication will be displayed in this box.

Our last task will be to have a list box display medication names (but really record primary key numbers in the “PatMed form”) so that we can assign medications to the chosen patient.

Now for a big surprise: The Table Control is not really a control but a container of controls that are displayed in tabular form. If you select a column header you will see, upon calling the properties dialog box, that each column holds a control relevant to the column

data type defined by the SQL code that created the table associated to the form. So, for example, if you select the “Dosage” column in the Table control, the header of the properties box calls it a text box, “End Date” is described as Date field and so on. If you examine their respective *Data* tabs, you will see that they were appropriately assigned by the wizard to the columns. Consequently, “Medication ID” corresponds to an Integer. We don't need this and we will change it to a List Box.

Let's first review what is it that we want to achieve: We want to record the primary key of the medication that is being assigned to the patient in display. Of course, we don't want to memorize which primary key belongs to which medication. Instead, we want the List box to display the names for each medication and assign the primary key of the one we click on. This also relives us from needing to type the name of the medication and making a mistake that can come and haunt us later. The selection we make will be recorded in the “Medication ID” field of the *PatientMedication* table, which we have associated with the “PatMed form”.

Let's get started: First, right click on the “Medication” column header in the Table control and select “Replace with...” and then chose “List Box”. With this column still selected, call the properties dialog box. In the *General* tab, change the Label to “Medication”. Now call the *Data* tab.

The *Data* tab of the List box has four properties relevant to the task at hand. These are: Data field, Type of list contents, List content and Bound field.

The **Data field** specifies the receiving cell of the table associated with the form. In this case, this would be “Medication ID” of *PatientMedication*. Because this control has already been assigned to such table, this property option has a drop down list that offers you all the available fields. Select it now.

Type of list contents is asking how the items to appear in the List box (the names of the medications) are to be generated. We are going to use some SQL code for this so you can select SQL. The SQL code that we are going to use is no more complex than the examples we have used so far. In any event, the next section explores the SELECT command in depth, if you want to have that information as background.

List content is going to be generated by the following type of SQL statement:

SELECT <field 0>, <field 1> FROM <selection table>;

This instruction displays all the records in “field 0” and “field 1” of the table “selection table”, which is the table from which we are going to make our selections. In our example, it must look like this:

```
SELECT "Name", "Medication ID" FROM "Medication";
```

After we have entered it, Base is going to enclose the entire instruction in its own set of double quotes. Don't be alarmed by this.

Note the order in which the SELECT statement was designed: first we call for “Name” (field 0) and only secondly do we call for the primary key “Memeber ID” (field 1). This is relevant for our last property:

Bound field specifies how to bind the fields for the list box. When you select the option 1, field 0 will provide data to be displayed in the drop down list while the field 1 will provide the data to be entered in the field specified in Data field. In our example, then, field 0 (“Name” from the table *Medication*) will populate the drop down list while field 1 (“Medication ID” from the same table *Medication*) will be entered in “Medication ID” of the table associated with this form (*PatientMedication).*

OK, we are ready! Let's test our form. First, populate the patient table with three or more fictitious patients. Then populate the medication table. You can use the following:

Name	Description
Librax	Treatment of peptic ulcer and irritable colon.
Loprox	Topical fungal skin conditions.
Maxaquin	Quinolone antibiotic for lower respiratory infections.

Now call our form. You should see that the “Medication” column has a box with an arrow head pointing down. If you click on it you should see the medications previously entered for you to select from. After doing so, a description of the medication will appear under “Medication Description”.

Because we have set “Start Date” to be populated automatically, we should see the date displayed when we return to an entered record. “End Date” is supposed to be populated by us. To make this easier for the user, you can open the form in edit mode and call the properties dialog box for this column. Then assign the “Drop down” property to yes. This will bring out the calendar widget. While you are here, don't forget to give this form an appropriate title. Write a header with: “Patient Medication Assignment and History” or something like this.

Chapter 9

Producing Queries.

Now we are ready to really take advantage of all the hard work we have done so far. Producing and reviewing queries can be truly exhilarating because this is the moment that all the *data* you have stored becomes *information* you can use.

Think about this: If you were the director of the psychological clinic we have been using for our example, wouldn't you find it informative to know that 74.6 % of all patients with a form of depression are female, that 12.7% of all the psychiatrists that handle the medication for your patients are in charge of actually 77.6% of all the patients that receive medication or that 26.8% of patients at your clinic have been diagnosed a form of post-traumatic stress disorder? Well, if you ask your database appropriately, this is the kind of information you can extract from it.

So, now you know that databases are much more than just a way to store names and addresses and later print labels for mailing. This kind of information can help you distinguish trends, calculate projections and help you make decisions.

For all this wonder to happen, you need to make sure that the information that you extract is valid information. This in turn rests on *Data Integrity*, the idea that you have stored the right information in the right place and that your system has not changed it for some obscure reason. You have been working on data integrity from the very beginning of this tutorial: The careful design of your tables and relationships -including column attributes and the handling of deletions, compliance to first, second and third normal form, making sure that the input of data minimizes entry errors... all this is aimed to ensure data integrity.

Now, the trick is to learn how to query your database in order to get this information out. If you have read other tutorials that describe reports and queries with Base, you know that a very friendly graphic user interface (GUI) has been developed to help you query your tables. In this tutorial, however, we are going to rely on the use of SQL code to create queries. First, because complex queries are really easier when formulated with SQL code (and sometimes, it's the only way to go). Second, because learning how to make SQL queries is not difficult and, third, because if you learn how to query with SQL, the GUI will be really easy to use.

Let's start with some generalities: A query always produces a table, that is, the result of your query will always be delivered as columns and rows. Even if it only has one column, it will still be a table. Reports are a fancier rendition of a query: You can include the logo of your company, include the day (date) or the person who designed the query and even organize the layout of your information. However, reports are based on queries. You should know that the Report Designer in Base can only use one table at the time as source

for creating your report. This means that we have to create a query that integrates the information that we need into one table before we can give it a fancy look with a Report. We have already done something similar when creating compound list boxes for data entry.

Are you ready? Let's get started.

SQL queries with Base.

You might remember that Base offers two places to enter SQL instructions. This time we are going to use the Query section. Click on the icon over the text “Queries” in your Base interface, located in the column to the left under the heading “Database”. Three options will appear in the *Tasks* panel. Select the last one that reads: *Create Query in SQL view...*

The workhorse for creating queries is the SELECT command. This command is very versatile and powerful and will be the focus of attention for the rest of this section.

In its most basic form, the SELECT command needs only one parameter: The table from which you want to extract your information, like this:

```
SELECT * FROM "Psychiatrist";
```
[8]

Let's analyze this statement:

First, the statement ends with a semicolon, which is true for all statements in SQL and, therefore, for HSQL. We saw this when we analyzed the code for creating tables earlier in this chapter[9].

The '*' sign is a *wild card*, that is, it replaces the actual names of the columns in your table and represents them all. If you were to read this statement aloud, you should say something like: 'Select “all columns” from [the table:] Psychiatrist'.

This would produce the entire “Psychiatrist” table. Let's see if it is true. Type the statement and then click on the icon that has two papers and a green check where they intersect, which is the “Run Query” button. If you made no typos, then the “Psychiatrist” table should appear.

You see, this is all. Not that difficult. What we want to accomplish now is to be able to reduce the data displayed, by filtering it, so that we can extract information that is not immediately apparent.

---

8 Look at this code: `SELECT * from “Psychiatrist”;`. Notice that the double quotation marks required by the name of the table are slanted. If you introduce code like this to the query window Base will throw a syntax error. The double quotation marks required by the SQL code must be straight. It seems like a small thing but they are different characters and knowing this will spare you a big headache.

9 However, HSQL is quite forgiving if you don't include it. This might not be true with other database systems.

Let's say that you want your assistant to contact all the psychiatrists by phone. She could review record by record and write down their phone numbers or you could make her life easier and swiftly produce a printed list for her. To do this, instead of selecting all columns in the table, you just want the name and phone number. Then, we should try this:

```
SELECT "First Name", "Surname", "Phone Number" FROM "Psychiatrist";
```

Notice that I am indicating the names of the three columns that I want, inside double quotes and separated by commas, and the table from which I want them extracted. Run the query. Ah! This list is easier for the assistant to use.

Remember that if I had not used different case sizes and blank spaces for the names of columns and tables we would not need to use double quotation marks.

To make this list clearer to your assistant, we would like to make explicit that these are the psychiatrists and not patients or the medical doctors. For this we can use an alias. You can see that every first row in the resulting table is appropriately labeled with the name of the columns they represent: “First Name”, “Surname” etc. But we can change this and request that the column be referenced with another name. We do this with the key word “AS”, like so:

```
SELECT "First Name" AS "Psychiatrist",
       "Surname",
       "Phone Number"
FROM "Psychiatrist";
```

Note that the command has been fragmented. We have done this just to make it easier for humans to read. The SQL window will not pay attention to this and still understand, and execute, the instruction as if it were one long line. We should take advantage of this because it makes finding mistakes easier when we are composing complex instructions.

If you run this query now you will have almost the same output than before, except that the column with the first names will have the heading “Psychiatrist”. The alias command not only changes the label in the column, it also allows us to reference this column by the new name given. We will do this later.

If you are starting to get the hang of this you could be asking: Hey, can we concatenate the name and surname into one column like we did for the list boxes? That would really make the resulting list a very cool one! And, of course, we can. You could just copy the instruction we used then, with the concatenation pipes and the strings within simple quotes. Just remember not to run the command by Base if your version of Base is previous to 3.2 and use instead the “Run SQL command directly...” button (the one that reads SQL with a green tick on top).

But instead of going this way I want to introduce the use of functions.

Built-in Functions in HSQL.

Functions are statements in HSQL code that return a value. Most of these are understood and used by Base. We already know some:

CURRENT_TIME returns the time in the clock of your computer at the moment the function is called.

Other functions take one or more parameters. Think of the COS(d) function: you give it the parameter 'd' (which is supposed to be an angle) and it will return the cosine of the angle. Let's imagine that you have a database that has collected some measured angles and you need their cosine. This would be as easy as:

```
SELECT "Angle Data" AS "Angle α",
       COS("Angle Data") AS "Cos α"
FROM "Measurements";
```

Note that we have given the name of a column as a parameter. Again, because of capitalization and spaces, this name is within double quotes. This instruction produces a result table with two columns: The angles you stored in the column “Angle Data” in the “Measurements” table, and the cosine of those very same angles, and all clearly labeled for ease of use.

There is a great deal of interesting functions available for HSQL, and you can find them in their website:

http://hsqldb.org/web/hsqlDocsFrame.html

If you check their documentation, you will find a function like this:

CONCAT (string1, string2)

This one would allow us to concatenate name and surname, like this:

```
SELECT CONCAT("Surname", "First Name") AS "Psychiatrist",
```

Let's replace our first line in the previous example with this instruction and run the query. What happens? This should really make your assistant very happy!

```
SELECT CONCAT("Surname", "First Name") AS "Psychiatrist",
       "Phone Number"
FROM "Psychiatrist";
```

You will see that there is no space or comma between name and surname in the resulting set: They have just been joined. In this respect, the method using the pipes is better. It also happens that this function accepts only two parameters: string1 and string2, while the pipes allow you to connect as many clauses as you want. For instance, we could be adding name, middle name and surname. We can't do that with CONCAT directly.

However, there is a workaround to solve this problem that also helps us understand the versatility of functions and how we can use them. Let's think about the end result that we want: We want the column that displays the names of the psychiatrists to have the following format: Surname + comma + space + first name. Basically this means that we want to join the surname with a string containing a comma and a space (remember that with HSQL strings are enclosed within single quotes) and then join this result with the first name. We can represent this idea like this:

`CONCAT(α, "First Name")` where

`α` means: **`CONCAT("Surname", ', ')`** so our statement should look like this:

```
CONCAT(CONCAT("Surname", ', '), "First Name")
```

This could be somewhat difficult to read, with so many quote signs and with commas that are both part of the end result and also part of the definition of the function. But after you analyze it for a while it makes perfect sense. It also shows that HSQL will not complain about applying any combination of functions as long as we don't default the syntax and the data type. This allows for a very powerful use of functions.

Can we use the CONCAT function instead of the double pipes in building compound list boxes? Yes, which will also allow us to run the instruction through Base and not skipping directly to HSQL.

This makes for a nice introduction to the versatility of functions. We will see later that our ability to synthesize data and produce global numbers (like percentages, averages and sums of columns) are all based on functions. We will come back to this when we finish exploring the SELECT command.

Saving and Calling a Query.

At this point you should save your query (File> Save, or click the icon with the diskette). You will be asked a name for this query (I used “qryPsyPhoneList”). If you close the query now and go back to the Base interface, you will see that this query has joined our list of available queries. If you double click on it, the resulting table will appear.

Because this query is executed when it is called, any additions or deletions to the Psychiatrist table will be reflected here, that is, the query updates itself every time it is called. Now, that is very useful.

Should you want to edit this query, right-click on the name of the query and select “Edit query in SQL”. If, after you amend your query, you want to keep both versions, simply save this new query with “Save as...”.

Where?

So far you know how to select columns for a query, give them an alias if necessary and even run them through a function.

The SELECT command also allows you to pick a subset of your total data. You do this by imposing a condition with the WHERE clause. HSQL will go through all the records checking this condition. If the evaluation returns TRUE, that record will be displayed with the resulting set. If the evaluation of the condition for that record returns FALSE then the record will be skipped.

Let's imagine that you have an interest in listing all the patients of the clinic that are male only. In this case, you would use:

```
SELECT * FROM "Patients"
WHERE "Gender" = 'Male';
```

You can see that we are using the wild card, which means that all columns from the "Patient" table will be returned. However, only the records that have 'Male' in the "Gender" column will satisfy the condition imposed by the WHERE clause and, consequently, only those records will be displayed.

Notice the use of the equality sign "=". You have all comparison operators at your disposal: <, >, <=, >= and even <> (or !=). You can compare columns to strings, dates and numbers and even to other columns.

Using these operators with numbers is quite straightforward. When you compare dates you should know that an older date is considered "*less*" that a more recent one, so:

(Date1 < Date 2) will return true only if Date1 is an older date than Date2.

String comparison can also use these mathematical operators. In this case, they represent the position of the letter in the alphabet. 'A' is *less* than 'Z', 'C' is *bigger* than 'A'. In the comparison, if the first letters are equal then the second letters are compared, and so on.

Let's say that you want to break down your list of patients so several assistants can work on them simultaneously. Lets say that your first assistant will work with all patients where the surname starts with A, B or C. Then you can produce this list by requesting:

```
SELECT * FROM "Patients"
WHERE "Surname" < 'D';
```

Please note that in all these examples, the name of the tables are enclosed by double quotes while the string literals are enclosed by single quotes. This is how SQL tells them apart.

The WHERE condition accepts expressions comparing the value in a column with the NULL value, like this: WHERE <column_name> IS [NOT] NULL. The square brackets mean that you can decide to include or not the “Not” statement, depending on what you are looking for: the record to have a null value or the record NOT to have a null value in the particular attribute. In order to find a more comprehensive list of accepted expressions, check the documentation provided by the HSQL website at:

http://hsqldb.org/web/hsqlDocsFrame.html

Now, you can ask for exact matches (with the '=' sign) but also can request *similar* matches using the LIKE operator, which you use instead of the mathematical operators.

Because you, more or less, know the *kind* of matches you are looking for, SQL offers you two other wild cards to help you be more precise with your condition: The '%' (percent) sign and the '_' (underscore) sign. The '%' stands for any character, any number of times (including *zero* times). The '_' sign stands for any character, exactly once.

Let's see this in action:

```
SELECT * FROM "Patients"
WHERE "Surname" LIKE 'B%';
```

This will select all names that start with the letter “B”, no matter what characters -or how many characters- appear after the 'B'. If you had a record that only had a 'B' for surname, it would also be included in the result.

```
SELECT * FROM "Patients"
WHERE "Address" LIKE '%Main St.%';
```

This query will select all records that have 'Main St.' somewhere in the address. Notice that I used the '%' sign not only at the end of the string literal to match but also at the beginning. If I had omitted the first '%', it would have meant that I am looking for records whose address start with 'Main St.'. If I had omitted the last one, it would have meant that I am looking for records that end with 'Main St.'.

Believe it or not, these two wild cards are enough for many complex combinations you could want to match.

Compound queries

You can include more conditions to the WHERE clause with AND, OR and NOT to make more complex queries.

Lets say that you are planning a surprise party for Belinda, one of the psychiatrists, and wish to invite all female psychiatrists that practice in your same ZIP code (which, for this

example, will be 13044). Of course, Belinda should not get an invitation or the surprise would be spoiled. The list your assistant needs could be composed with the following:

```
SELECT * FROM "Psychiatrist"
WHERE "Gender" = 'Female'
AND "Zip" = '13044'
AND NOT "First Name" = 'Belinda';
```

Note that this query will exclude all psychiatrists that have 'Belinda' as a first name, not only *our* Belinda. To avoid this you might want to include the surname as part of your query. But you see where I am going: you can make SELECT commands as complex as your search might require.

Order in the room, please!

Now, the output table of your query might not show any particular order. The database went about collecting the records that conform to your conditions and then displayed them in the order they were found. Many times you need to sort the result based on some parameter. In order to achieve this, the SELECT command accepts an ORDER BY clause.

Let's say that you want the list for the party ordered alphabetically according to the surname. Then you should ad the following clause:

```
SELECT * FROM "Psychiatrist"
WHERE "Gender" = 'Female'
AND "Zip" = '13044'
AND NOT "First Name" = 'Belinda'
ORDER BY "Surname" ASC;
```

ASC means that you want the list in ascending order (remember that 'A' is *less* than 'Z'). Optionally, ORDER BY also accepts DESC for descending, that is, from *more* to less (from 'Z' to 'A').

For example, if you wanted a list with all your therapists, ordered according to years of service, with the more recent additions to the top of the list, you should request:

```
SELECT * FROM "Therapist"
ORDER BY "Hiring date" DESC;
```

User defined parameters.

Let's say that as the director of the clinic you need to find a particular patient whose name was Sanders or Sandoz or something like that, that was admitted sometime between 2004 and 2006. Instead of going through 2500 records one by one, you can create a query:

```
SELECT * FROM "Patient"
WHERE "Surname" LIKE 'San%'
AND "Time of Registry" <#2007/01/01#
AND "Time of Registry" >#12/31/2003#
ORDER BY "Surname" ASC;
```

Note that the dates are enclosed by "#" signs, so Base can tell that they are dates. Also note that I have used two formats for entering date: the ISO format that uses yyyy/mm/dd and the ***Local*** format used in the US that uses mm/dd/yyyy. Of course, I should stick to using one or the other for consistency but I want to show that I can use either. If you live in Europe or Latin America, your Local format would be dd/mm/yyyy. The local format is set when you chose the Language Settings for OpenOffice.org.

This could be a very useful query for finding patients, except that you will have to write it again, changing name and dates, every time you are looking for patients with other parameters.

Actually, Base allows you to leave variables in place of your parameters, and ask you to fill them at the time you run the query, without having to change the code.

This is how it works: instead of defining the parameters (in this case, the name and the date limits) you write the operator followed by a colon (the ":" sign) and then a variable name. It will look like this:

```
SELECT * FROM "Patient"
WHERE "Surname" LIKE: patientName
AND "Time of Registry" <: topDate
AND "Time of Registry" >: bottomDate
ORDER BY "Surname" ASC;
```

This is what it's going to happen when you run the query:

Base will display a small window called "Parameter Input". At the top of the window you will see a label that reads "Parameter" and immediately below it a box with the three variables present in this query: patientName, topDate and bottomDate. Under the box you will see another label the reads: "Values" and then a single text entry box. You then enter the parameters for your search in the order they are asked for, pressing the ENTER key after each one. The window also displays the buttons: *OK*, *Cancel* and *Next*. You can also enter the values by clicking on *Next* after each value and *OK* after you enter the last one.

After that, the matches will appear in table format ordered by surname from A to Z.

Because we built our query with the LIKE operator for patient name, you are entitled to use the wild cards "%" and "_" just like we have been doing so far. The dates you enter need to be enclosed by "#" signs[10], so Base can tell that they are dates. Note that the dates given are excluded from the search. For example, you could enter this:

10 I have also entered date information without the '#' sign and Base responded without problem.

Smi%
#01/01/2005#
#12/31/2003#

The query will look for all Surnames of patients that begin with Smi (like Smith or Smithers) that were admitted in 2004 (but not in the last day of 2003 or the first day of 2005. For that you would need to include the equality sign "="). You can run the query again and again, making new searches without having to amend your SQL code.

Querying more than one table at a time

The things that you can do by now with the SELECT statement are already pretty impressive. We take this to the next level when we think about the ability to use the data in more than one table to create a query.

Imagine the following circumstance: You need a list of of all your patients and their psychiatrists. The first and last name of patients is in one table and the first and last names of the psychiatrists is in another table. How do we do this?

First, we need to reference both tables being used in the FROM clause of the SELECT command. Also, we need a convention to make explicit what tables are we extracting our columns from. Notice that both, the "Patient" table and the "Psychiatrist" table have columns called "First Name" and "Surname". In order to tell them apart we are going to use the following syntax in our query:

<table name>.<column name>

This is to say that we are going to use the name of the table and the name of the column, separated by a period, to unequivocally reference the columns that we need. Following the example, we should write:

```
SELECT "Patient"."First Name",
          "Patient"."Surname",
          "Psychiatrist"."First Name",
          "Psychiatrist"."Surname"
FROM "Patient", "Psychiatrist";
```

Please note that, because our names for columns and tables use upper and lower case and spaces, we need to enclose them in their own set of double quotes.

Let's assume that you have exactly the following three patients in your database: Alex Smith, John Serrato and Frank Thrung; and exactly the following two psychiatrists: Amanda Perez, Debbie Dobson. Also, we know beforehand that Alex and John have Amanda as their psychiatrist and Frank has Debbie.

However, if we run the above query we get something like the table in the next page:

First Name	Surname	First Name	Surname
Alex	Smith	Amanda	Perez
Alex	Smith	Debbie	Dobson
John	Serrato	Amanda	Perez
John	Serrato	Debbie	Dobson
Frank	Thrung	Amanda	Perez
Frank	Thrung	Debbie	Dobson

This is clearly not what we expected. First of all, and if we look carefully, what we really have is a result where every patient is combined with every psychiatrist. We have 3 patients and 2 psychiatrists and a resulting set of 2 x 3 = 6 combinations. What we really want is the information of which patient has which psychiatrist. Second, both columns read “First Name” and “Surname” and, with no other clue, we can't tell which is a patient and which is a psychiatrist.

Well, this last thing is easy to correct. All we need to do is add an alias for each column. In general, when we are working with more than one table, it is a very good practice to use an alias for each column to avoid these uncertainties. It is also more elegant.

Now, the first problem is also easy to solve. If you remember, each psychiatrist has a primary key that appears as a foreign key in the patient's table. If you don't remember, review the SQL code that created the tables and the ULM diagram that describes them. It is this foreign key that tells us which psychiatrist sees which particular patient.

So what we want to do is produce a result set where the foreign key in the “Patient” table is identical to the primary key in the “Psychiatrist” table. Of course, we will use a WHERE clause to include this condition. Consequently, our SQL code will look like this:

```
SELECT "Patient"."First Name" AS "Patient Name",
          "Patient"."Surname" AS "Patient Surname",
          "Psychiatrist"."First Name" AS "Psychiatrist Name",
          "Psychiatrist"."Surname" AS "Psychiatrist Surname"
FROM "Patient", "Psychiatrist"
WHERE "Patient"."Psychiatrist ID" = "Psychiatrist"."ID Number";
```

Every time we are joining tables, we will need to match foreign and primary keys. Otherwise, we will get results like the one on the top. Such result, that shows all possible combinations between patient and psychiatrist, is known as a *Cartesian Join*. The WHERE clause that help us match key numbers transforms this into an *Inner Join*, which gives us the information we want.

With an inner join we only select the instances that match our conditions. This is fine most of the times but not always. Let's imagine that instead of two psychiatrists you have twenty three and that instead of three patients you really have five hundred and twelve. What you don't know is that two of your psychiatrists are currently not seeing any pa-

tients. If you run this inner join you will produce a list that associates every psychiatrists that has a patient with her corresponding patients but that excludes from the result set all psychiatrists that do not have patients. This fact will not be obvious with a long list like this one and it will take you a serious analysis of the list and memorizing the names of all the psychiatrists to note the exclusions. It would be much better if the result of our query includes all psychiatrists, even if they don't match our conditions.

To accomplish this we can use an O*uter Join*. With this join we are requesting to include all the information in one table even if it is not associated with the corresponding information in the other table. This result set will display all the psychiatrists but will leave blank the cells for patients if they don't have one. This would help you know that there is information that you don't have, which can be a valuable bit of information in itself.

Because SQL, like English, is a linear language, you need to write the name of one table first and then the name of the other table second. If you want the query to produce the complete information in the table written to the left then you want a *Left Outer Join*. If you want the query to include all the information from the table written secondly (which would be placed to the right of the first one) then you want a *Right Outer Join.*

The syntax for an outer join is a bit more complex at first but you will see that it makes complete sense. Let's produce the query that will include all the psychiatrists even if they do not have a patient in our database -so that this fact becomes obvious!

```
SELECT "Psychiatrist"."Surname" AS "Psychiatrist Surname",
       "Psychiatrist"."Name" AS "Psychiatrist Name",
       "Patient"."Surname" AS "Patient Surname",
       "Patient"."Name" AS "Patient Name"
FROM {OJ "Psychiatrist" LEFT OUTER JOIN "Patient"
      ON "Psychiatrist"."ID Number" = "Patient"."Psychiatrist ID"}
```

Note that the FROM clause starts with an opening curve bracket ('{') and 'OJ' for Outer Join. Then we write the name of a table, in this case the *Psychiatrist* table which, because we wrote it first, is placed to the left in the declaration. Then we write 'LEFT OUTER JOIN' and then the name of the second table, in this case *Patient*. Because we are asking for a left outer join, it is the table to the left (*Psychiarists*) from which all names and surnames will be extracted even if they do not have patients that will match foreign keys. Note that instead of writing 'WHERE' we use the keyword 'ON' to set our matching conditions. Finally, we finish with a closing curve bracket ('}'). Run this query now. Nice!

Let's now return to our inner join in the previous example. Say we also want to include the patient's phone number as a result of the query. If you remember, this data had been placed in its own table because we wanted to record an unspecified amount of phone numbers for the patients.

What we will need to do is to include the Patient Phone table to the query (with the FROM clause), use the proper <*table*>.<*column*> syntax and include an AND condition to the WHERE clause that links the primary key of the patient with the patient ID foreign

key in the Phone table. Try writing such query now. If you notice, joining two tables requires one WHERE statement. Joining three tables requires two WHERE statements, connected by an AND. You can start seeing a pattern here: Joining four tables will require three statements. In general, if you are joining n tables, you will require n-1 joins.

As an exercise, do write this query that asks for a psychiatrist and lists all his/her patients with their respective phone numbers.

Intermediate tables are no more difficult that what we have done so far. If you remember, we use intermediate tables between two tables that have a Many to Many relationship (m..n). The use of an intermediate table transforms that cardinality to a manageable 1..n. However, an intermediate table basically just stores several combinations of foreign keys. What we want to do is to extract the *meaning* from those combinations. In our example we have an intermediate table between patient and medication. One patient can use one or several medications. One medication can be used by one or more patients.

Let's say that we want to list all the medications used by a patient. This is stored in the intermediate table, which holds the patient's key number associated with the key numbers of the medications he/she uses. But this table only lists such combination of key numbers. In order to know what they mean, we also need to consult the “Patient” table and the “Medication” table. To avoid a Cartesian join, we need to include the appropriate conditions. Because we are joining three tables, we know that we will need two joins. To make the result of this query easy to read, we will also include informative aliases. Can you write this query before looking further? You should!

```
SELECT "Patient"."First Name" AS "Patient Name",
       "Patient"."Surname" AS "Patient Surname",
       "Medication"."Name" AS "Medication",
       "Medication"."Description"
FROM "Patient", "Medication", "Patient Medication"
WHERE "Patient Medication"."Patient ID" = "Patient"."ID Number"
AND "Patient Medication"."Medication ID" = "Medication"."ID Number"
ORDER BY "Patient"."Surname" ASC;
```

Let's dress up this query by including other elements reviewed in this part: Let's make the query ask us for the patient for which we want the report and let's have the output of the name be displayed in one column:

```
SELECT CONCAT(CONCAT ("Patient"."Surname", ', '), "Patient"."First
Name") AS "Patient Name",
       "Medication"."Name" AS "Medication",
       "Medication"."Description"
FROM "Patient", "Medication", "Patient Medication"
WHERE "Patient"."First Name" LIKE: PatientName
AND "Patient"."Surname" LIKE: PatientSurname
AND "Patient Medication"."Patient ID" = "Patient"."ID Number"
AND "Patient Medication"."Medication ID" = "Medication"."ID Number"
ORDER BY "Patient Name" ASC;
```

Note that the order clause uses the name given to the column by the alias. Remember that aliases not only change the heading of the column but also allow us to reference the column with that name in the SQL code.

This is as complex as our SELECT statements will ever get, but by now you should be able to read them, and produce them, with no problem.

Aggregate Functions:

Aggregate functions also return a value, like regular functions, but instead of returning one value for each record evaluated, they return one value that represents the entire collection of evaluated records.

The most clear example of this is the COUNT function. COUNT will return the number of records in a specified table. Let's say that you need to know how many patients have been recorded in the database. Then you can run the following instruction:

```
SELECT COUNT(*) FROM "Patients";
```

which, in my computer, returns a result like this:

COUNT(*)
8

(Because I only entered data for eight patients)

Now, believe it or not, this result is still a table, albeit with only one column and only one row (plus the header row with the name of the column).

Compare this result with the COS(d) function that we reviewed earlier. In that example the parameter 'd' stood for a column name and the function returned the cosine for each and every angle recorded in that column. Aggregate functions, on the other hand, return a summary number that represents an aspect of the entire set.
We will briefly review the following aggregate functions:

COUNT, MIN, MAX, SUM, AVG, VAR_POP, VAR_SAMP, STDDEV_POP, STDDEV_SAMP.

In general, these functions use the following syntax:

SELECT <*function*> **("Column Name") FROM "Table Name" [...]**

where <*function*> stands for one of the 9 aggregate functions named above.

Let's check them out:

● As sated, COUNT returns the number of rows in a table. Of course, we can filter this number to obtain a count of subsets. For example, say that you need to know the number of male patients in the patient table. I am sure that you can already anticipate the needed code:

```
SELECT COUNT(*) FROM "Patients"
      WHERE "Gender"='Male';
```

You can also change the header row for a more descriptive name of your result with an alias.

```
SELECT COUNT(*) AS "Number of Male Patients" FROM "Patients"
      WHERE "Gender"='Male';
```

Let's say that you need to know the number of *unique* phone numbers in the Phone Number table. We can imagine that two or more family members that live together share at least the home number, so a simple count instruction will not do: there will be *repeats* that are counted. We can use the DISTINCT qualifier to eliminate repeats from a query, like this:

```
SELECT DISTINCT COUNT("Number") FROM "Phone Number";
```

In short, if the DISTINCT qualifier is included, only one instance of several equivalent values is used in the aggregate function. If we wanted the contrary, that is, to make sure that we include all instances in our query, we can use the qualifier ALL.

You should know that the data type of COUNT is Integer, in case that you want to do some kind of arithmetic operations with it or wish to use this result with a drop-down list. We will come back to this.

Also note that COUNT will include rows even if they have null values. For example, if you request the query:

```
      SELECT COUNT("Surname") FROM "Patient";
```

even if you have some records that do not yet have a surname, they will be counted anyway.

Being able to obtain the total number of records in a table and the count for subsets of it allow us to determine all kinds of percentages that can describe the entries in our database.

You can do this calculation by hand or have Base produce it for you. The truth is that getting such a number requires some work but, once you have it all in place, you can get updated results with a simple double click of a mouse. Let's say that we want to calculate

the percentage of patients in the database that have been diagnosed with post traumatic stress disorder (PTSD for short).

From a conceptual point of view, we first want to calculate the total number of patients in the database. We can use a simple SELECT COUNT clause for this. Next we need to calculate the number of patients that have been assigned a diagnosis of PTSD, for which we include a WHERE clause to match the condition. Now we divide the number of PTSD patients by the total number of patients, which gives us a decimal number between zero and one, and then multiply this number by one hundred to get the percentage, something like this:

$$\% \text{ PTSD Patients} = (\frac{n_{ptsd}}{n_{total}}) x\ 100$$

Notice that in calculating a percentage, the subset always goes in the numerator (above the division line) and the total always goes in the denominator (below the division line).

One difficulty of this procedure is that we need two SELECT clauses to produce this result, one with a WHERE clause and one without it. We can't have both SELECT instructions in the same query. How do we solve this?

Well, we produce two queries for each SELECT and then we produce a third query that combines the results of the previous queries. Here is when you could say: *Nah! leave it. I'll just calculate the two COUNTS and then divide their results with my calculator.* Fair enough. But with the three queries in place, in the future, after the database has seen new additions, you just call for the third query and you will have an updated percentage of patients diagnosed with PTSD.

The complete procedure is as follows: You create the first query for the total number of patients with code like:

```
SELECT COUNT (*) AS "Total Patients" FROM "Patients";
```

Then save the query with a name like “qryTotal Patients”.

Next you create the query for the subset, patients with PTSD, with code like:

```
SELECT COUNT (*) AS "Total PTSD Patients" FROM "Patients" WHERE
"Diagnosis"='PTSD';
```

and save it with a name like: “qryTotal PTSD”.

Now, at the Base interface in the Queries view, we right click on these queries and select “Create as view”. If you remember, this will create virtual tables with the result of the queries. Virtual, because the tables will be updated reflecting any relevant changes to the

records in the database every time they are called. For the names of the views, leave the same name but delete the 'qry' component of the name. You find these views in the Tables section of the Base interface.

For the third query we will call the results from the two previous queries with our dot syntax using first the name of the view and then the name of the column as established by the 'AS' clause, with code like the following:

```
SELECT ("Total PTSD"."Total PTSD Patients"/"Total Patients"."Total
Patients")*100 AS "Percentage of PTSD Patients"
FROM "Total PTSD", "Total Patients";
```

Notice that this SQL code not only selects the columns that we want but also includes arithmetic operations (the '/' sign for division, the parentheses and the "*" sign for multiplication). This is completely valid code and, in fact, most database engines like HSQL allow SELECT to modify the output with mathematical operations like these.

But alas, if we run this query we get a result of zero! What happened?!

Well, we know that the division produces a number between zero and one (because the subset is a fraction of the total) but, if you recall, the COUNT function produces an Integer data type, which means that the result is always rounded down -in this case to zero. Zero by one hundred is zero. What can we do?

One very clever solution is to change the formula, multiplying by one hundred before making the division, something like this:

```
...("Total PTSD"."Total PTSD Patients"*100)/"Total Patients"."Total
Patients"
```

From a mathematical point of view both formulas are equivalent and from a practical point of view we avoid being completely rounded down to zero. But this solution is not entirely satisfactory because we still lose all decimal numbers, which can add to produce important error in our calculations[11]. The problem is the Integer data type of the COUNT result.

The functions will save us again. If you check the HSQL documentation that we have cited you will discover two functions that allow us to change the data type of our result. These functions are:

`CAST (term AS type)`	converts *term* to *type* data type
`CONVERT (term, type)`	converts *term* to *type* data type

11 You could multiply by ten thousand to keep two decimal numbers, but because we do not have a decimal separator (comma or period) this solution makes reading the result ambiguous.

Their descriptions in the documentation suggest that they are equivalent in their result and only change in the syntax they use. Maybe this is not complete and there are differences in the way they behave. We will only know this by trying them in different situations and comparing them. For now, I will just use the CAST function to achieve my goal, and will assign my results a Real data type, like this:

```
CAST ("Column 1" AS REAL) / CAST ("Column 2" AS REAL) * 100
```

Where: Column 1 = `"Total PTSD"."Total PTSD Patients"` and
Column 2 = `"Total Patients"."Total Patients"`

so the total code of our last query would look like this:

```
SELECT( CAST ("Total PTSD"."Total PTSD Patients" AS REAL ) / CAST
( "Total Patients"."Total Patients" AS REAL )* 100) AS "Percentage
of PTSD Patients"
FROM "Total PTSD", "Total Patients";
```

This way we will get enough decimal numbers and the proper format in the calculation of the percentages.

Remember that we can change the data types in the last query, like in the example above, but could also change them in the queries that produce the partial results that we need.

> *For the fun of it, create a query that compares the percentages of PTSD patients that were active three, two and one year ago with the patients active today.*

● MIN and MAX will browse the specified column and will identify the smallest and the biggest number, respectively, of your set. Obviously, these functions apply to numeric data types only. The resulting number will be of the same data type as the column over which the function operated.

● SUM will do exactly that: it will return the sum of all the values in the specified column. This comes in very handy, for example, when we need to know the total of sales, earnings or losses that we have tabulated. Again, SUM applies to numeric data types. HSQL will assign a variable type to this result in a way to ensure loss-less results, that is, with a level of precision commensurate to the result of the sum.

We can describe SUM mathematically. Let's assume that you have n records (of numerical data) in a column, then the SUM of that column returns:

$$\text{SUM (X)} = \sum_{i=1}^{n} x_i$$

Note that SUM, MIN and MAX will exclude null values from the calculation.

● Statistical functions include: AVG, STDDEV_POP, STDDEV_SAMP, VAR_POP and VAR_SAMP.

AVG calculates the average of the numbers in the specified column by calculating the sum of all the records and then dividing that by the total number of records. To be precise, this function performs the following calculation:

$$\text{AVG} = \overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

The average is a tool of statistics that allows us to describe a collection of quantitative measurements. For example, we cite the average income of the people living in one neighborhood or the average time of reaction to a specific environmental signal. To fully appreciate how representative that average is of the set it describes, we will also want to know how close of far apart each particular measurement is from that average. The standard deviation is *an average* of the deviation of all the particular measurements from the set's average[12]. A smaller number means that the individual measurements are close to the average, a bigger number means the opposite.

Not surprisingly, STDDEV_POP calculates the standard deviation of the numbers in the specified column. Conceptually, STDDEV_POP represents the following formula:

$$\text{STDDEV\_POP} = \sigma = \sqrt{\frac{\sum (X_i - \overline{X})^2}{n}}$$

However, it is more likely that the standard deviation is calculated by HSQL using the following formula, which mathematical science has shown to be equivalent to the previous one but does not require the computation of the average:

$$\sigma = \sqrt{\frac{\sum x^2 - \frac{(\sum x)^2}{n}}{n}}$$

The average is of the same data type as the column from which it's calculated. Meanwhile, the standard deviation is always assigned a Double data type, to ensure its precision.

STDDEV_POP assumes that the values in the column belong to all the members of the population that we are trying to describe. If this is not the case, we say that the measurements belong to a *sample* (a subset) from the population. In this case, statistical theory requires that we adjust the formula to account for this:

12 Computed using the sum of squares of the distance between each measurement and the set's average.

$$\text{STDDEV\_SAMP} = S = \sqrt{\frac{\sum (X_i - \overline{X})^2}{(n-1)}}$$

Conceptually, that is the value that STDDEV_SAMP will produce, although it is more likely that HSQL will compute it using the equivalent formula:

$$S = \sqrt{\frac{\sum x^2 - \frac{(\sum x)^2}{n}}{(n-1)}}$$

The Variance is another measure of variability used in statistics. From a mathematical point of view it corresponds to the square of the standard deviation. VAR_POP assumes that it describes the entire population, and is defined by:

$$\text{VAR\_POP} = \sigma^2$$

while VAR_SAMP assumes that it is describing a sample from it and, consequently, corresponds to:

$$\text{VAR\_SAMP} = S^2$$

However, it is more likely that these values are calculated by formulas like the ones used by STDDEV_POP and STDDEV_SAMP, correspondingly, but without extracting the square root.

Variance is also assigned a Double data type by HSQL.

Note that none of these statistical functions allow for ALL or DISTINCT qualifiers and that they will exclude null values in their calculations.

Let's now tackle a problem that uses statistical functions: let's produce a report that calculates the average age (and the standard deviation, to make sense of this number) of all women with a diagnosis of depression in our database. Can you imagine the excitement of a researcher that is hinted on the idea that most women attending for depression significantly belong to the same age/phase in life?

I am sure that you have spotted the need to use the WHERE clause to extract the group we are interested in: Women with a diagnosis of depression. More subtle is the fact that we have not recorded the age of any woman but instead we have their dates of birth. What do we do now?

Some time functions:

If you are thinking that maybe there is a function for this... you are absolutely right. In fact, time calculations seem to be very important in our society (determining when to pay dues, when to collect, the calculation of compound interest rates, analysis of last quarter, projections for next year, etc.) so it is not strange that SQL code comes with very mature tools to handle time calculations. You really need to check this out in the website for HSQL that we have cited before.

In order to obtain the ages of our sample we will need to calculate the difference between their birthdays and today's date. Can we do this? Not only we can, but we can also specify the time unit that better suits our needs: do you want the result in years? In months? In years and months? In days? In seconds? (Yeah, seconds). For our needs, months seems precise enough. If you review the HSQL website at

http://hsqldb.org/doc/2.0/guide/sqlgeneral-chapt.html#sqlgeneral_types_ops-sect

you will see that the code required for this looks like:

```
(value1 - value2) <time unit>
```

Where value1 and value2 can be columns that hold datetime data, can be constants (a particular date or time value), can be the result of a function that provides datetime data, like `CURRENT_DATE()` and can even be another time interval. The only condition is that both values are of a comparable nature: you could not compare a time value (that gives hours, minutes and seconds) with a date value (that provides year, month and day).<Time unit> options include: *year, month, day, hour, second* and *year to month* which provides the result in years and months.

In order to get our age data, we try the following snippet of code:

```
(CURRENT_DATE - "Date of Birth") MONTH
```

This code calls for the CURRENT_DATE function, which fetches today's date, and subtracts from it the date of birth of the patient, providing the difference in months. Now, this grammar is correct according to the cited documentation. There is also a DATEDIFF() function. It essentially works with the same logic but you write it differently, like this:

```
DATEDIFF (<time unit>, value1, value2)
```

where value1 is the starting date of the interval and value2 is the ending date (most current date) of the interval. Consequently, the code we need would look like this:

```
DATEDIFF (MONTH, "Date of Birth", CURRENT_DATE)
```

Now, comparing the date of birth with today's date makes sense only for cases currently active. To filter this we should include a WHERE clause that selects only those patients

that are female, that have a diagnosis of depression AND have an empty "Date Case Closed" cell in the Assignment table. For a more general evaluation, we would want to get the age in relation to the "Date assigned" that we stored in that same table. Then, the snippet of code could look like this:

```
("Date assigned" - "Date of birth") MONTH
```

Or, depending on your version:

```
DATEDIFF (MONTH, "Date of birth", "Date assigned")
```

Now I am going to divide this result by 12, which will provide the age in years with a decimal extension for the months (so 27 years and six months of age will appear as 27.5) and then I will request the calculation of the average and the standard deviation. The final code could look something like this:

```
SELECT AVG((("Assignment"."Date assigned" - "Patient"."Date of
birth") MONTH)/12),
STDDEV_POP((("Assignment"."Date assigned" - "Patient"."Date of
birth") MONTH)/12)
FROM "Assignment", "Patient"
WHERE "Patient"."Gender" = 'Female'
AND "Patient"."Diagnosis" = 'Depression'
AND "Patient"."ID Number" = "Assignment"."Patient ID";
```

For completeness, if you want to know the interval of time between dates in a column and a specific day -that is, a constant-, you can specify the constant's format and then provide the data. For example, if I want to calculate the age (in years and months) that all patients had in, say December 16th, 2007 then the code would look like this:

```
SELECT ("Date of birth" - DATE '2007-12-16') YEAR TO MONTH
FROM "Patient";
```

You can also work with time and timestamp data types, defining them like this:

```
TIME 'hh:mm:ss'
TIMESTAMP 'yyyy-mm-dd hh:mm:ss'
```

As an exercise, write the SELECT code that compares the average age (and standard deviation) of men with PTSD that were receiving therapy 10 years ago, 5 years ago, 3 years ago and that are active today.

Chapter 10

Creating reports with Base.

If you notice, there is currently no way to print on paper the tables you have created in the "Queries" view. For this to happen you need to create a report.

The report will provide the same information found in the table created by your query. [13] The nice thing about a report is that you can customize its layout, add a company logo, add info about the date or the person who created the query and things like that so that when you print it, these elements will frame the validity of the information provided in your report, will make it easier to read and will also make it look quite professional.

For our master exercise I want to create the Clinical Summary Report. This is the report where we provide contact and clinical information about a particular patient, his/her medical doctor, psychiatrist if there is one, medication and other data.

If you now click on "Report" in the left column of the Base interface, the task bar will offer you the option "Use Wizard to Create Report". This option is quite straightforward: The wizard will make a series of questions about what table or query you want to use, what fields you want to write, how do you want to group them and what template for layout you want to use. Actually, by now you should be able to understand most of the options offered and I recommend that you play with the wizard and become familiar with the way it works. For most things, the wizard will be just the right tool.

The only thing is that the Report Wizard will always display the result of your query as a table. This way, the result of the query for the Clinical Summary will be displayed as one long row. Because we are only interested in one particular patient, there will be no other rows to fill the page. You will end with one (quite long) row, spanning several pages, but leaving most of the printed page empty. This is not a satisfactory solution and departs greatly from the layout we did in Part II.

Don't despair. Sun Microsystems has made available a "Report Builder" that allows you the flexibility of design view in re-organizing the layout of your reports. There is no reason why you should not download it. So go ahead and install it now.[14]

After you download and install the extension you will find that the Tasks menu in the Report section offers you a second option: "Create Report in Design View".

Just to get acquainted, click on this option to see what we get.

---

13 Actually, you can even chose to omit, reorganize or summarize certain items in your query before printing them.

14 Although one of the beauties of the extensions is the way they integrate with OpenOffice.org so that you can use them straight out of download without having to restart, the integration with the SRB is not so seamless. I found that it is better if you quit OpenOffice.org and the Quickstarter before you open it. In the case of Windows XP I've had less trouble if I simply restart the computer after download.

Wow, I bet that this was more than what you bargained for! Don't worry, all these options are here to help us.

Below all the menus and toolbars there is a page divided in three. To the left you can see that the top section is the Header, the middle section is the body of the report, called “Detail” and to the bottom we have the footer. To the right we have a column with two tabs: *General* and *Data*. They work just like the Properties Dialog Box that appears while working with the Forms in Design View, providing the characteristics of any objects we select and allowing us to customize them.

Here is an important trick: in the *Data* tab of the report's properties you find the box that allows you to connect the report you are editing with one particular table or query. If you select any section of the report, or a particular label or text box, the Properties Dialog Box and the corresponding tabs change to reflect the properties of that particular object. If you need to go back to the properties that belong to the report then you must click outside of the report, typically, on the gray area below the report layout.

Let's focus on the three sections in the page. You can click on any to activate it. The tab to the right shows options to customize each section. You should read them now to become familiar with them. If you don't see a tab to the right, go to View>Properties, or click on the *Properties box* button (usually second from the left in the lower row of the toolbars).

The header section will receive data that you want displayed on the top of every page in the report (good for titles, dates, company logos and things like that). The footer is similar, displaying the info at the bottom of every page (good for page numbers, contact info and the like). The central part of the page, labeled “Detail” will receive the information from the tables and will repeat itself for every record collected by the query. This info will appear in Text Boxes, which we will tag, if needed, with Labels.

You will notice the property: “Height” followed by some unit in the properties panel. You can change the height of any section by either changing the value here or by dragging the horizontal line that separates the two sections.

Let's assume that you want to insert a label. First you select the Label button (just to the right of the Properties button). Now you can click and drag the label box anywhere in your form. Immediately after, you will see this object's properties appear in the column to the right. Find the “Label” option and write there “Name:” to see the text in the label change in your form.

Let's now place a Text Box next to our label. You will find the Text Box button next to the Label button. Click, place and drag. You will see that in the column to the right there are two tabs this time: *General* and *Data*. Study them. If you examine the *Field* property in the *Data* tab you will see that it is empty. This is where the connection to a particular field in our table is defined and can be set automatically by the report builder or by you, as we will see later.

It goes without saying that you should also read a very good reference found here:

http://wiki.services.openoffice.org/wiki/SUN_Report_Builder/Documentation

Our first Report with Base:

Before attempting our more ambitious project of developing a Clinical Summary report, let's do something simpler, just to get the hang of it. For this exercise we are going to use both, the Wizard Report and the Sun Report Builder. Later on, when we tackle the clinical summary, we will only use the Sun Report Builder from beginning to end. However, there is nothing wrong with using this dual approach: the report wizard will have us up and running quite quickly and then we can use the Report Builder to fine tune our work.

Our report should produce a list of all patients in our database with their corresponding phone numbers. This exercise should be interesting for a number of reasons: First, it uses data from two different tables that need to be properly linked; second, we have an unknown number of phone numbers for each patient: some patients will have one but others can have two, three or more phone numbers. This gives our report a degree of unpredictability. Lastly, we are going to use the CONCAT function to create a column that we will name with an alias and then we are going to reference that column with the alias in another part of the SQL for our query. It is going to be very interesting and informative to see how do the report wizard and the report builder respond under these circumstances. Before advancing further, make sure that you have some entries in the Patient Table and that you give them differing amounts of phone numbers.

We should start by coming up with a desirable layout for the report. I would like it to have a heading that identifies the report clearly, something like “Patient's Phone Numbers”. We should also include the date in which we generated the report so that we know up to what moment in time this report is valid. I also want to number the pages so that I know if one has gone missing. Then we should have the name of the patients, surname first and separated from the first name by a comma and a space, and ordered alphabetically. I want to have the phone numbers underneath the names and in line with a reference to which number this is: home, office or mobile phone, etc.. Now that I am scribbling the layout of this report on a napkin, I realize that I would also want to include some kind of graphic, like an horizontal line, that would help me separate each record, making it easier to scan.

We will obviously need the “Phone Number” table, which holds the number and descriptor. But the names are not located here. For this I need the “Patient” table. We will need to use two CONCAT functions to produce the output for the name that we requested in the previous paragraph and we will have to name this new column with an alias so we can reference it in our ORDER BY instruction. We are going to need other columns so that we can properly join these two tables -mainly primary and foreign keys, although those columns will not be part of the output. Because we are joining two tables we know that we need only one join.

All right then, can you produce the SQL code for the query that we need before reading further? You should!

```
SELECT CONCAT( CONCAT( "Patient"."Surname", ', ' ), "Patient"."First
Name" ) AS "Patient Name",
      "Phone Number"."Number",
      "Phone Number"."Description"
FROM "Patient", "Phone Number"
WHERE "Patient"."ID Number" = "Phone Number"."Patient ID"
ORDER BY "Patient Name" ASC;
```

Let's produce our query now. Click on the *Queries* button in the Base interface and then click on *Create Query in SQL view...*. Now type the SQL code and, just to make sure it works, run it (by clicking on the box with the green tick on top of the two overlapping papers). If every thing goes fine, you should see three columns, one with the header "Patient Name" and the other with the names: "Number" and "Description". The names should be in the "Surname, First Name" format and ordered alphabetically. The names should be repeated for every different phone number the patient has.

At this moment we need to save the query. You do this by clicking on File>Save or by clicking on the diskette symbol. I used the name *qryPatientPhoneNumber* but hey, that is just me and you can use any name you feel comfortable with.

We are ready to build the report now. Click on the Report button on the Base interface and then select *Use Wizard to Create Report...*

At this moment you will see the Report Builder pop up and, over it, the Report Wizard. This is fine. The Report Builder provides the ground for our work. You will see that all our selections -while using the wizard- will be placed, or otherwise affect, the page on the report builder.

The wizard is composed of two columns. The one to the left names six steps that the wizard will walk us through and highlights the current step. The column to the right is wider and holds the options and stores our selections that correspond to each step. The first thing the wizard wants to know is which table will provide the information and which columns will constitute our report. The drop-down list under "Tables or Queries" displays all the available tables and queries in this database. Find the query we just built (that I named *qryPatientPhoneNumber)* and select it. Quite immediately you will see that the boxes below are populated with the columns that belong to this table. You can select now which columns from this table you want in your report. We want them all so click on the ">>" button. (Notice that the other columns part of the query do not appear as options here. This is because they were not really selected. They were *used* by the WHERE clause to match the info on the tables but were not called by the SELECT command). Click on "Next".

Did you notice how the Report Builder, in the background behind the Wizard, has been automatically populated with the selections we have made so far? Nice!

In the second level, the Wizard wants to know what labels to use to name each field. The wizard uses the names of columns provided by the SQL code that either created the tables used or are defined in the code of our query. Because we took the time to enter informative names that use caps and spaces (and the double quotes they need), the names are actually quite appropriate. However, and just to get a feeling of how this works, change the "Description" label to "Phone type". Then click on "Next". Again, the Report Builder is updated by the Wizard.

In the third step, the Wizard needs to know if we want to use any grouping levels and offers us the names of the columns in our query. Let's explain what this means. Imagine that you have a patient, Stan Smith, that has a home phone number and an office number and a mobile phone and even a pager. When we display this information, we get something like this:

Patient Name	Number	Phone type
Smith, Stan	1234567890	Home
Smith, Stan	4567801239	Office
Smith, Stan	8529630147	Mobile
Smith, Stan	8945612307	Pager
Etc. ...	...	...

...repeating the name of the patient for each phone number she/he has. This is not very satisfactory. Instead we would like to group this information by name. This means that the name is displayed only once, followed by all the phone numbers that belong to him. That is what this step is offering us. To request for this we should select *Patient Name* from the box to the left and transfer it to the box to the right by clicking on the ">" button. You can see that the wizard offers us to transfer more columns if we wanted. Imagine that you have a database of movie and video producers from all over the world. If you were making a report of them you could group them by country, for example, and then add a second layer grouping them by specialty. You can notice some buttons to the right of the right box that display "^" and "v". These allow you to change the order of precedence. If you place the specialty on top of the country then the report would organize your producers first by specialty and then by country, and this is not a minor difference!

However, for our example it is enough that we group our results by patient name only. Select this and click on "Next". If you can peek at the Report Builder in the background you will see that, automatically and following our instructions, it placed the patient name on top of the other two fields. It also created a new section on the structure of the page, indicated by a blue shade at the left margin called "Patient Name Header". This is a header because it will head a section of multiple results. This structure is repeated for every record in the patient table. Later, when we want to create groupings without the help of the wizard, we will be looking to place headers like this. Because now we know what they mean, it will be as simple and more flexible.

Now the Wizard wants to know if we want to give any particular order to the display of our data. Consistent with the SQL code in the query, the “Patient Name” column appears selected in ascending order, that is, alphabetically. This is enough to conform the requirements of the specification we did earlier. But now that we are here we can have a little fun and request that the “Description” field be also ordered. Go to the drop-down box under the “Then by” label and select “Description”. Now click on *Descending order*. This means that Stan Smith's number would appear in the order: Pager, Office, Mobile and Home. Notice that the Wizard gives us up to 4 levels of sorting. The levels also denote a hierarchy. In our example, the data will be ordered by name first and then by description (phone type).

In the fifth step the wizard gives us some options to organize the layout of the information in our report. Of course, the options are not many and in this task the Report Builder will excel, giving us enormous flexibility. At this moment I will select the “In blocks, levels above” option. Notice that the wizard has no options for the footers and headers. That is fine: the Report Builder will help us with this. Below, the wizard asks us what orientation we want to give to the page. Because most of the time the information appears as tables with long rows the default option is “Landscape”, which makes plenty of sense. But in this case we only have three columns and one of them is being used as a header, so the “Portrait” orientation results in a better use of our paper. Select this option. The report builder adjusts immediately and changes the layout of the graphic interface. Don't get startled by this. Now click on “Next”.

In the last step the wizard needs to know three things: i) the name we want to give to this report, ii) whether we want to create a dynamic or a static report and iii) whether we want to keep on working on the layout of the report or if we want to produce it now. For the name of the report, the Wizard will offer us the name of the query used to build it. We could change it to something like: Patient Phone List or whatever you find informative. In any event, if you used my suggestion for the name of the query, erase the “qry” prefix so you don't get confused. A *Static Report* is built with the data in the database at the time the report is created and is not updated when the data is edited. This could be necessary for information that is time sensitive and that you are interested in tracking, allowing you something like taking a photo of the data at a moment in time. In contrast, a *Dynamic Report* acts as a template of a report, rebuilding itself every time it's called and reflecting any editions to the database. This is necessary when you need the reports to always be up to date. At this time select *Dynamic Report*.

We could go to fine tune the report and select the “Modify report layout” option but instead I am curious about the result so far so I am going to ask for the Wizard to create the report now. Do the same.

TADAAAH! Here we are: Our first report! This appears in a viewer as a .rpt document which is Read-Only. Now we have icons that allow us to print it directly or export it as a PDF document. This is very cool!

The information is quite clear. I see that the wizard included a horizontal line to separate the patient name header from the numbers and this helps to make the report easier to browse. I like this. The font chosen is a Sans Serif, which looks modern and is easy to read in a tabular context. Fine too. After initial happiness, however, I begin to notice several things I would like to change. First, we need a header with urgency. Only because I just created this piece of paper I understand what it is but give me a couple of weeks and I will have to scan it for several seconds before I remember what is this report about.

Then, I see than the label “Patient Name” is completely redundant. Actually, all labels are redundant in this report. This is because we don't have many categories and the information is quite self explanatory. Also the boldness of the labels attracts a lot of attention. In reality I would like the patient names to be in bold, so it can help me structure the layout of the page.

Finally, I would also like the distance between phone numbers to be less, as I feel that it makes it more difficult to perceive the structure of the page and also makes me feel that we are wasting too much paper. The font size could also be smaller, allowing more records per page and making the page easier to read. However, I do admit that the actual font size also fills the page quite nicely, providing with an harmonious balance of ink and white.

Of course, this fine-tuning is done with the Report Builder. Let's go. Start by closing the viewer and then go to the Report section of the Base interface. Find your report and right click on it. Then select the option “Edit”. The Report builder opens.

First of all, make sure that you have the Properties panel on (usually found to the right of the screen). Most of the times it appears automatically. If not, you can click on View>Properties or you can click on the second button at the lower level of the toolbar and to the left; the one whose tooltip indicates as “Properties” and that displays several form controls.

Let's start by providing a title to the report. First select the page header and then click on “Label Field” (the button to the right of the Properties button). While on the header, your cursor changes to the shape of a cross and a small box to the right. Use this to click and drag a rectangle. Immediately, the Properties panel displays its properties. This label object only has a *General* tab, but here we find all we need. First, enter into the *Label* attribute the text that you want. I used “Patient Phone Contact Information”. The default font size is not very impressive for a title. Click on the “...” button next to “Font”; this calls the Font dialog box. Leave the defaults but change the size to 22. You will notice that the text is now bigger than the boundaries of the box that holds the text. Left-click on it to reveal the green boxes delimiting the boundaries of the text box and drag them until the entire caption can be read. Actually, drag the left boundary of the box all the way to the left margin of the page and drag the right boundary to the right margin of the page. Now look in the Properties panel for the “Alignment” attribute and select “Center”. This ensures that the title will be centered. Now, just to show you the possibilities, call again for the

“Font” dialog box and select the “Font Effects” tab. Click on the “Underlining” drop-down box and click on “Double”.

Now select the “Patient Name” label and delete it. Actually, delete all the labels. The rectangles that remain are Text Boxes. You can notice that the text box for the patient name field is somewhat to the right of the page. Select it by left-clicking on it and positioning your cursor over the box. The cursor will change to a black cross with arrow heads. You can now drag the box to the left. Use the guide lines that appear to align it with the box underneath it. The Text boxes don't really need to be so long. You can make them smaller by positioning the cursor on top of the green boxes. At that moment, your cursor changes to a white double-headed arrow with which you can change the size of the boxes in any direction. Reorganize them to your own taste.

Let's make sure that the patient name will be displayed in bold. Select this text box and click on the “...” button of the font properties. You can see that you can add any effect that you could like, even color the field. For this exercise, click on the “Font” tab and select “Bold” under typeface. Exit by selecting “OK”.

To arrange the distance between the phone numbers we need to select the *Detail* section under the Patient Name heading. If you check the General tab of the Properties panel after making the selection you will see the Height property, which, in my computer, is set to 0.50”. You can change the height here, by changing the number in the box, or you can drag the gray line that separates the detail from the page footer. Try this now and see how the height data in the Properties box is automatically updated. (Make sure that no other object is selected.)

To make the detail slimmer I found that I also needed to adjust the position of the boxes for number and phone type, moving them closer to the header (I also gave them some indentation by moving them to the right) and finally left the detail with a height of 0.40”.

Let's finish this by including the date the report is created and folios to number the page. Remember that we made this a dynamic form, so we need to update this data each time the report is called. If we used a label to write down the date, we would need to change it every time, which is not helpful. In contrast, the Report Builder offers us an automatic option: For this, activate the page header by clicking on it. Then select on *Insert* in the Menu bar and click on *Date and Time...* . At this moment the Date and Time dialog box appears. The dialog box shows two check boxes so you can decide if you want to include the time, the date or both. Drop-down boxes allow you to chose the format in which you want them displayed. I don't want the time in the report, only the date, so I unchecked the time box. Then I selected the format that I fancied best. Do the same.

After clicking on “OK” a box appeared on the page header at the upper-left margin. Now you can drag it, re-size it and, in general, change any attributes permitted by the Properties dialog box. I moved mine to the right and under the report title. I also added a label to the left of this box wit the caption: “Report created on:”. This way I make obvious the

meaning of the date. Although not completely necessary, it provides a good excuse to practice your new skills. Go ahead and do the same.

Finally, let's add folios to our page. Select *Insert* and then click on *Page Numbers...* . The corresponding dialog box appears. First, select a format that you like (I picked *n of m*). For the position, indicate that you want this on the footer. Finally, select a left alignment and click on "OK". A box will appear on the footer. With this, I believe that our work matches the specifications we previously did for this report quite well!

Let's see our masterpiece in action. Lets first save it, just in case we have a crash (it could happen) by clicking on the blue diskette or clicking on File>Save. Now find a button in the first row of the toolbar with a page and a green arrow pointing to it from the left. This is the Execute Report button. Alternatively, you can find it under Edit>Execute Report or you can activate this function directly with Ctrl+E.

What do you think?

Fell free to experiment and analyze the attributes in the Properties dialog panel for the different objects. Be curious about the menus and the functions of the buttons. Play with them. This is the very best way to learn!

Now that we have the basics. Let's produce our Clinic Summary report.

Steps in designing a report:

In general, when you are producing a report, you should follow more or less the following sequence:

1. Analyze the report's requirements and decide on the overall layout.
2. Select needed tables and columns.
3. Compose query (include functions).
4. Build the report.

If you notice, this is exactly the same procedure we did while developing our first example. Not surprisingly, the first two steps can be carried away with paper and a pencil. In composing the query you need to make sure that you are using the SQL language properly, for which the earlier part of this tutorial should come in handy. Only the fourth step is done sitting in front of the Report Builder. So you see: plan before executing.

Creating a report with Report Builder (that is not in tabular form).

The first thing that we need for our Clinical Summary is to design the layout for our report. I am going to base this report on the draft presented in part II. Note that this report is divided into two sections: Patient Information and Clinical Information. The first section displays information that we are mostly going to find in the Patient's table, so selecting

this should be quite straightforward. The second section is a bit more tricky as it combines information form several tables (seven, actually!). However, this should not be difficult for us. According to our draft, if the case is still active then we will want to write "Open Case" where the "Date of Termination" goes. This should be fun and we will tackle it when we work with the report builder later on. Let's first focus on the challenges facing the SQL code for the query: Note that we have two distinct sections that would require grouping: the phone numbers and the medication, both of which can display unknown and differing amounts of information. Let's focus on these for a moment.

Of course, we want to display all available phone numbers and all corresponding medications, the only condition is that they belong to the selected patient. Note that there are no conditions linking them to each other, they both relate only -and independently- to the selected patient. Because of this independence we are going to find that our result set (the data organized in tabular form selected by our SQL code) will appear as a Cartesian join with a number of rows equal to the number of phone numbers times the number of medications that the particular patient has. Both the wizard and the report builder allow us to nest our groupings but what we really want is two separate and independent groups. This could be solved if we could compose two queries that produce two result sets that we then display in the same page. Unfortunately, neither the wizard nor the report builder allow us to work with more than one result set per report. Until this changes our only solution is to produce two queries that will be printed independently.

I can divide my desired report into two result sets in several ways. For example the "Patient Information" could be one result set and the "Clinical Information" could be the second one. Or I could make all the information in the "Patient Information" and all the information in the "Clinical Information" up to the "medication history" be in one result set and have the "medication history" alone as the second result set. No matter how I divide it, however, this report will always be at least two pages (one for each result set) because each page will be a different report file. I can call each section a "logical page" even if in theory they could be longer than one physical page.

As long as we are being creative, we can think of other possible solutions. One could be to limit the number of phone numbers we display so that we don't need to do a grouping around them and we reserve the grouping for the medication history. However, this would mean losing some information. It could also result in empty records if the patient we want a report from does not have the phone number we have decided to include. We could use a LIKE operator in order to be asked for the number we know beforehand the patient will have, but this will render the production of this report less automatic. We could also think about excluding the phone information from this report completely (in the understanding that the medication history is essential in this report).

As you can see, not being able to include two or more result sets in one report creates some limitations. In the future, if you see yourself facing such a problem you might be considering options similar to those outlined here. In any event, we can think that one beefy report, with several parts that analyze complex relationships of data could be composed of several Report type files, that is, be made up of several logical pages.

In this example I will do one SELECT for the Personal Information and another one for the Clinical Information. You will notice that they are basically identical in form so I will guide you through the first report and you will do the next one as homework. Let's get started.

Analyze the report's requirements and decide on the overall layout.

Overall, the header of the report should include the name of the clinic and maybe a logo. Because this header will appear for every page in the report and because this report displays data related to only one entry (the particular patient) I find that it makes sense that I also include the name of the patient in the header, just in case the report requires more than one page. The footer could have the page number, the date the report was created and maybe a disclaimer about confidentiality and contact data for the clinic. Again, because this report displays data for only one entry -as opposed to most reports that would display data for several records where a tabular format is better suited- I will prefer a portrait orientation for it.

Select needed tables and columns.

The result set for Patient Information will basically include information stored in the Patient Table and the Phone Number table. If you analyze the Clinical Summary report drafted in part II you will see that the data required by the Clinical Information is scattered among the Patient Table, the Assignment Table, the Therapist Table, the MD Table, the Psychiatrist Table, the Medication Table and the Patient/Medication Table. Because we don't want to re-write the code every time we need a particular report, we will use the “Like” operator with the parameters First Name AND Surname.

You should be able to produce the required SQL code. See if yours look like my versions.

Compose query.

SQL for Patient Information:

```
SELECT CONCAT(CONCAT("Patient"."First Name", ' '), "Patient"."Surname")
AS "Patient Name",
            "Patient"."Date of Birth",
            "Patient"."Gender",
            "Patient"."Street and number",
            "Patient"."City",
            "Patient"."State",
            "Patient"."Postal code",
            "Phone Number"."Number",
            "Phone Number"."Description"
FROM "Patient", "Phone Number"
WHERE "Patient"."First Name" LIKE: PatientName
AND "Patient"."Surname" LIKE: PatientSurname
AND "Patient"."ID Number" = "Phone Number"."Patient ID";
```

How was that? Notice how I have changed the format in which I want the patient's name because I want to conform as close as possible to the specification for this report. When you run this query, Base will ask you for the name and surname of the patient you want and then will display all the information that our report asks for. Not bad. Now let's work on the Clinical Information. We know that we will need information from 7 tables and therefore we will need 6 joins. Let's start selecting:

SQL for Clinical Information:

```
SELECT CONCAT( CONCAT( "Patient"."First Name", ' ' ),
"Patient"."Surname" ) AS "Patient Name",
      "Patient"."Time of registry" AS "Date of admission",
      "Patient"."Diagnosis",
      CONCAT(CONCAT("Psychiatrist"."First Name", ' '),
"Psychiatrist"."Surname") AS "Psychiatrist",
      "Psychiatrist"."Phone Number" AS "Psy Phone",
      CONCAT(CONCAT("Medical Doctor"."First Name", ' '), "Medical
Doctor"."Surname") AS "Head Doctor",
      "Medical Doctor"."Phone Number" AS "MD Phone",
      CONCAT(CONCAT("Therapist"."First Name", ' '),
"Therapist"."Surname") AS "Therapist",
      "Assignment"."Date case closed" AS "Date of termination",
      "Medication"."Name" AS "Medication",
      "Patient Medication"."Dosage",
      "Patient Medication"."Start date" AS "Med start date",
      "Patient Medication"."End date" AS "Med end date"
FROM "Patient", "Psychiatrist", "Therapist", "Assignment", "Medical
Doctor", "Medication", "Patient Medication"
WHERE "Patient"."First Name" LIKE: PatientName
AND "Patient"."Surname" LIKE: PatientSurname
AND "Patient"."Psychiatrist ID" = "Psychiatrist"."ID Number"
AND "Patient"."Medical Doctor ID" = "Medical Doctor"."ID Number"
AND "Patient"."ID Number" = "Assignment"."Patient ID"
AND "Assignment"."Therapist ID" = "Therapist"."ID Number"
AND "Patient"."ID Number" = "Patient Medication"."Patient ID"
AND "Patient Medication"."Medication ID" = "Medication"."Medication ID";
```

Wow! Quite long SQL code. Is this what you had arrived to? Here, again, Base will first ask you for the name and surname of the patient for which you want the Clinical Summary and then will produce the report.

We now have the queries for the patient information and the clinical information of our Clinical Summary report. Because reports can include the information from only one result set (SQL query or table) each page will be its own report. When the time comes that the clinic needs a clinical summary on one of its patients, they are going to need to produce both reports (providing the same name and surname each time), print the pages and staple them.

A note on SELECT queries that use intermediate tables: All intermediate tables must be included in the FROM component of a query, even if their columns are not selected for display. This is not obvious in the SQL code of the previous example because we are

selecting DATE information stored in the "Patient Medication" table and therefore it could look like we are including this table for that reason[15].

Now that we have our queries ready, lets create our reports using the Report Builder. This should be step four of our little process of producing a report but because we will go into the SUN report Builder with some depth, we will make it a section of its own.

Using the SUN Report Builder.

In principle, the SUN Report Builder (SRB for short) is all about giving you maximum flexibility when it comes to present your data. Not only can you chose not to organize your information in tabular format (rows and columns) and instead try other layouts but you can also display graphics, collect, process and display information from your result set (the table or query you are basing your report on) according to conditions decided by you and even organize your data in charts of several kinds. Of course, all this flexibility and power means that you need to apply more knowledge. Unfortunately, at the time of the writing of this tutorial the information on how to use the SRB is rather cryptic and sparse. I really hope that this will change in the future. Also, there are bugs in the way the SRB and Base interact which means that things not always happen as planned. Despite these caveats, having and using the Sun Report Builder pays off every time in proportion to the extra effort it requests.

SUN Report Builder overview.

The SRB works with basically the following elements: a) The report layout, organized in sections, b) Labels to print static text like titles or tags and c) Text boxes that will display the data produced by a query or a table. Most of the buttons in the SRB interface are there to help you display, position, format or size these elements.

a) The report layout is organized in sections. Upon opening, the layout offers three sections: Page Header, Detail and Page Footer. The Page Header holds information that will appear at the top of every page in your report. Here you might want to include information like the name of the report, the company that has produced it, the date in which it was produced and other data that frames the validity of the information presented and that you need repeated at the head of each page. The Detail is the area where all the rows collected by your query or that belong to the table used for the report will appear. Here is where you organize the text boxes that represent column data and the labels that tag them. The Page Footer is a section that appears at the bottom of every page where you can add more information about the report, like a disclaimer, the name of the author of the report or the page number, for example. You can adjust how much from your physical page will be taken by these sections by adjusting their lengths. A ruler to the left is offered for measuring. You can see to the left of the ruler that there is a colored area that designates each section. When you click on one to work with it, a white frame appears indicating

15 See the section "Querying more than one table at a time" and the note on intermediate tables in chapter 9.

that the section is active. At the same time the Properties dialog box changes to reflect the options available to the section selected. Try this now.

Should you not want to have a page header and footer you can go to Edit and click on Delete Page Head/ Footer. There are more sections available. Just below this option in the Edit menu you can find the Insert Report Header/ Footer. This header will display information that will only appear in the first page of the report and will not be repeated at the head of the following pages. Likewise, the Report Footer will display information at the very end of the report, in the last page, although not necessarily after the page footer. You can further customize the way these headers and footers appear by modifying the options that appear in the *General* tab of the report's properties dialog box.

By the way, In order to call the report's properties dialog box you must click on the gray area below the report's layout, deselecting whatever section, label or text box was active at that moment.

Finally there are headers and footers available to Groups. We form groups when we ask the report to associate a number of records to one attribute. In our first example we grouped phone numbers around the "Name of Patient" so that the name of the patient appeared only once and, below, we had all the phone numbers that belonged to the named person. We will do something similar in this example (but without the aid of the report wizard). Later on we will see how Group footers can become very useful. To display headers and footers for groups click on View>Sorting and Grouping or click on the tenth icon from the left at the first toolbar row[16], the one that looks like a page with nested lists (you can also use the key command: Ctrl.+G). The Sorting and Grouping dialog box appears, which offers the options to have the header or footer visible, how to sort the results and whether to force the elements together in the same page or display some in the next page if there is overflow.

b) The next element is the Label. The SRB will create many labels for you automatically, particularly those based on the names of the attributes as they have been assigned by the SQL code. Any written tags that you might need are written with labels. To place a label you click on the third icon that appears in the second toolbar row and that displays "**ABC**". The tool-tip for this button reads: "Label Field". Your mouse cursor changes to reflect your selection. Now you can drag and drop a label of any size on any section of your report. The label's Properties dialog box will appear. Do this now and study the properties offered. You can see that labels only have a *General* tab and that the properties are quite self explanatory. With Font you can change size, color and other attributes by clicking on the button with the ellipsis (those three dots).

c) Finally we have the text boxes. They represent where the data for the attribute they are connected to will appear. Mostly, the text boxes will be created automatically by the SRB when you are inserting the fields your report will use. You can select them to change their properties (including dimensions) or relocate them. The other way to introduce a

16 According to the default configuration of the SRB interface. All toolbars can be relocated, left floating or even be closed at your convenience.

text box is by clicking the fourth icon at the second toolbar row, the one that shows the letters "**ABC**" enclosed by a box and whose tool-tip reads "Text Box". After you click on the icon the mouse cursor will change to reflect your selection. Now you can drag and drop the text box wherever you want it. Create a text box now so that you can study it's properties dialog box. You can see that, unlike labels, text boxes have a *Data* tab in addition to the *General* tab. Through the *Data* tab you can associate the text box with a particular field.

Text boxes and labels have a rectangular limit surrounding them that becomes visible when you select them thanks to small green squares. Their presence indicates that the object has been selected, that the current display of the Properties dialog box belongs to such object, that you can relocate it and that you can drag and modify the dimensions of the object. Please note that the SRB will never allow you to overlap two objects. If you need to get them closer, you are going to have to reduce their dimensions. Note that a label or text box too small could result in partial display of your information or even its complete omission from the final result.

Building a report with the SRB.

Now that we know the basics of the SRB, let's build our report. For this example we are going to produce the Clinical Information report.

Typically, when you open the SRB a small box with names in it pops out. This is the Add Field dialog box and the names included belong to the attributes/columns of the selected query or table.

Start by right-clicking once on the gray area bellow the report's working area to make sure you deselect any object and can access the report's properties dialog box. Go to the *Data* tab and make sure that the "Content Type" states "Query". Bellow it make sure that the "Content" box has the name of the query you gave to the Clinical Information. If you click on the box, a drop down menu will offer you all the queries you have composed for this database. Select the one we need. After you hit Enter or take focus out of this box, the Add Field dialog box will display the name of all the attributes selected by our SQL code.

For this exercise we will only need the page header and footer and the detail area. Make sure no other sections are present.

According to the design decisions we made earlier for this report, we want the name of the patient to be repeated in every page in the improbable but plausible case that a long history of medication use requires the printing of more than one page. This means that we are going to want the name of the patient to appear in the page header of this report. First, click on the Page Header section. You will see that it becomes selected and the properties dialog box changes. Now either right click twice on "Patient Name" in the Add Filed dialog box OR click just once and then click on the "Insert" button once it becomes highlighted. You can see that the SRB places a label and a text box in the upper margin of the

header section for the Patient's name. I recommend that you now relocate it to the bottom of this area. If you place the mouse cursor inside of the area defined by the green squares it changes to a cross with arrow heads. You can now click and drag it with the mouse or move it with the arrow keys in your keyboard. Notice horizontal and vertical guide lines that help you position these objects.

If you deselect these objects and select them again, only one of the pair becomes selected, either the label or the text box, depending on which one you made the click. Now you can access its properties and modify it to your hearts content.

If you want to re-select them as a group, you first select one by right clicking on it and then click on the second while holding down the Shift key in your keyboard.

Select the label. In the dialog properties box you can change the content in the second row (appropriately called “Label”) The caption is quite informative but I want to add a colon. Do this now and hit Enter. You will see the label update in the report. Now click on the ellipsis button of the Font property and change the typeface to bold.

You can see that the text box where the name will appear is quite distant from the colon. I want them to be closer. Position your mouse cursor on top of the right green square. This time the cursor changes to a horizontal line with arrow heads. Now you can click and drag a new width for this label. Notice how the Width attribute in the properties dialog box changes correspondingly.

Now click on the text box for the patient's name. Drag it closer to the label and then change the typeface to bold as well. Also, take some time to study the *Data* tab for a moment. You can see that only two of four boxes appear active at the moment. The first one states that the Data Field Type is either a Filed or a Formula. In this case it is exactly a field. Further down the Data Field correctly connects this box to the *Patient Name* field.

Now that we are working on the header, lets add a title to the report. Select a label and position it above the Patient Name. Make it read: “Clinical Information” give it a font size of 22 and a bold typeface. If necessary, change the dimensions of the surrounding box (defined by the green squares) to fit your text.

You have several options if you want to center this caption. First you could expand the surrounding box's width to go from margin to margin and then, in the properties dialog box, change the alignment property to “centered”. Or you could just click on the second button of the Align toolbar, the one with a blue rectangle (with blue guide lines) flanked by two opposing orange arrows whose tool-tip reads “Center”. Notice that this will appropriately center the surrounding box (not necessarily the text) so make sure that there is no asymmetric space between the text and the margins of the box, or the text will seem off-center.

We are now going to place the remaining information. What we want to achieve is to have all the clinical information, including diagnosis, head medical doctor, psychiatrist

and the rest under the patient's name and, at the bottom, list the medication history. If we place all these in the detail, we are going to have the big block of information repeated for every medication in the record. This is not what we want. Clearly, we need to make a grouping here.

Call the Sorting and Grouping dialog box. Because this report will produce information for only one patient at the time you can chose any field for grouping that is not part of the medical history. In the Groups box, under Filed/Expression click on the list box and chose -say... Diagnosis. When doing so, a new section will appear for the report called "Diagnosis Header" with a nice blue shade in the left margin. Here we will place all of our grouped elements. Click this area to select it.

We will follow the design we made in part II for this report. You can start by adding a label that will read "Clinical Information:" with the colon and all. Leave it to the top of the section, make it bold and give it a font size of 10. Below this insert the "Date of Admission" field. Note how the SRB places it automatically under the previous label. Now insert "Date of Termination" and position this pair to the right of "Date of Admission". Notice how the guide lines aid you to place them neatly.

Insert "Therapist", "Diagnosis", "Head Doctor", "MD phone", "Psychiatrist" and "Psy phone", following our design. If you need extra space in the group header just position the cursor over the lower limit and when it changes to vertical arrows drag the limit down. Notice that the SRB will also enlarge the area to fit new elements.

Finally, insert a label at the bottom of the Group Header the reads "Medication:". Change the font by making it bold. We also don't want to repeat this label more than once, which is why we are placing it here. However, it will serve here as a header for the detail section.

Now we will place all the elements for the medication history in the detail area. Select the detail and insert "Medication". Select the label and erase it because we will not need it. Now, flush the text box to the left margin. Following, insert Dosage. Again, delete the label. Position this to the right of the Medication text box. Now insert "Med start date". Change the label to "From:" and adjust dimensions. Place this pair to the right of Dosage. Finally insert "Med end date" and change the label to "Until:". Finally, place the pair to the right of Med start date on the same horizontal line.

What you should have now is one horizontal line in the detail area with all the medication information. Now drag the lower limit of the detail area up until there is very little space between it and the boundaries of the labels and text boxes. Don't get startled if some gray area appears: that is fine.

At this moment you better save your work.

Before running the report you might want to add some elements like a disclaimer at the page footer (with a label), page numbers and current date (as we did in our first example)

and maybe insert a graphic as a logo. You will find most of these options in the INSERT menu. Do this to get a hang of using the SRB.

Now click on the eleventh button in the first toolbar row, the one that looks like a report with a green arrow in it. You should be asked for the name and surname of the patient you want the report for and then the clinical information should appear, ready for you to print or save as pdf. Just make sure there is data in the tables to appear in this report.

Maybe the spacing between lines and sections is not what you thought you were getting. Practice moving the boundaries for each section, and even reposition your page elements, until you get a layout that doesn't disconcert you and seems natural.

How is it going? Let's take this a notch further. There are two text boxes of interest here: The one to the right of the label: “Date of termination:” in the grouping header and the one to the right of “Until:” in the detail. These boxes will display data if the patient's case is closed and medication regime has been terminated, respectively. In real life, if the patient's case is still active then the “Date of termination” cell in the table will be empty. The same thing if the patient is still being administered any medication. Consequently there would be no data displayed in the report. In order to dismiss any ambiguity, we would like the report to print something different, like: “Case Open” or “Still Active”. This could be managed by altering the SQL code that created the query or by modifying a table if we are working directly with one. Hum! This could be quite cumbersome and we don't really need to: We can have the SRB take care of this for us. To accomplish this we use formulas and functions.

However, before you get to this, consolidate what you are learning and produce the Patient Information report.

Using formulas and functions with SRB.

The SUN Report Builder allows us to introduce formulas or functions with which we can collect data from some of the text boxes, process it and achieve a new result or piece of information, very much like we did with functions in SQL. For example, we can calculate sums of numbers, averages, replace text in strings or change case, we can ask for text to be inserted when SRB finds a blank cell and a long lists of etceteras.

The only difference is that this time we will not be talking in SQL to the HSQL engine. Instead, we will be talking directly to the SRB in its own lingo. Don't panic because you only need to understand some simple conventions and later let the very friendly Function Wizard take care of the rest.

The first thing you need to know is how to denote a field value. This is basically the name of a column in the table or query that we are working with, either belonging to the table or established by the use of an alias in the query. When we work with HSQL we identify these with double quotes. When working with the SRB we identify them by enclosing them within square brackets (“[“ and “]”).

In order to identify text strings we will now use double quotation marks. If you remember, when working with HSQL we used single quotation marks. But that was then.

Finally, composing a function or formula is basically about connecting field names and maybe string literals with SRB's built-in functions.

Using Formulas.

You can get a complete overview of the available built-in functions for the SRB here:

http://wiki.services.openoffice.org/wiki/Base/Reports/Functions

If you check carefully, you will find many functions you already know about: Mathematical functions (e.g. the calculation of averages), date and time functions, (e.g. the calculation of date differences), string manipulation and others. You will also find other types of functions like logical functions, that allow us to make decisions in the way the function behaves, and metadata functions that allow us to introduce info about the author or title of the report. Do take a look.

Furthering the example we have been working, we want the Clinical Information report to write “Open Case” if there is no termination date for a particular patient.

Let's break this down: If the field with the termination date in the report is empty then write “Open Case”. Otherwise, just write the date in which the case was terminated.

If you check the documentation, you will find two very useful functions: IF and ISBLANK.

IF is defined in the following way: *IF(test; true_value; false_value)*. This means that an IF function takes three parameters: A condition that is evaluated, a value to return if the condition proves true and a value to return if the condition proves false. Note that each of the parameters are separated by a semicolon and that the entire expression is enclosed within parentheses[17].

The ISBLANK function takes only one parameter: A name of a field; and results true if the field is empty and false if not.

This is all we need really! To compose our formula we just need the name of the field as known by the query. If we check the SQL query we see that this field is called “Date of termination”. So we write:

```
IF(ISBLANK([Date of termination];"Open Case";[Date of termination])
```

17 The documentation also states that the test parameter is optional and that only the return values are needed. But this example will use the three.

This formula captures our logic with no problem: If “Date of termination” is empty then return the value 'Open Case'. Otherwise return the value in “Date of termination”. Notice how the string is enclosed by double quotation marks and the field name is enclosed within square brackets. Also, notice that you don't need to leave white space between parameters, unlike written English.

Let's now go back to our report in edit mode. Select the text box that displays the “Date of termination” data. The properties dialog box will change, offering you the box's *Data* tab. Click on it.

This tab offers only four boxes and two of them are not active at the time. That is just fine because we will not need them for now. The first box is called “Date field type” and specifies what type of origin determines the content of it. By default the box states: “Field or Formula”. This is correct: at this time, the content of this box is determined by a Field. Other options include Functions and User Defined Functions. We will use these later. The second box is called: “Data field” and specifies the origin of the content. In this case it says: Date of termination, which is the name of the column of the query we are working with. So the content of this box come from this column.

Here is where we want to introduce our formula. We don't need to change the data field type (the first box) because it already allows for a formula. We just need to replace the content of the data field (second box) with our formula. You can now place your cursor there, erase the content and type the formula we have created.

OR you could click on the ellipsis and call for the Function Wizard and let the wizard take care of the syntax and spelling and set the formula for you. This wizard is really cool. First, it's a library of all available functions, explaining what they do and giving their formal definitions. Then, it allows you to build your formula just by point and click, making sure you include all the proper parameters and taking care of syntax and spelling for you. What more can you ask for? A body rub? Given that we know the form our formula must take, let's compose it with the wizard to learn to work with it.

Star by clicking on the button with those three little dots...

The Function wizard interface pops-up showing two text boxes, one vertical to the left and one horizontal to the bottom. To the right and above there is an area that will change depending on the options that you select. The box to the left shows two tabs: Functions and Structure. Chances are that the Functions tab is selected at the moment. If not, select it now. To the top in this tab there is a drop down list with the name Category. The wizard organizes all available built-in functions in categories so that they are easier to find. You can click on the button with the arrow to peek on the options. Notice that there is an option called “Last Used”, which suggests that the wizard takes notes of our actions and helps us repeat them with ease. You can see that the options include mathematical and logic functions, date and time manipulation and even user defined formulas. However, because at this moment we are not familiar with them, make sure that the option “All” is selected. This way, all functions will appear in the lower section labeled Function.

The lower box has the tag “Formula”. At this moment it should show an equality (or assignment) sign “=” and the name of the field in use: “Date of termination”. On top of this box you will see the function currently selected in the Function box defined and with a short explanation of what it does. Chances are that the ABS function is selected (highlighted) to the left and explained now. You can read that this function accepts one parameter (a number) and the explanation that this function returns the absolute value of the number.

Place your cursor over the function box to the left and scroll until you find the function IF. Note that the functions are organized alphabetically. When you find it, right-click on it once to select it. Note how the explanation now provides the definition of the function (similar to the one we gave earlier) and a short explanation. Double right-click on it to select it.

Several things happen now: The category box changes to Logic and the number of available functions shrinks. The Formula box now includes the text “IF()” and there is a blinking cursor between the parentheses. If the name of the field was highlighted before then it has been replaced by the new text. If not then it remains at the tail of our insert.

However, the most amazing transformation happened in the explanation area because we now have three text boxes flanked by buttons. These boxes correspond to the three parameters that this function uses: the test, the true value and the false value. Notice that the word “Test” is the only one without a bold typeface to suggest that it is not a mandatory parameter. The buttons to the left seem to have a calligraphic 'Fx', meaning functions. The buttons to the right show a table with a green arrow that points to one of it cells. The first button allows you to select and insert a function. The second button allows you to select and insert the name of a field.

The test in our formula asks us to evaluate if the field “Date of termination” is empty or not. We will be using the ISBLANK function for this. Consequently, click on the Fx button for the Test text box. Doing so takes you back to the “All” category listing. Scroll down until you find the function we are looking for and double click on it. At this moment the function info pops up and a text box for selecting the value appears. This time click on the button to the left (Field Button). Now, the Add Field dialog box appears, with all the fields present in our query. Look for the “Date of Termination” field and either select it and click on the Insert button or double click on it. Now the Formula box displays the name of the selected field. Note that the square brackets, that precisely indicate that this is a field name, are automatically included.

To continue you now need to set the cursor in the formula box after the first closing parenthesis. This way, the wizard brings us back to the IF page so we can populate the two remaining boxes. Note that our compound test, that includes a function and the name of a field, correctly appears in the Test box.

In the second box we must indicate the value that the function must return if the test results true. In this case we want the string 'Open Case'. We will not find this value among

either the functions or field names. What we do is type it. Because this is a text string value, we must enclose it within double quotes. Type this directly to the box now. Notice that when you click on the second box to insert the string a semicolon is added automatically in the formula by the wizard in order to separate the first and second parameters.

Now, for the third box, click on the button that calls the Add Field box. Again a semicolon is added to separate parameters while the dialog box waits for your selection. For the second time, select and insert the "Date of Termination" field name.

If you check now, the formula in the Formula box is identical to the formula we had presented earlier, just that this time it took very little typing from us. If by any chance the field name that was present before we built our formula is still trailing, erase it now.

At this moment you could click on the Structure tab and get a glimpse of what it offers: basically, you get a more graphical and structured rendition of the functions and field names that your formula uses. Nice!

Finally, click on OK to insert our formula in the "Data field" box of the Data tab for the corresponding text box. You can see it there now but you might need to click on the arrow button if you need to read it whole.

Okay, enough dilation. Let's check the effect right way. Click on the Run Query button and insert the name and surname of an entry in your database that does not have termination data. The report should come out stating "Open Case".

Hey! Isn't this great?! Test this to your hearts content. With the help and ease offered by the function wizard it's very easy to compose and insert functions that enhance the value of the information in our reports.

Now do the same with the field in the detail section that records the date when certain medication was terminated. Use the string "Still Active" to give the report more variety. We will use this in our final example.

Using Functions.

Now that we have some experience using formulas we are going to take this a step further and use some functions that come preloaded with the SRB. There are several circumstances when a report would require that some information in the report be processed and be presented as part of the report. For example we could want to add the totals in a series of numbers (e.g.: sales in a region or per salesperson), find the highest number (Salesperson with most number of sales) or the lowest number in a column (Department with fewest number of complaints). SRB offers these exact functions for us to include in our reports. Of course, you know by now that you can easily produce SQL queries with this type of information and even other more complex ones. But you also know that combining aggregate and regular functions might require more than one data set and therefore, more than one logical page (that is, multiple reports). Your exploration of the function

wizard should have shown you that many of those functions can be replicated by the SRB without the problem of multiple data sets. Even better, the process of finding the sum, min or max of a column of numbers has been further simplified. This is what we are going to test now.

One of the goals of the database for the psychotherapy clinic is to learn about the money in and the money out of their operations and several of our forms require that we sum the money received and the money that must be payed. We are going to do a version of the Clinic's Summary of Payments report. We might simplify this report somewhat just so that we can focus on the process of designing a form that uses the sum function. Later on you should attempt to replicate the form as described in part II.

I will now briefly go through the steps for designing a report as detailed earlier:

This report will list all patients that made a payment within a time frame, the date and the amount of the payment. The report will ask for the time window and then list alphabetically all patients that made a payment in that time slice, the amount they payed and the timestamp of payment. We also want the sum of the Patient's total after displaying the detail. We will also want the report to display the total amount received in that period from all patients previously listed, information that we will place at the end of the report. So we will have subtotals (by patient) and an grand total (all payments together).

The data needed comes from the Patient table and the Payments table. The SQL code could look like:

```
SELECT CONCAT(CONCAT("Patient"."Surname", ', '), "Patient"."First
Name") AS "Patient Name",
"Payment"."Date and time of Payment",
"Payment"."Amount"
FROM "Patient", "Payment"
WHERE "Patient"."ID Number" = "Payment"."Patient ID"
AND "Payment"."Result" = 'CREDIT'
AND "Payment"."Date and time of Payment" >=:BottomTime
AND "Payment"."Date and time of Payment" <=:TopTime
ORDER BY "Patient Name" ASC;
```

Save this with a name you can easily identify. Notice that the query will only select the payments that were successful ("Result" = 'CREDIT'). You could later produce a report with all the failed payment attempts ("Result" = 'DEBIT') and include the Memo field that explains what happened.

This query will produce a table with only three fields: Patient Name, Timestamp of payment and the amount. Make sure that you have created the forms for payment[18] or otherwise populated that Payment table so that you have actual data to play with. Include data from at least two different months so you can test the time window function and also include both CREDIT and DEBIT transactions so that you can check the SQL code in its full splendor.

18 You will need a form with patient name and surname and a sub-form for the rest. Let the wizard help you.

You might also have advanced that we are going to be doing grouping of data around patient name. Groupings do have headers and footers. The header will be a good place to insert the name of the patient. The footer appears at the end of the section and therefore is the logical section to place the column's sum.

The total sum of all payments needs to go at the end of the report. The only section that can be placed at the end of a report is the Report footer. The page footer is not the place to use because this footer is repeated at the end of every page and therefore does not constitute the ending of a logical section but of a layout section (the end of the page). On understanding these conventions it becomes obvious that we are going to need to include the Report header and footer and the Group footer, none of which is on by default.

With this in mind let's start our work: go to Report and call the SRB. Go to the *Data* tab, select Query for the Content type and then select the name you gave to our query. Now go to the Edit menu and click on Insert Report Header/Footer.

Notice that the report's header and footer appear AFTER the Page header and BEFORE the page Footer. This is not a mistake. The page header will appear on the head of every page by default, including the section for the Report header. Even if your Report header takes a full page of explanation and other boilerplate text, the top of the page will have the Page header. The same thing goes for the page footer. Anyway, you know that you can change this behavior in the *General* tab for the report.

However, don't change the default for this exercise.

In the Page header write a title for the report. Something like “Clinic Payment Summary”. We are using the page header for this because we want this title printed in all extra pages the report could generate. Adjust the margin of this section to something you feel as appropriate. Make sure the boundaries of the label are not preventing you from downsizing this section as much as you want.

I do not really need to use the Report Header (but I do need the Report Footer) for this exercise. Anyway, I'm going to place here a caption that reads: “For payments made between … and ...” and will insert there the *BottomTime* and *TopTime* variables (which the Add Field dialog box kindly displays for us). Because I am using the Report Header, this info will only appear in the first page of the final report. Let's do this. We are going to use two labels: One for the part of the text until the first ellipsis ans the other for the “... and ...” bit. Click on label and write the first part. Then insert *BottomTime*, erase the tag and place the box next to your text. It is very convenient if you place both against the upper margin because we are going to reduce the size of the Report Header's section. Now insert a label that just reads “and” and finally insert the *TopTime* (also discarding the label). Now drag the bottom limit of this section up, leaving only a small margin before the next section. Don't worry if gray area starts showing underneath.

Now we are going to work on the Detail. We already identified our need to use groups and organize them around the name. Call for the Sorting and Grouping dialog box and

click on the box with a down arrowhead under Field/Expression. Now select the Patient Name field. At this moment the Group Header pops up. If you read under properties, the third option states: “Group Footer: Not present”. Change that to “Present”. Now the group footer pops up.

At this moment you have the detail section surrounded by the group header on the top and the group footer at the bottom. Just like in previous examples, we will place information that identifies the group in the group header. In this case we will write the name of the patient here. In the detail area we will place the timestamp and the amount payed. In the group footer we will place the sum of the amounts payed by the particular patient. Notice that the detail will be repeated for every amount payed by the patient in the time frame and the whole group will be repeated for every patient that was recorded making a payment within the time frame.

Select the Group Header and use the Add Field box to insert the patient's name there. My sense of order doesn't need the tag so I am going to erase it. I will change the font properties for the text box to bold typeface though. This should be enough to frame the beginning of the group. I also recommend that you flush the box to the left margin. Now you can insert a label one row further down and to the right of the Patient's name text box. This is going to be like the header of a table that will list the date and amount of payments. To conform a bit closer to our original report draft you can give it the text: “Payment received on:”. To the right of this label add a second one with the text “Amount”. To make them look like headers change their typefaces to bold. Now you can drag the bottom margin up. If you have deselected the text box you should see the section's property dialog (select on it or the bluish area to the left to make it active). You can see the Height property change as you drag the bottom margin. I left mine at 0.50”. See what you like.

Now select the detail and insert the “Date and time of Payment” field. Erase the tag and place the text box under the first label you created. Now insert the amount field, erase the label and place the text box under the “Amount” label. Notice that while the labels are in the group header, the text boxes for the actual values are located in the detail area. Now drag the bottom margin until your sense of aesthetic is pleased (I got to 0.22”). Placing margins in pleasant ways might take some practice.

Now comes the fun part: Select the group footer and place a text box there aligned with “Amount”, making a column with it and the other text box we placed in the detail section. Here is where we will display the sum of all the amounts in the group. Now go to its *Data* tab. The Data Field Type reads Field or Formula. Change it to Function. Now for Data Field select Amount. At this moment the third box, which reads Function, becomes activated. Click on it and select Accumulation. This combination will produce the sum of the values in Amount. But we don't want all of the amounts. Only the ones for this group. Don't despair. Base identified this quite quickly upon noticing that we are placing this text box in the group footer. So the last box, descriptively named Scope, automatically displays: “Group: Patient Name”, which is exactly the scope that we want. The other option for scope is Report, which we will be using shortly.

To finish the Group footer we are going to add some text that helps frame the info here. From the Add Field box insert Patient Name (again, I know) and make sure it falls to the left of the accumulated amount text box. Now change the the text in the label from “Patient Name” to “Total payments for:” and adjust the width of the box and the distance with the text box so that it looks like a sentence “Total payments for such and such:”. To separate a group from another (in this case, a patient block from the next one) Leave more white between the lower limits of the label and text boxes and the lower margin of this section. Because I am using the Arial front, size 12pt. and the boxes are flushed against the top margin, a height of 0.30” works for me. This is only a reference and you can do what feels better to you.

Now select the Report Footer. Remember: this section appears only once, at the end of the report. Here we are going to place the total sum of all payments received. Start by placing some labels and the time frame variables so that the caption in the report footer reads something like: “Clinic's total income for period between *BottomTime* and *TopTime*:”. If you are not sure how to do this follow the instructions concerning the Report Header in this exercise. Now Insert a text box to the right of the previous caption. Select the *Data* tab. On the Data field type select Function. On the Data field select Amount and on the Function option select Accumulation. Now the report will automatically select “Report” for the Scope of the function.

This is a beautiful report. Make sure to save it now (you never know) and run it: First you are asked for a time frame. Start by providing the first day of the earliest month you inserted in your data and the last day of the latest month. This way all the successful (“Result”='CREDIT') payments that you entered will be included. Now Base produces your report. It could be the case that some boxes are too short to display all data, or too long; or that section margins need to be further adjusted or that you need to tweak the alignment of text in the boxes. SRB gives you all the flexibility that you need to fix this. But for now focus on the totals by patient and the grand total calculated by your report. Realize that we don't need to do any more programming to request income information for any period of time that we have recorded in our database; and that our time windows can be of any length and precise to a second. Confirm this: run the report again, now giving different time windows. Notice that you can omit time (as opposed to date) parameters to make your input quicker.

Ok, enough. You can dry the tears in your eyes now. You should practice your new skills and produce a report that displays the highest payment by patient and the highest payment overall (use the max function) and then one that displays the lowest payment by patient and overall lowest payment.

These functions where placed automatically by Base. We didn't even have to worry about writing the formulas needed for these calculations. However, it would be very informative for us to see how the experts have created the formulas that work with these functions. Let's examine under the hood and see how this marvel happened so we can replicate it with other functions. To do this we need to become familiar with the Report Navigator.

The Report Navigator.

The Report Navigator provides a summary of all the elements you have placed in your report: Labels, text boxes and functions by section (header, footer, detail, etc.), gives you quick access to their Property Dialog boxes and activates any element in the report if you click on its name in the Navigator. This is not unlike the Form Navigator we have studied earlier.

You call this window by clicking on the button next to the ones that call for the Add Field and the Sorting and Grouping boxes: it's the one that looks like a form with a compass over it.

If you call this window now you can see that it displays some kind of tree structure with a root in Report and, dependent of it, the elements: Functions, Page Header, Report Header, Groups, Detail, Page Footer and Report Footer. This pretty much describes the content of the sections in our report. You can also see a small box to the left of these elements with a plus sign. If you click on it then the contents of the sections become visible. Meanwhile, the little box now displays a minus sign instead. If you click on it, you hide the contents. For example, if you click on the Page Header, the Report Navigator shows that the only content there is a label, displaying also the content we gave to it. If you click on this section, the label in the report becomes active and the properties that belong to it are displayed in the Properties dialog box. This is an easy way to find any element in a report.

Expand all the boxes to see a complete description of the contents of our report. Adjust the margins of the Report Navigator window if necessary.

Everything we placed in the report is described here except for one element that might look quite novel to you: Functions. And there are two:One just under the Report root and other associated with Group. This is not so strange because we do have two functions: One that calculates the sum of money amount by patient and another that gives us the grand total. If you think that the function that calculates the partial results must be the one associated with Group, you are right. The function is defined at the head of the Group by Patient Name although its used at the footer. Because it's defined at the very beginning of the group section, it will consider data that is only local to the group and we could use it anywhere (in our case, we use it at the footer) but only in the group section. It also means that this function is local to the group and we might not be able to use it outside of the group.

With the previous in mind then it's easy to understand that the function defined at the very beginning of the report, which is used at the Report Footer, could be used globally, that is, anywhere, in the report; and that it will consider data available to all the report. Hence its ability to calculate the total sum of the amount field.

So now we know that in the future, when we want to include a function that can be used anywhere in the report or that uses data from the entire report, we want to define it at the beginning of the report. When we want functions that make calculations local to a group, we will define it within the group section.

Now note that the functions have been given names: The global function is called: *Accumulation Amount Report*. The local function has been called *AccumulationAmountPatient Name*. Base has assigned the names of the functions by joining the name of the function option, the field it bases its calculation on and the scope of its domain. The relevant thing is that functions are given names although we don't need to follow the same convention.

If you expand the Group Footer section you will see that the Report Navigator describes the following contents (and not necessarily in a left-to-right and up-to-down order but organized by the order in which they were created): There is a Label that holds the text: "Total payments for:" Then there is a text box with the field corresponding to Patient Name and finally a Formatted text box (the one we placed manually) associated to the function name: *AccumulationAmountPatient Name*. Now, note that this function is placed within bracket parentheses. By making this association (which Base made automatically this time for us) we manage that a box display the result of a function.

If you now select one of the functions in this report by clicking on the name of the function in the report navigator, right next to the Fx symbol, you will see the Properties Dialog Box of the function. Do this now. You can see that a function has only a *General* tab and only five boxes with properties: Name, Formula, Initial Value, Deep Traversing and Pre-evaluation. We are mostly going to work with the first three boxes: Name, Formula and Initial Value. The name allows you to identify your formula which allows you to later use it with a formatted field and even with other formulas. The second box, Formula, tells Base what calculations to perform exactly. You can see a button with an ellipsis. Certainly, it calls for the Function wizard. The documentation states that this field accepts and understands formulas as defined by the OpenFormula standard. The initial value box can hold a field, another formula or a constant, depending on what we want to achieve. We will see some examples of this[19].

Let's start with the formula for *AccumulationAmountPatient Name*. In the Report Navigator expand the Fx box below Groups and Patient Name and then click on *AccumulationAmountPatient Name*. Now check the Name, Formula and Initial Value boxes in the *General* tab of the Properties dialog box. They should look like this:

```
Name:  AccumulationAmountPatient Name
Formula: [Amount] + [ AccumulationAmountPatient Name]
Initial Value: [Amount]
```

The first thing we can notice is that this formula uses an addition sign ("+"). In effect, formulas allow us to use mathematical operators like: "+" (addition), "–" (subtraction), "*" (multiplication), "/" (division) and parentheses for grouping "(" and ")".

The second thing is that this formula has been given a (quite long) name but later is treated as a field, that is, its value is referenced by using the name withing square brackets "[" and "]". This is a very important feature. It basically means that the function itself becomes something like a variable, a name that keeps a value.

19 Just for the record, if you need the formula evaluated before the report is run then you must set Pre-evaluation to Yes.

Let's see how this works: the function definition starts by allocating a name to identify the variable (first box). Then is states that it will start by assigning to it the first value found in the field (column) "Amount" (third box). Finally it states that every new amount will be added to the previous value of the function and stored in the function (the Formula, or second, box). Because this function has been declared inside the Patient Name Group, it will reset every time the Patient Name field changes.

Go to the Report Navigator and click on the *AccumulationAmountReport* function and see if you can understand it's definition and the way it has been set.

Following this insight, I could try the following definition for a function:

```
Name:  numberOfItems
Formula: [numberOfItems] + 1
Initial Value: 1
```

This proposes to create a function called 'numberOfItems', that will start receiving the value of one and will increase by one every time a new field is evaluated. This allows me to keep a count of the number of records in a field for the scope of a group or a complete report, depending on my selection of the scope of this function (which is done in the last box). This is exactly what the Count function does, which you can find in the Data Field Type box of the *Data* tab of any Text Box.

But formulas don't need to limit themselves to mathematical operators. They can in fact use any of the functions available to the SRB. The Min and Max formulas show this clearly:

```
Name:  Min
Formula: IF([FieldValue]<[Min];[FieldValue];[Min])
Initial Value: [FieldValue]


Name:  Max
Formula: IF([FieldValue]>[Max];[FieldValue];[Max])
Initial Value: [FieldValue]
```

You can see that the definitions for the Min and Max functions use the IF function. Let's interpret the Max function: *Assign to "Max" the number stored in the first "FieldValue" evaluated. Compare the next value for "FieldValue" to the number stored in "Max". If the number is bigger, store it in Max, replacing the old value. If not, keep the old value.*

With this information we should be perfectly able to venture in the creation of our own functions. Let's try this next.

Custom made functions.

Now that we understand how to use the Report Navigator and where and how functions are defined, we are going to start from scratch and create a custom function that calculates the average payment by patient and the overall average of payments. This way we can use the same query we created for the previous exercise. Also, the general layout of the form that we need is basically the same to the one we just created; only the function definitions will be different. Anyway, my recommendation is that you don't just try to reuse this one: Build the report from scratch and practice what you have learned. Take your time. I will sit here and wait for you.

As you know by now, the SRB allows us to create our own functions (called User Defined Functions) at two levels: the grouping level and the report level. In this exercise we are going to create a function that calculates the average payment by patient and the overall average payment for a defined period of time. Because we are using the elements created in our previous exercise, most of the functionality required by this report has been explained already (the SQL code that requests the time period and filters the information, the grouping of the information by patient, the organization of the report in report and grouping sections, etc.). Now we will just focus on creating the functions and making sure that they are used by the report.

We know that we need two user-created functions: One to calculate the average amount payed by each patient and another to calculate the overall average. The first one needs to be placed at the grouping level and the other one needs to be placed at the report level.

This should be a fun exercise because we can't use the AVERAGE formula that appears in the Formula Wizard here. How come? If you analyze its definition you will read that this built-in function returns the average of a sample and that allows for the input of up to 30 elements that can be either column names -present in your table or SQL code basis for the report- or other formulas. However, this function does not allow you to calculate the average of values inside one column but rather uses values across several columns, usually in the same row. In general, all functions work with values in one row, let's say, horizontally; and not vertically with values in one column. If your SQL code or the table you are using as basis for the report has, say, a column for first quarter total sales and a second column for second quarter total sales and so on (up to thirty columns) then the function will happily return their averages, row by row.

Wait a minute! you could exclaim: Were we not calculating the sum of values in a column (and the biggest or smallest value in a column) just in the last section? Yes, we were. Thanks to the fact that the name of a function works like a variable that stores a value, we were storing there the accumulation of a column and the bigger or smaller number in a comparison. BUT, we were doing this row by row: *We land in a new row? Then add the value to the number we are keeping or compare the values and keep the larger one, etc.*. In the case of the accumulation, this is made possible because the sum of a number to two numbers previously added is equal to the sum of the three numbers, that is:

```
(a+b) + c = a + (b+c) = a + b + c
```

However, this is not true for averages: The average of three numbers is not equal to the average between one of the numbers and the average of the other two:

```
AVG(a,b,c) ≠ AVG( AVG(a,b), c)
```

For example, the average of 4, 5 and 6 is 5. But the average of 4 and 5 is 4.5 and the average of 4.5 and 6 is 5.25. Oops!

Is this the end of the world? Not really. We have the goal to build our own functions and this exercise demands just that. Let's start by understanding what we need: We want to calculate the average of a set of numerical values. How do we do this? We add the values and we divide this result by the number of values in our set. We will not know before hand how many values come in our set. Then, let's count the values as we add them. These two functions rely on addition, which can be done row by row with no problem. This way we know that we will need three functions in total: one to calculate the sum of the numerical values, one the calculate the number of values in the set and one to calculate the division of these results.

To make things even more pleasant for us we don't even need to produce the first two functions: we can use the Accumulation function we analyzed in the previous section to calculate our sum and we can use the Count function to determine the number of numerical values. We just need to put them in place and then create a function that calculates their ratio. This is what we are going to do now.

We are going to start with the Group average. In order to illustrate as clearly as possible what we are doing, we will take some intermediate steps you might want to replicate when you attempt to do the report average.

Let's start by adding a Count. Because we have not done this before, I want to show you how it works. Let's start by adding a small text box to the right of the text box that displays the “Date and time of Payment” field in the detail section of this report. Then access the *Data* tab of this small new box. In the Data Field Type select Counter. At this moment Base deactivates the following two boxes in the *Data* tab and correctly sets the scope to Group: Patient Name. That's all it takes as Base has taken care of the rest for us. If you run this query now you will see that each payment per patient has an ordinal number that starts with one for each new patient name.

The Group footer should have three elements: a label that frames the info and should say “Average payment for”, a text box where the patient name field will be displayed and a text box where we will display this average. I want you to add a fourth element: another text box this time to the right of the last one. We will display our sum here. After creating the box go to its *Data* tab and select the Function option for the Data Field Type. Then in Data Field select “Amount” and select Accumulation for the Function. Yes, we have done this before. In any event, if you run the query now you will see a sum of the amounts payed by each patient.

At this moment we have the number of payments and the total sum of payment made by each patient. These are the numbers we need to calculate the average. We didn't really need the boxes for the Count and the Sum But I wanted to show these elements in action at least once. It also simplifies our work: If you call the Report Navigator you will see that these functions have been properly and automatically placed by Base at the Group level, sparing me the need to type them myself.

Now we need to create the function that will use our partial results. Go to the Report Navigator and find the function branch for the Group level Patient Name. Select it by right-clicking on it. Then left-click it, calling the contextual menu. Find and click the option “New Function”. Now a new function with the name “Function” appears under the two functions created automatically by Base. Select it, calling its Properties dialog box.

Let's first change the name of the function with something more descriptive. I will do “fnPaymentAvg”. The prefix identifies this as a function. The rest of the name reminds me that this function calculates the payment average. But you can try any name, like Edna or Maribel, and with or without prefixes. Notice the Report Navigator updating the name as soon as you shift focus from the box.

Now we will introduce our formula. We want the sum of payments divided by the number of payments. The function that holds the sum of payments is: *AccumulationAmount-Patient Name* and the function that holds the count is: *CounterPatient Name*. The formula should look like this:

```
([AccumulationAmountPatient Name]/[CounterPatient Name])
```

Type this and we are ready. Notice that we don't need an initial value so we are leaving that field empty. Let's leave Pre-evaluation set to “Yes”.

Now we need this formula displayed. Select the text box in the group footer that you had prepared for this. Go to the *Data* tab and select “User Defined Function” for the Data Field Type. Base responds by deactivating the Data field and Scope selections. When you click on Function you will find all of the functions available for this report (that should be three if you started this exercise from scratch). Find and click our function.

If you now run this report you will see the average of payments done by each patient displayed in the group footer section. Depending on how you have formatted this box you could see an integer number (with no decimal places) or a decimal number. You can further specify the format displayed by this box in its *General* tab.

You can now eliminate the boxes that display the count or the sum of the amounts. We used them only as guides and to have Base insert the needed functions for us.

Now it will be your turn to calculate and display the full average of payments in this report. Just remember to define your functions at the report level (at the top in the Report Navigator) although you will be using them in the report's footer.

I am sure that all the possibilities opened by the use of functions and formulas might be whetting your appetite: The ability to gather and process information in your report to produce further information not immediately evident and then dynamically adapt the output to the results is really tempting.

But wait, there is still more.

Conditional formatting.

Let's imagine that we need to highlight a possible value in one field. We could make it stand out by changing the color of the font, the typeface or some other element of the formatting but ONLY if it happens to be a certain value or within a range of values. What we need here is the ability to do conditional formatting.

For example, let's go back to the example for our Clinical Summary: It could be very important to highlight any active medication because the therapist might need to check on that. So we would like the report to write the “Still Active” string with some enhancement, say, using the color red and italics.

To achieve this go back to your report and select the text box that displays the “Med end date” field (the text box to the right of the label “Until:”). Now go to the menu and click on Format and then on Conditional Printing...

The Conditional Printing dialog box pops-up. You can see the name “Condition 1” and down of it two rows of controls. The first one sets a condition and the second row specifies how must the text appear if the condition is met. The name “Condition 1” suggest that more conditions and resulting formatting styles can be added. For this you can click on the button with a plus sign down and to the right.

By default, the field value is compared with a range of values by offering the operator “Between” and two boxes for parameters. Each of these boxes have buttons with the ellipsis. If you click on these you will call the Function wizard. So you can build very sophisticated conditions for the conditional printing.

However, our goal is to highlight when we get exactly the string “Still Active”. Let's start by clicking on the button with a down arrowhead in the second box, the one that states the operator, and find and select the “Equal to” operator. Notice that because this operator needs only one parameter, the condition row adjusts accordingly. In the Third box just type the string that we are looking to match: “Still Active”. Remember that for the SRB, we must enclose strings within double quotes.

This way the first row should read: *Field Value is Equal to “Still Active”*.

Now let's establish the way we want the formatting: Click on the button for italics and see the sample text change accordingly. The fifth button form the left calls for the color

menu. Select the color red and see again the sample text reflect this. Notice that the six buttons allow you to change virtually any aspect of the rendering of text.

Finally press the OK button and run the query. Pick a patient that has current and terminated medication records. Wow! What bout this!!

With this dialog box you could easily set that, for example, values below zero come out in red, values between one and 20 are printed in orange and numbers over 20 are rendered in blue. What about that? Or you could make that patients that have been in therapy for more than a year to the date the report is made appear highlighted; or you could compare values from different fields... in short, the possibilities are endless.

A word of caution:

With all its flexibility and powerful functions, the SRB is not without bugs. To my understanding, this is problematic at least at two levels: First because by not working the way it's supposed to it makes it difficult to accomplish what one wants. But second, it complicates the learning curve a lot: If I am not achieving the end result that I want, is it because I made a mistake or is it a software bug?

If we knew beforehand the effects of the bug in our programming we would probably not call them bugs but *issues* and just work around them. But we don't always know in advance what's happening. My advice? If things are not going the way you thought they should then be patient. Try again. Start anew. Check the forum's post (chances are that someone already confronted the bug, posted a question and received an answer). And, yes, report bugs.

Chapter 11

Maintenance of a Database.

Throughout the useful life of your database there are things that you might want to do to keep it current, reliable and safe. This includes modifying data structure, updating your forms, defragmenting your database and creating backups.

Modifying data structure.

You might remember all the time we spent carefully shaping our tables and their connections. It is not a good thing if we find ourselves trying to modify this with a database that we are actually using because the potential for doing some irreparable damage is quite big. We could lose some important data and we could even lose some critical data for the internal operation of the database itself (like primary key numbers, for example).

But business practices do change and the effort to start a new design, and later populate it with data, could be quite big compared to the effort of just adapting the database to the new practice. Where does the boundary lie will depend on your ability, available time and the degree of change in data structure required by the new business practice.

For example, let' say that the director of the clinic decides to include new offerings aside from therapy for adult individuals like family therapy, couples therapy and child and adolescent therapy. They will cost differently and the therapist will be payed differently. Now it makes sense to record the kind of therapy being provided so that we can charge and pay accordingly.

If we study the UML diagram for our database we will easily discover that the best place to include this would be in the "Assignment" table. We need to include a new column, that we could call "Modality", where we can store this new piece of information.

The instruction for doing this is the ALTER TABLE. You can find a complete description of what this instruction does in the documentation for HSQL at:

http://www.hsqldb.org/doc/guide/ch09.html#alter_table-section

There are many things you can modify with this instruction, including: Adding columns or constraints, dropping columns or constraints, renaming tables, altering columns and even changing the auto numbering of primary keys. Make sure to read the section so that you can understand the syntax by comparing the definition given there with the cases examined here.

For our example we will need to call the SQL command box at Tools>SQL... and type and execute:

```
ALTER TABLE "Assignment" ADD COLUMN "Modality" VARCHAR (25);
```

Now we have a new column of a varchar data type in the specified table. Of course, this column is completely empty.

Other examples of this SQL instruction include:

```
ALTER TABLE "Assignment" DROP COLUMN "Modality";
```

This instruction would eliminate the column "Modality" and all data in it would be lost.

```
ALTER TABLE "Assignment" ALTER COLUMN "Modality" RENAME "Therapy
Type";
```

This instruction changes the name of the column "Modality" to "Therapy type" in the Assignment table.

```
ALTER TABLE "Assignment" ALTER COLUMN "Assignment ID" RESTART WITH
32;
```

This instruction resets the counter for the primary key auto numbering in "Assignment ID" to 32

Go and check the documentation to get a feeling of other things that you can do.

Modifying forms.

You might want to modify your forms to make them easier to use or as a consequence of changing data structure. Let's see this in the context of our previous example.

Now that we have our new column we need to start recording the type of therapy that the patients are being assigned to. The modalities are: Adult individual, child and adolescent, couples therapy and family therapy. We know that we want to insert this data in a way to minimize data entry error because the generation of reports that charge or pay money are going to be based on precise counts of the types of therapy. With long names like these, the potential for error increases. I am sure that you are already thinking about radio buttons for this and you are right. You are going to need to modify your Assignment Data Entry Form and include these four new modalities using radio buttons so that the user only needs to point and click the desired option. You already know how to do this.

But what happens if the director of the clinic keeps changing the offer of therapeutic modalities, looking for products with the necessary public appeal? Let's say that he includes support groups for obsessive-compulsive disorders, grief and mourning and sex addiction. Now your form is getting really crowded with radio buttons. Later he decides to drop child and adolescent therapy because he has no qualified therapists at the moment and also sex addiction because no one has registered in it in the four weeks that it has been available. Two days later he comes with the request to include a drug addiction pro-

gram and AA meetings. There you are changing your form, adding and dropping radio buttons with no end in sight.

Don't despair, I have a little trick to offer you.

Let's think for a moment what is it that we are trying to achieve: We want to insert strings of text in a varchar column in a way that is simple and minimizes entry errors like misspellings. Our first inclination is to use radio buttons but with a big number of options the form becomes crowded quite quickly. We are also experimenting with several and new options, adding and changing them, so we want a mechanism that would simplify maintenance.

To achieve this we are going to use a List Box. Yes, I know that List Boxes and their drop-down feature are used for displaying names and then storing the corresponding primary keys while radio buttons just pass a string. But we can change this behavior. Also, Lists Boxes feed the options from a table, making maintenance easier.

Our first step will be to create an extra table to hold the different therapy types. This will be a simple table with a primary key and a Varchar column we will call "Type". You can include a description column as well. We do not need any other element in this table. You should know by now how to compose the SQL code for this:

```
CREATE TABLE "Therapy types" (
"ID" INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH 0) NOT NULL
PRIMARY KEY,
"Type" VARCHAR(25),
"Description" VARCHAR(100)
);
```

You can go to the Tables section, open this table and populate the "Type" column with the options required by the director. (Use: Adult individual, Child/Adolescent, Family Therapy and Couples Therapy). This way we have data to populate the drop-down list[20].

Now we'll edit the form where we make the assignments, which is linked to the “Assignment” table. Insert a List box. If the List box wizard appears then select the new table, "Therapy types", as source in the first step. In the second step chose "Type" to populate the drop down list. In the third step, under "Field from the Value Table", which is the receiving field, chose "Modality" (column we created earlier and should appear now) and for "Field from the List Table", which is the source field, chose "Type". You are ready to go. Notice how we made sure that both "Modality" and "Types" are of the same data type.

Now, it could happen that the wizard does not appear because Base has no way to know what tables you are trying to link. No problem if this happens because we can as easily use the Properties dialog box of the List box. Call for it and go to the *Data* tab. In the

20 The “Description” column allows you to record any important information about the different types, although is not really used in this example.

"Data field", which specifies the receiving column, pick "Modality". In "Type of list contents" chose SQL. For "List content" insert the following code:

```
SELECT "Type", "Type" FROM "Therapy Types";
```

and for "Bound field" select 1.

Note that the procedure is almost identical to the one reviewed at the end of chapter 8 for manually creating list box's links. The big difference is in the SQL code where, instead of selecting the column with the names we need and the corresponding primary key, we selected the names twice: The first call for “Type” populates the drop down list and the second call provides the content to be passed to the “Modality” column.

If you check the relationships (Tools>Relationships) you will see that our new table does not even appear with the rest of the tables. This is because we have not included a constraint that establishes a relationship via a foreign key like the other tables do and, so, the tables are not linked in this sense.

But right now the List box works like a super radio button: it offers mutually exclusive options and passes the selected string. Each time the Director comes with a new therapy modality, all you need to do is to include it in the “Therapy types” table and it will appear in the Drop-down list. If the list is getting too crowded, you can even delete from the table the options that are no longer being offered. How is that for easy maintenance!

Defragmenting your Database:

At times you could discover that your Base application requires a rather big amount of memory although it holds comparatively a small number of records. I've read several explanations for this: First, that the application appropriates an amount of memory in anticipation of the number of records it could need to hold. Second that, although our data appears to us in graciously ordered tables, the truth is that it's stored at different times and not necessarily in order. Alongside the piece of data we care for, HSQL is storing several indexes that relate the data to a particular table and a particular column, bloating the amount of data we need to keep. Lastly, that when you delete records these are not really expunged from the database but really just disconnected from the indexes of the database, lying there until the memory they use is overwritten. All this extra data means bigger file sizes. Just like with a hard disk, if we can reorganize our data then we will be able to discard those indexes and make our files smaller, faster and less prone to breakdowns. We do this by defragmenting our database file.

All you need to do is open the SQL window (Tools> SQL...) and type and execute:

```
CHECKPOINT DEFRAG
```

This will take care of all the extra unneeded information in the file. Don't be startled: This instruction will first close the database, reorganize data and then re-open the database.

There is another instruction that behaves similarly: SHUTDOWN COMPACT, but does not re-start your database after reorganizing and minimizing data. Read more about them in Chapter 9 of the HSQL documentation at www.hsqldb.org

Backups:

Terrible things can happen to a computer: Viral infections, hard disk drive breakdowns, computer crashes that damage disk sectors and a long etcetera. Most of these things don't even have to do with Base although Base can also crash. Good thing that OpenOffice.org comes with a good recovery wizard that could minimize the data lost.

However, there is always potential for data loss.

The smart thing to do, of course, is to keep backups. All you need to do is make copies of your Base database files and keep them in other places of storage like secondary hard disks, flash memory drives or optical media (CDs DVDs). You just have to select the *.odb file of interest, copy it and paste it in the secondary media of storage. With this solution you save all at once: Data structure, data, forms, queries and reports. Of course, any backup will be good only to the moment it is created. How often should you create them depends on how often you modify forms or reports or how often you update or modify data.

Chapter 12

Some final words:

I really like Base working with HSQL. When I started writing this tutorial -mostly to learn how to use them- I didn't know much about databases. Today I can see that you can use them in a wide range of applications, not only keeping customer data and recording rendition of services -which are already fine areas of application- but also in scientific research, administrative work and even, if you think about it creatively, for writing literary fiction or creating games. I remember reading a suggestion on using Base with the SRB for writing very focused cover letters and resumes with the ease of point and click.

Base can work as a front end for many other database management systems, not only HSQL. But HSQL is nice because it integrates very well with Base. This means that they will work together with not much discord, taking advantage of the ease with which Base can set up things and the reliability with which HSQL stores and processes data. The programmers at OpenOffice.org tested many options before deciding on the integration of Base and HSQL into an embedded application. Furthermore, if you check the HSQL web site (www.hsqldb.org) you will read that their programmers are very proud of its speed, the small amount of memory the program itself requires, the reliability with which it handles data and other advanced functionalities.

But if you read the ooforum.org you will find lots of skepticism about Base working with the embedded HSQL engine. They would definitively not recommend this solution to store critical data and would shy away from the idea of having your business depending on it. This might come as a disappointing and disquieting revelation to you, who has just finished a quite long journey understanding Base and learning how to take advantage of it. Why do they say this?

In simple words: Data corruption. The thing is that by having Base work with HSQL in embedded mode, ALL of your information is stored in ONE file (the *.odb Base file). Your forms and reports are stored in this file but also your data structure and your data, all conveniently zipped for storage. And it so happens that there is a chance of Base inadvertently transforming some element of this data and thus potentially making it impossible to read and write the file appropriately. The change could be trivial but it can have a devastating effect on your ability to access and use your information. If this happens to you and you are courageous enough, you could try unzipping the file and exploring the scripts that are read to build your application every time you open it. If you spot the corruption and you can fix it, you might be able to recuperate your data and even the database[21]. Unfortunately, this might not always be possible

21 Find info at www.oooforum.org on how to do this. Just to learn more, you should unzip and explore a spare odb file. Before unzipping you must change the .odb extension to .zip so it can be recognized as a zipped file.

So the fact remains: This corruption can occur. Therefore the embedded option is not completely reliable.

What do we do now? Because the problem is related to having all your data in one file, the recommendation is to have the data and data structure in a separate file to the rest of your application (forms, reports, etc.). This way, any corruption or crash created by Base will not affect the tables and their stored information. If the corruption in a Base file pushes it beyond recovery, you can just open a new Base file and ask to connect to your tables. Problem solved. All the stability and reliability promised by the database engine becomes yours to enjoy again.

This is not difficult to achieve. It basically consists of downloading an HSQL file (with a .jar extension) and storing it somewhere in your computer as resident and then have Base connect to it. If you want to, you can even chose a different database system altogether. For example, at the time of the writing of this text, H2 has become a very attractive alternative as it offers encrypted tables and other perks. HSQL 2.x also offers attractive features (including encrypted data). Both these options integrate very well with Base. Furthermore, you can find at the ooforum.org an overview of the pros, cons and features of Base working with several other relational database management systems. I will not cover how to connect Base with an external database because the process will be different depending on which one you pick and they are all quite well documented at the ooforum.org, which should become the continuation of this tutorial to you.

And what happened with all the techniques that we had learned here? They remain valid. You can work with Base just the same -so we haven't lost our time! The only difference is in the way you open your application (by asking to connect to it). After that, you keep working with Base the way it has been proposed in this tutorial (insert a sigh of relief here). If you chose another database management system, you might need to adapt to the naming conventions and slight changes in SQL syntax that the new system uses, but this should be a snap to you. Also notice that there could be varying degrees of integration, with HSQL and H2 offering the best level.

As I said, I started writing this tutorial because I wanted to learn how to use databases -and Base particularly- applied to my needs, understanding enough about the subject to help me remain flexible with the options offered and using them in the best way possible. I trust that you have read so far because you had similar motivations. I did reach my goal. I am not an expert but do go around explaining my friends how a database could help them with their problem and, so far, have been able to devise solutions for the (rather simple, I admit) problems I have encountered. I hope that this tutorial has provided you with the same help in designing database applications that fit your needs and has given you a foundation from which to continue exploring Base.