NEAR EAST UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

DATABASE SYSTEM FOR A HOSPITAL

GRADUATION PROJECT COM-400

Student : **Selçuk GİKİOĞLU**

Supervisor : **Assist. Prof. Dr. Firudin Muradov**

Nicosia-2003

ACKNOWLEDGEMENTS

I would like to thank Assist. Prof. Dr. Firudin Muradov for accepting to be my supervisor and his support for this project.

I am so grateful to deceased my father. Also my mother, brother and sister thanks who had always shown patience and understanding to me.

Also, I would like to thank all the lectures for helping me see this graduation term and finally, I would like to thank all my friends for their support in school and in social life.

ABSTRACT

Data, gathered around us as a collection of facts, is of no use unless it is organized and represented in some meaningful form. Data represented in some meaningful form like, tables, charts or graphs become information, which can be easily processed. The collection of data, usually refereed to as the database, contains information about one particular enterprise. These days database are used by a variety of users and organizations, which are important tools in data processing DBMS, are designed to manage large bodies of database information.

This project has as its goal to develop software, processing information about patient of a Life Hospital . software developed in this project contains both of hospital department information. I wish to develop this software for processing information of the Hospital.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

DBMS	Data Base Management System
RDBMS	Relational Data Base Management System
GUI	Graphic User Interface
SQL	Structure Query Language
DDL	Data Definition Language
DCL	Data Control Language
DML	Data Manipulation Language
I/O	Input/Output
IT	Information Technology
ISO	International Standard Organization
ANSI	American National Standards Organization
SEQUEL	Structured English Query Language
CPU	Central Processing Unit
OLAP	On Line Analytical Processing
CGI	Common Gateway Interface
ER	Entity-Relation
VB	Visual Basic
OOP	Object Oriented Programming

INTRODUCTION

A Database management system (DBMS) is a collection of programs that enable users to create and maintain a database.

A DBMS is a computerized record-keeping system that stores, maintains and provides access to information. A database system involves four major components DATA, HARDWARE, SOFTWARE, USERS. DBMS are used by any reasonably self-contained commercial, scientific, technical or other organization from a single individual to a large company and a DBMS may be used for many reasons. The objective of this project was to design software for a company, which deals with the computer sales and purchase, so fully qualified software has been made, and at the making of the software two companies, was visited to understand the requirements. And the problem these types of company may have. The Software is fully capable to store any computer parts with the manufacture name. For the simplicity purpose a manufacture ID and Product ID has been generated, the first digit consists of manufacture name and the rest of it contains the product name. How ever both (ID and NAME) has been entered in the form. As for the mentioned problem from one of the company, employee information is also entered in the software, so that the complete information about the Company employee can also be maintained. Voucher is also design to minimize the handwork. The project consists of introduction, 4 chapter and conclusion.

Chapter One: Introduction to DBMS contains brief information about the database, data model, advantages of database the architecture of the DBMS, properties of DBMS data and further different information related to DBMS.

Chapter TWO: Relational Database Management System describes that what is RDBMS , components of RDBMS and the Issues.

Chapter Three: Describes pattern language for Object – RDBMS

Chapter Four: Help Option, contain the help information about the software which also describe the tables, form, quires, reports in general, and how to use the software option which contain the pictures of the software for the helpdesk.

Finally, the conclusion section presents the knowledge gain during the making of the project.

CHAPTER ONE

INTRODUCTION TO DBMS

1.1 Database

In a typical file-processing environment, each user area such as payroll, personnel, and the speakers' bureau, has it own collection of files and programs that access files. Since there is usually overlap of data between user areas, there is redundancy in the system. The address of a faculty member can occur in many places, i.e. while this is certainly wasteful, trying to produce reports or respond to queries that span user areas can be extremely difficult. These problems lead to the idea of a pool of data, or database, rather than separate collections of individual files.

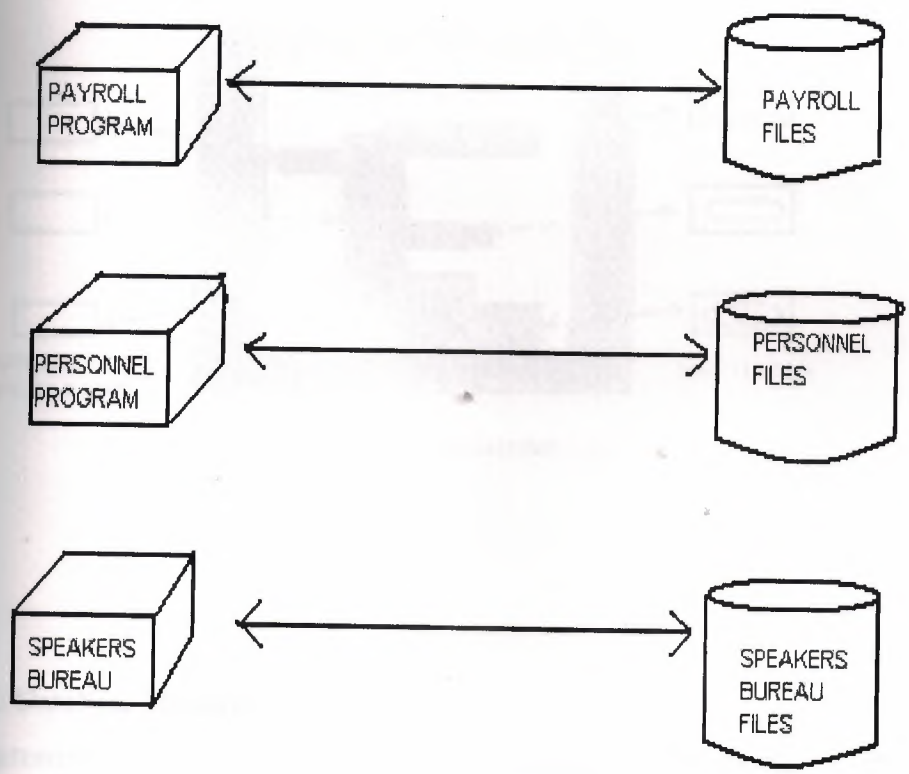


Figure 1.1 Database

1.2 What Makes Up a DBMS?

A DBMS is a computerized record-keeping system that stores, maintains and provides access to information. A database system involves four major components, which are as follows.

1. DATA
2. HARDWARE
3. SOFTWARE
4. USERS

DBMS are based by any reasonably self-contained commercial, scientific, technical or other organization from a single individual to a large company and a DBMS may be used for many reasons. Data itself consist of individual entities, in addition to which there will be relationships between entity types linking them together. Given an enterprise with a nebulously defined collection of data, the mapping of this collection onto the real DBMS is done based on a data model. Various architectures exist for databases and various models have been purposed including the relational, network and hierarchic model.

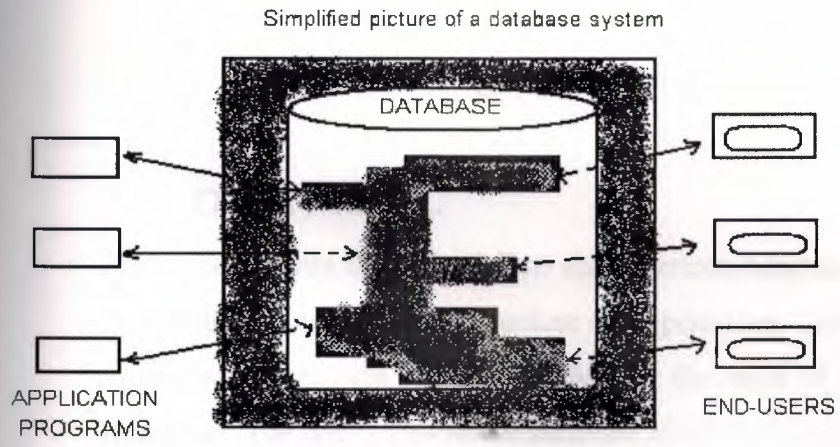


Figure 1.2 [web_pages111notesDBMS.htm]

1.3 Database Management System

Fortunately, software package called database management system can do the job of manipulating actual database for us. A database management system, at its simplest, is a software product through which users interact with a database. The actual manipulating of the underlying database structures is handled by the DBMS.

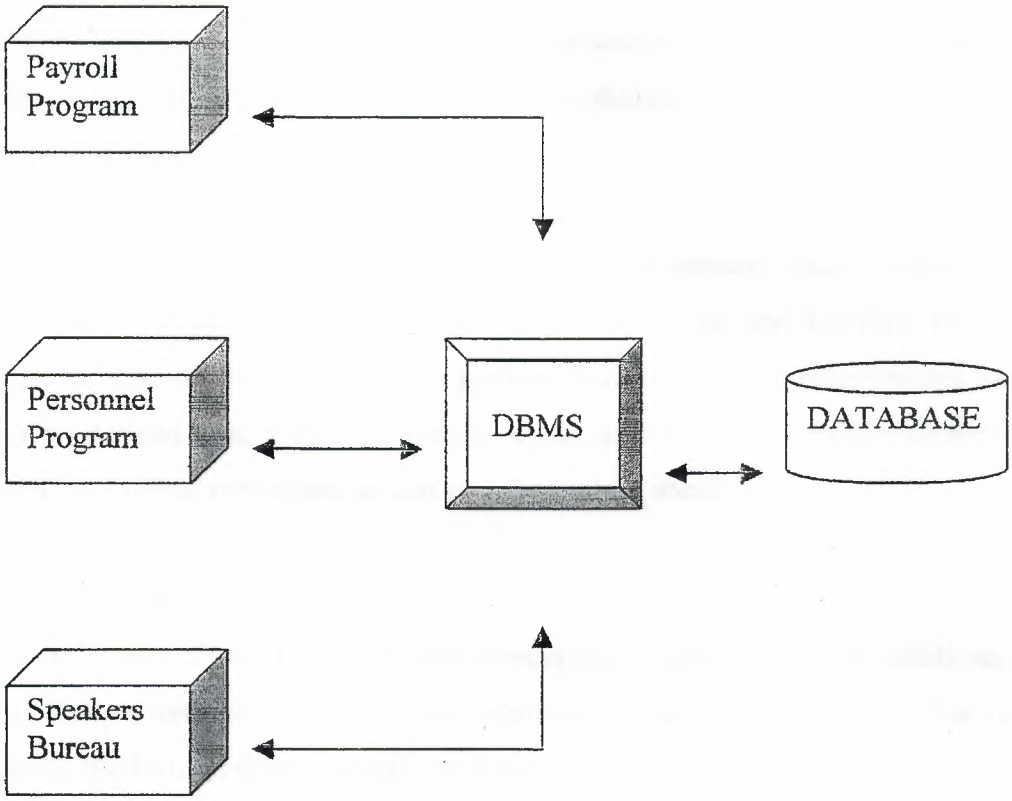


Figure: 1.3 Database Management

1.4 DATA MODEL

The model of data that they follow characterizes database management systems. A Data model has two components-structure and operations. The structure refers to the way the system structures data or, at least, the way the users of the DBMS feel that the data is structured. The operations re the facilities given to the users of the DBMS to manipulate data within the database. What is crucial is the way things feel to the user, it does not matter how the designers of the DBMS choose o implement these facilities behind the scenes.

There are three models, or categories, for the vast majority of DBMS's :

- Relational model
- Network model
- Hierarchical model.

1.4.1 Relational Model

The user as begin just a collection of tables perceives a relational model database. Formally, these tables are called **relations,** and this is where the relational model gets its name. Relationships are implanted through common columns in two or more tables.

1.4.2 Network Model

The user as a collection of record types an relationships between these record types perceive a network model database such a structure is a network, and it is form this that the model takes its name. In contrast to the relational model, in which relationships were implicit (being derived from matching columns in the tables), in the networks model the relationships are explicit (presented as part of the structure itself).

1.4.3 Hierarchal Model

A user as a collection of hierarchies (or trees) perceives a hierarchies model database. A hierarchy is really a network with am added restriction; no box can have more than one arrow entering the box. (it doesn't matter how many arrows leave a box). A hierarchy is thus a more restrictive structure than a network.

1.5 ADVANTAGES OF DBMS

The main advantages of using a DBMS is that the formalism of the model of data underlying the DBMS is imposed upon the data set to yield a logical and structured organization of the data. Given a fuzzy, real-world data set, when a model's formalism is imposed in that data set the result is easier to manage, define an manipulate. Different models of data lead to different organizations. In general the relational model is the most popular because that model is the most abstract and easiest to apply to data while still begin powerful.

Therefore, using a DBMS we have the following advantages.

- Clear picture of logical organization of data set.
- Centralization for multi-users.
- Data independence.

1.6 THE 3 LEVEL ARCHITECTURE

The three level architecture is an architecture for a DBMS to provide a framework for describing database concepts and structures. Not all DBMS fit neatly into this architecture, but most do. The model has been proposed by ANSI/SPARC and has three levels. Mappings exist between the three levels and it is the responsibility of the DBA to ensure these mappings are correct.

- External level (individual users view)
- Conceptual level (community user view)
- Internal level (storage)

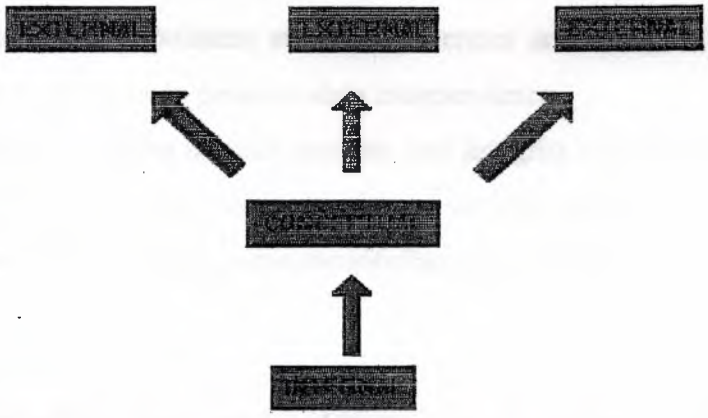


Figure: 1.4. Three Level Architecture [www.compapp.dcu.ie]

1.6.1 External Level

The external level of the three level architecture is the individual user level. At this level each user has a language at their disposal of which they will use a "data sub language" i.e. a subset of the total language that is concerned specifically with database operations and objects. For the application programmer, the language will be a conventional language e.g. COBOL with embedded SQL, or a specific one e.g. dBASE. For the end user, it will normally be a query language like SQL or a special purpose language. In principle, any given data sub language consists of a DDL (to declare data objects) and a DML (data manipulation language) to manipulate these objects

An individual user's view is an external view, which is thus the content of the database as seen by that particular user. There will thus be multiple occurrences of multiple types of external records. The external view is defined by an external schema, which in turn is defined by the DDL part of the user's data sub language

1.6.2 Conceptual level

The conceptual level of the three level architecture is essentially a representation of the entire information content of the database in a form abstracted from physical storage. It may also be quit different or similar to external views held by a particular user. It is data as it really is. Rather than as users are forced to see it- it is multiple occurrences of multiple types of conceptual records.

The conceptual schema is defined by the conceptual data definition language (DDL). There is no reference in the conceptual DDL to stored record concepts, sequences, indexing, hash addressing, pointers etc. the references are solely to the definition of information content, in order to preserve data independence.

Conceptual schemas will also include security and integrity constraints as well as data definitions. Normally the conventional schema is little more than a union of all individual external schemas, plus some security/integrity checks.

1.6.3 Internal level

The internal level of the three level architecture is a low level representation of the entire database; it consists of multiple types of internal record. It does not deal with block/pages or device-dependant concepts like cylinders and tracks. The internal system defines types of stored records and indexes, how fields are represented, various storage structures used, whether they use pointer chains or hashing, what sequence they are in, and so on. The internal schema is written using yet another data definition language, the internal DDL.

Programs accessing this level directly (i.e. utility programs) are dangerous since they have by-passed the security and integrity checks which the DBMS program normally takes responsibility for.

1.7 PROPERTIES OF DBMS DATA

DBMS are available on any machine, from small micros to large mainframes, and can be single or multi-user obviously, there will be special problem in multi-user

environments in order to make other users invisible, but these problems are internal to DBMS.

Data may be shared over many databases, giving a distributed DBMS, though quite often it is centralized and stored in just one database on one machine. In general, the data in the database, at least in a large system, will be both integrated and shared.

1.8 WHO USES A DBMS

There are three broad classes of users who use a DBMS

- Application programmers
- End users
- Database administrator

1.9 HARDWARE FOR A DBMS

Conventional DBMS hardware consists of secondary storage devices, usually hard disks, on which the database physically resides, together with the associated I/O devices, device controllers, I/O channels and so forth. Databases run on a range of machines, from microcomputers to large mainframes.

Other hardware issues for a DBMS includes database machines, which is hardware designed specifically to support a database system.

1.10 DATABASE SECURITY

The DBA can set up the DBMS such that only certain users or certain application programs are allowed perform certain operations to the dataset e.g. only admissions are allowed create records for students, only library are allowed to create records for books etc. Different checks can be established for each type of access to each type of information in the database. Different users should have different access rights to different objects.

SQL provides two methods for implementing security restrictions. These are:

- Views - can be provided to hide sensitive data.
- GRANT/REVOKE - grant or remove access privileges to specific users for specific tables.

There is, however, a major drawback to SQL security

1.11 HOW DATA IS STORED

A data model is defined as a set of guidelines for representing the logical organization of data in the database; a pattern according to which data and relationships can be organized; an underlying mathematical formulation for building logical data organizations.

A data model consists of

- A named logical unit (record type, data item)
- Relationships among logical units

A data item is the smallest logical unit of data, an instance of which is known as a data item value.

A record type is a collection of data items, and a record is hence defined as an instance of a record type.

Note: A data model does not specify the data, data implementations or physical organization only the way it can be logically organized.

1.12 DEFINITION OF ENTITY

An entity is any distinguishable real world object that is to be represented in the database; each entity will have attributes or properties e.g.

The entity *lecture* has the properties *place* and *time*. A set of similar entities is known as an entity type.

DATABASE APPLICATION LIFE CYCLE

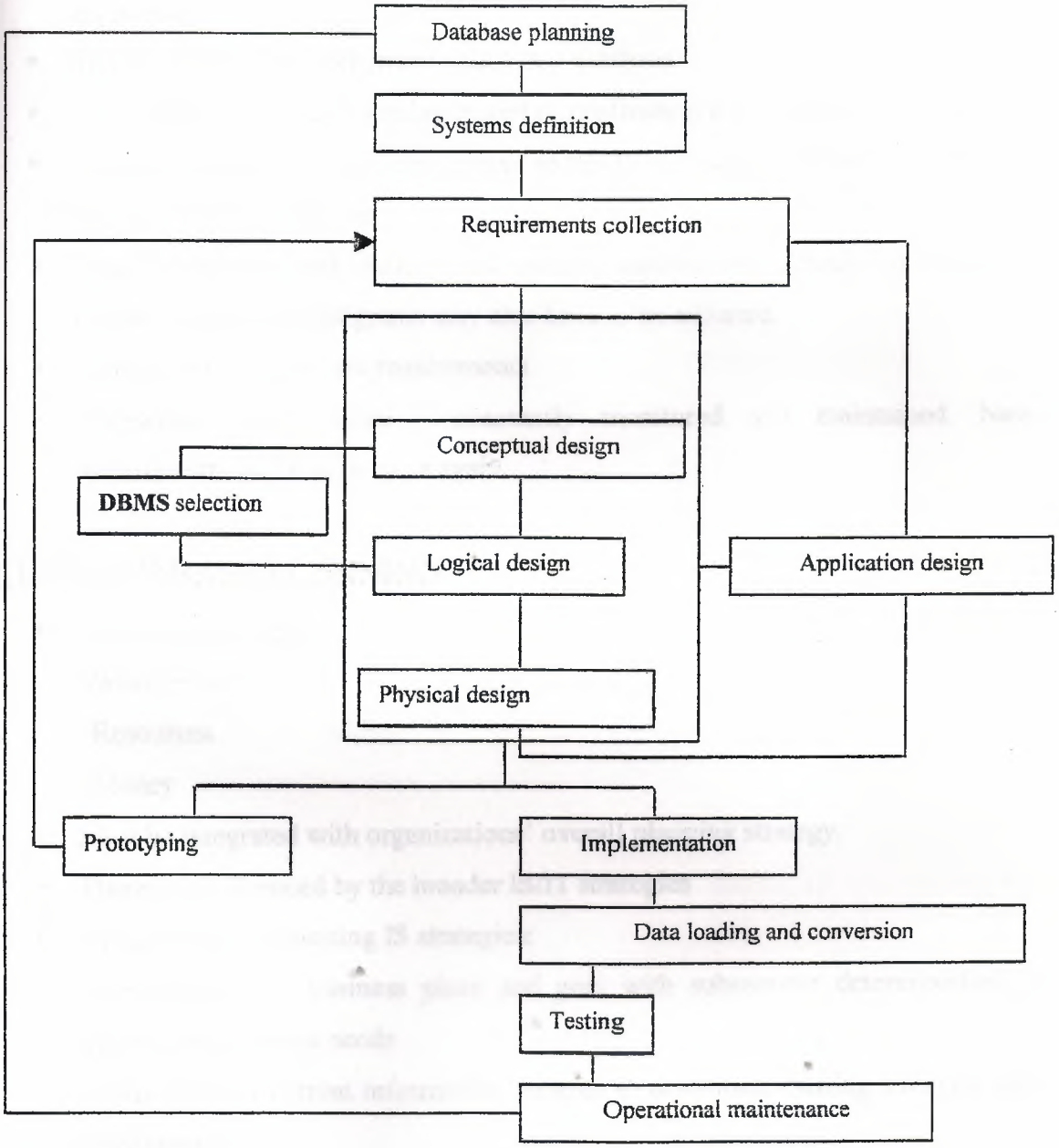


Figure: 1.5 [www.compapp.dcu.ie]

- Database system is a fundamental component of the larger organization information system. Therefore associated with the information system lifecycle*Database Planning* -involves planning how the stages of the lifecycle can be realized most efficiently and effectively

- *System definition* - scope and boundaries of application, users, areas
- *Requirements* - from users and previous applications
- *Database Design* - of the database itself
- *DBMS Selection* - optional and involves getting a suitable product for application
- *Application Design* - programs which use database
- *Prototyping* - optional, working model of application for designers and users
- *Implementation* - creating conceptual, external and internal database definitions and application programs
- *Data Conversion and loading* - old system replacement, directly or with new format. Application programs may also have to be adjusted
- *Testing* - against the user requirements
- *Operational maintenance* - constantly monitored and maintained. New requirements go through cycle again.

1.13.1 DATABASE PLANNING

- 3 main components:

  –Work to be don

  –Resources

  –Money
- Must be integrated with organizations' overall planning strategy.
- Therefore influenced by the broader IS/IT strategies
- 3 main issues concerning IS strategies:

  –Identification of business plans and goal with subsequent determination of information systems needs

  –Evaluation of current information systems to determine existing strengths and weaknesses

  –Appraisal of IT opportunities that might yield competitive advantage.
- A corporate data model can be developed showing main entities and relationships of the organization and functional areas of the organization.

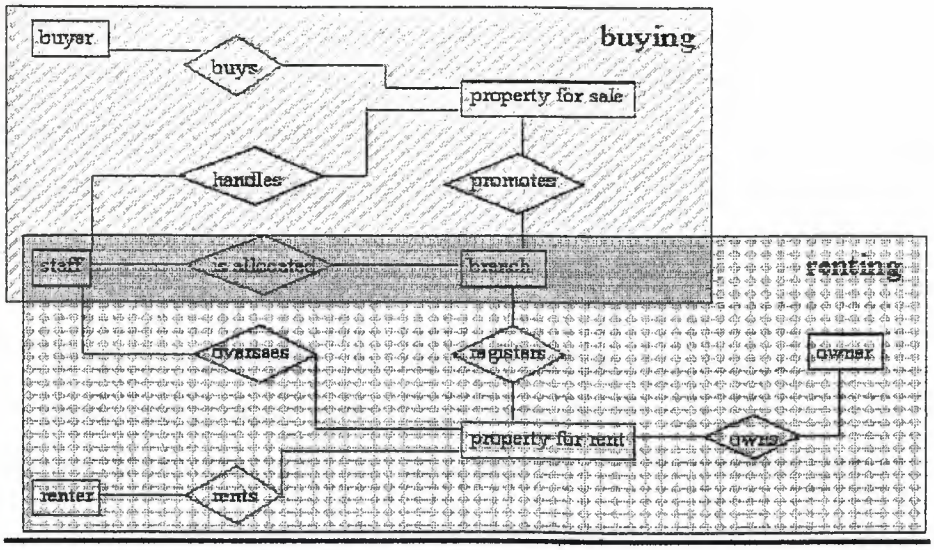


Figure: 1.6 Data base Planning Architecture [www.compapp.dcu.ie]

- The functional areas may be assigned priority in line with the corporate strategy to define scope of the database for system development.
- Database Administrator can develop plans to achieve this.
- Standards may be developed

  –How data is collected

  –Necessary documentation

  –Design and implementation procedures
- Good for training staff and quality control
- Legal or company requirements concerning data should b3 documented e.g confidentiality.

1.13.2SYSTEM DEFINITION

- Identify boundary of the system
- Identify how it interfaces with other parts of the information systems
- Include current users and application areas
- Future users and application areas

1.13.3Requirements Collection and Analysis

- Gathered:

  –Interviewing

–Observation

–Examination of documents (record and display information)

–Questionnaires to users

–Experience form the design of similar systems.

- Results in users' requirements specification of the enterprise.
- Perhaps from many viewpoints
- Too much study too soon - Paralysis by analysis
- Too little - unnecessary waste of time and money
- Convert to formal requirements specification (DFD's and CASE tools etc.)

1.13.4DATABASE DESIGN

- Major aims;

  –Represent data and relationships required by all application areas and user groups

  –Provide data model that supports transactions required on the data

  –Specify a design, which will achieve stated performance requirements for the system e.g. response time.

- Bottom-up approach - good for simple databases

  –Starts with data fields

  –Normalization

- Top-down approach - good for complex database systems

  –Development of data models

  –Refine to identify lower-level entities, fields and relationships

  –ER modeling

- DBMS Selection

  –Selection could be done at any tie prior to logical design

  –Based on system requirements

- Performance
- Ease of restructuring
- Security
- Integrity
- Application Design

  –May not be able to complete application design until db design finished

  –Must match requirements

–User interfaces

- Prototyping

–Does not normally have complete functionality

–Allows users to identify, which parts work well or not

–Suggest changes/improvements

–Inexpensive but time consuming (ask user, get feedback, fix, ask user....)

–Useful if clarification of user' requirements is required before implementation of a high cost, high risk or new technology.

- Implementation

–Achieved using;

- DDL

-Complied and used to create database schemas and empty database files and define user views

- Application programs implemented using 4GL or DML of target DBMS or both
- Security and integrity controls implemented
- Conversion and Loading–If new database system is replacing old system (legacy)

–Common to have conversion utilities

–Plan transition

CHAPTER TWO

INTRODUCTION TO RDBMS

2.1 What is an RDBMS?

In recent years, database management systems (DBMS) have established themselves as the primary means of data storage for information system ranging from large commercial transaction processing applications to PC-based desktop applications. At the heart of most of today's information systems is a relational database management system (RDBMS).

RDBMS's have been the workhorse fro data management operations fro over a decade and continue to evolve and mature, providing sophisticated storage, retrieved, and distribution functions to enterprise-wide data processing and information management system provides organization data into meaningful information systems. The evolution of high-powered database engines has fostered the development of advanced "enabling" technologies including client/server, data warehousing, and online analytical processing all of which comprise the core of today's state-of-the-art information management systems.

Examine the components of the term relational database management system. First, a database is an integrated collection of related data. Given a specific data item, the structure of a database facilitates the access to data related to it, such as a student and all of his registered courses or an employee and his dependents. Next, a relational database is a type of database based in the relational model; non-relational database commonly use a hierarchical, network, or object-oriented model as their basis. Finally, a relational database management system is the software that manages a relational database. These systems come in several varieties, ranging from single-user desktop systems to full featured, global, enterprise-wide systems.

2.2 THE RELATIONAL DATABASE MODEL

Most of the database management systems used by commercial applications today are based on one of three basic models:

1. Hierarchical Model; Network Model OR
2. Relational Model

2.2.1 Hierarchical Model

The first commercially available database management systems were of the CODEASYL type, and many of them are still in use with mainframe-based, COBOL applications. Both network and hierarchical database are quite complex in that they rely on the use of permanent internal pointers to relate records to each other. i.e. in an accounts payable application, a vendor record might contain a physical pointer in its record structure that points to purchase order records. Each purchase order record in turn contains pointers to purchase order line item records.

The process of inserting, updating and deleting records using these types of database required synchronization of the pointers, a task that must be performed by the application. As you might imagine, this pointer maintenance required a significant amount of application code (usually written in COBOL) that at times could be quite cumbersome.

2.2.2 Relational Model

Relational database rely on the actual attribute values as opposed to internal pointers to link records. Instead of using an internal pointer from the vendor record to purchase order records, you would link the purchase order record to the vendor record using a common attributer form each record, such as the vendor identification number.

Although the concepts of academic theory underlying the relational model are somewhat complex, you should be familiar with are some basic concepts and terminology.

Essentially, there are three basic components of the relational model:

1. Relation Data Structure
2. Constraints that Govern the Organization of the Structure
3. Operations that are Perform on the Data Structure.

2.3 RDBMS COMPONENTS

Two important pieces of RDBMS architecture are the Kernel, which is the software, and the data dictionary, which consists of the system-level data structures used by the kernel to manage the database.

2.4 Relational Data Base Management Issues

- Integrity
- Security
- Recovery
- Concurrency

2.4.1 Security

The advantage of having shared access to data is in fact a disadvantage also

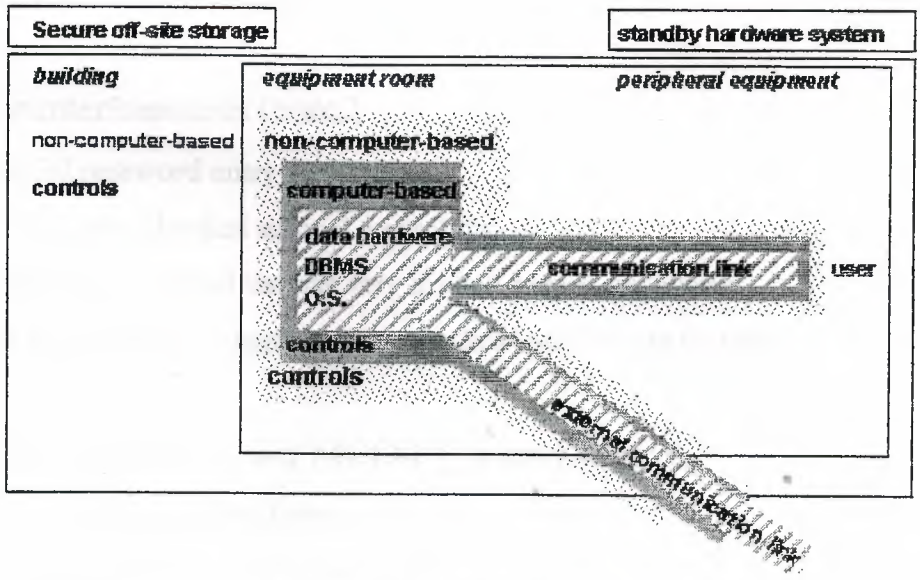


Figure: 2.1 Security [Aptech]

- Consequences: loss of competitiveness, legal action from individual
- Restrictions

  -Unauthorized users seeing data

  -Corruption due to deliberate incorrect updated

-Corruption due to accidental incorrect updated

- Reading ability allocated to those who have a right to know
- Writing capabilities restricted for the casual user - who may accidentally corrupt data due to lack of understanding
- Authorization is restricted to the chosen few to avoid deliberate corruption

2.5 Countermeasures (computer based)

2.5.1 Authorization

-Determine user is who they claim to be

-Privileges

Passwords

-Low storage overhead

-Many passwords and users forget them - write them down!!

-User time high - type in many passwords

-Held in file and encrypted.

2.6 Countermeasures (cont.)

-Initial password entry to system

-User name checked against control list

-The access control list has very limited access, superuser

-If many users and applications and data then list can be large

2.7 READ, WRITE, and MODIFY access controls

-Restrictions at many levels

-Database Level: 'Adds a new DB'

-Record Level: 'delete a new record'

-Data Level: 'delete an attribute'

- Remember there are overheads with security mechanisms

2.8 Countermeasures (cont.)

- Views

- Subschema
- Dynamic result of one or more relational operations operating on base relations to produce another relations
- Virtual relation - doesn't exist but is produce at runtime
- Back-up
- Periodic copy of database and log file (programs) onto offline storage
- Stored in secure location

2.9 Countermeasures (cont.)

- Keeping log file of all changes made to database to enable recovery in the event of failure
- Check pointing

  -Synchronization point where all buffers in the DBMS is force-written to secondary storage
- Integrity (see later)

  -Encryption

  -Data encoding by special algorithm that render data unreadable without Decryption key

  -Degradation in performance

  -Good for communication

2.10 Countermeasures (cont.) Associated procedures

- Specify procedures for authorization and backup/recovery
- Audit: auditor observe manual and computer procedures
- Installation/upgrade procedures
- Contingency plan
- Escrow agreement.

2.11 Non-Computer Counter Measures

- Establishment of security policy and contingency plan
- Personnel controls
- Secure positing of equipment, data and software
- Escrow agreements (3rd party holds source code)
- Maintenance agreements

- Physical access controls
- Building controls
- Emergency arrangements.

2.12 Privacy in Oracle

- User gets a password and user name
- Privileges:

  Connect: users can read and update tables (can't create)

  Resource: create tables, grant privileges and control auditing

  DBA: any table in complete DB
- User owns tables they create

  They grant other users privileges:

  Select: retrieval

  Insert: new rows

  Update: existing rows

  Delete: rows

  Alter: column def.

  Index: on tables
- Owner can offer GRANT to other users as well

  This can be revoked
- Users can get audits of:

  -List of successful/unsuccessful attempts to access tables

  -Selective audit e.g. update only

  -Control level of detail reported
- DBA has this and logon, logoff oracle, grants/revolts privilege
- Audit is stored in the Data Dictionary.

2.13 Integrity

- Introduction
- Basic concepts
- Integrity constraints
- Relation constraints
- Domain constraints

- Referential integrity
- Explicit constraints
- Static and Dynamic Constraint

CHAPTER THREE

A PATTERN LANGUAGE FOR OBJECT-RDBMS INTEGRATION

3.1 The Static Patterns

The Static Patterns for the relational side deal with when and how to best define a database schema to support an object model. The identity of the objects, their relationships (inheritance , aggregation, semantic associations) and their state must be preserved in the tables of a relational database. Table Design Time deals with when is the best time during development to actually design the relational schema. Representing Objects as Tables, Representing Object Relationships as Tables, Representing Inheritance in a Relational Database, Representing Collections in a Relational Database and Foreign-Key Reference deal with defining the relationships between objects and defining each object's state. Object Identifier (OID) defines how to establish object identity in a relational database.

3.2 Table Design Time

Problem.

When is the best time to design your relational database during object-oriented development?

Forces.

Assume no legacy database exists prior to development or if one does exist, it is extremely flexible. When the database design is kept foremost in mind during development, the object model will tend to be data driven while the behavior and responsibilities of the objects will be deprived of the thought and energy they deserve. Consequently, the object model will tend to have separate data objects and stupid data objects rather than a better, more distributed, less-centralized design. If the database design is completely ignored until the application is completed the project may suffer. Since 25 % to 50 % of the code in such applications often deals with object-database integration, the design of the database is crucial and should be considered early in development. Consequently:

Solution.

Design the tables based on your object model after you have implemented it in an architectural prototype but before the application is in full-stage production.

Discussion.

Definition of domain object behavior and properties is in reality a first pass at the database design. A stopgap persistency approach (perhaps using flat ASCII files) is often "good enough" for an architectural prototype. A benefit of this approach is that legacy data can be quickly exported from existing databases to an ASCII file. The prototype can then be easily demonstrated on stand-alone workstations that may not have a relational database and still show "real" data familiar to customers.

3.3 Representing Objects as Tables

Problem

How do you map an object structure into a relational database schema?

Forces

Objects do not map neatly into tables. For instance, object classes do not have keys. Tables do not have the same identity property that objects do. The data types of tables in a relational database do not match the classes in the object model. Complex objects can reference other complex objects and collections of objects.

Solution

Begin by creating a table for each persistent object in your object model. Determine what type of object each instance variable is likely to contain. For each object that is represent able in a database as a base data type (i.e., String, Character, Integer, Float, Date, Time) create a column in the table corresponding to that instance variable, naming it the same as the instance variable. If an instance variable contains a Collection subclass, use 1 Representing Collections in a Relational Database. If an instance variable contains any other value, use 1 Foreign-Key

Discussion

The design of the database may need modification (for instance, denormalization) depending upon the access patterns required for particular scenarios. Remember that this design is an iterative process. There are several variations of mappings between classes and tables. These are:

- 1 Object Class maps to 1 table
- 1 Object Class maps to multiple tables
- Multiple object classes' map to 1 table.
- Collections of the same class map to a 1 table
- Multiple object classes map to multiple tables

The Database Access Architecture must handle each of these variations.

3.4 Representing Object Relationships as Tables

Problem

How do you represent object relationships in a relational database schema?

Forces

A variety of relationships exist between classes in an object model. These relationships may be:

- 1 to 1 (husband - wife)
- 1 to many (mother-child)
- Many to many (ancestor - child)
- Ternary (or n-ary) associations (student - class - professor)
- Qualified associations (company - office - person)

A Qualified association is an association between two objects where the association is constrained or identified in some way. For example a Company can be associated with a Person through a position held by that Person. The position qualifies the association between the Company and the Person.

The association between objects may represent containment, associated properties or have come special semantic meaning in their own right (e.g., a marriage is a special relationship between a man and a woman).

The choices for 1 to 1, and 1 to many relationships are either to merge the association into a class or to create a class based on the association.

It is important to remember that the semantics of relationship between objects can be significant. It is often is useful to create classes to represent the associations, especially if the relationship has values of its own. These classes will be represented as tables in the relational database. For many to many, 1 to many and 1 to 1 associations, when an association has a meaningful existence in the problem domain, create a class for the association. A meaningful existence is when the relationship itself can have value such as the relationship itself possessing properties such as duration, quality or type. A marriage is a relationship between a man and a woman that can have all these properties.

Solution

Merge 1 to 1 associations with no special meaning into one of the tables. If it has special meaning create a table based on the class derived from the association.

For 1 to many associations, create a relationship table (see Representing Collections in a Database).

A many to many relationship always maps to a table that contains columns referenced by the foreign keys of the two objects.

Ternary and n-ary associations should have their own table that references the participating classes by foreign key.

A qualified association should have its own table.

Discussion

Consideration of the forces of this pattern will often result in changes to a first-pass object model. This is desirable, since it will often generate a more general and flexible solution.

Related Patterns

1. Representing Inheritance in a Relational Database
2. Representing Collections in a Relational Database

3.5 Representing Inheritance in a Relational Database

Problem

How do you represent a set of classes in an inheritance hierarchy in a relational database?

Forces

Relational databases do not provide support for inheritance of attributes. It is impossible to do a true 1-1 mapping between a relational table and a class when that class inherits attributes from another class, or if other classes inherit from it.

There are two possible contexts that are used in this pattern, depending upon what is more important to your particular application, speed of queries, or maintainability and flexibility of your relational schema.

Solution

(When ease of schema modification is paramount)

Create one table for each class in your hierarchy that has attributes. This will include both concrete and abstract classes. The tables will contain columns for each of the attributes defined in that class, plus an additional column that represents the common key shared between all subclass tables. An instance of a concrete subclass is retrieved by doing a relational JOIN of all of the tables in a path to the root with the common key as the join parameter.

Discussion

This is a direct mapping, which makes it easy to change if a class anywhere in the hierarchy changes. If a class changes, you must change at most one table. Unfortunately, the overhead of doing multi-table joins can become a problem if you have even a moderately deep hierarchy.

Solution

(When speed of queries is more important)

Create one table for each concrete subclass of your hierarchy that contains ALL of the attributes defined in that subclass or inherited from its super classes. An instance is retrieved by querying that table.

Discussion

This avoids the joins of the previous solution, making queries more efficient. This is also a simple mapping, but has the drawback that if a super class is changed, then many tables must be modified. It is also difficult to infer the object design from the relational schema.

There is a third solution that may be more appropriate in a multiple-inheritance environment, but that does not have much to recommend itself beyond that. It is possible to create a single table that represents ALL of the super class's and subclasses attributes, with SELECT statements picking out only those that are appropriate for each class. Unfortunately, this can lead to a large number of Null's in your database, wasting space.

3.6 Representing Collections in a Relational Database

Problem

How do you represent Collection subclasses in a relational database?

Forces

The first normal form rule of Relational Databases prevents a relation from containing a "Multivalued" attribute, or what we would normally think of in Object terms as a Collection. The kind of 1-N relationships represented in OO languages by collection classes are represented in a very different form in a relational database.

Collection classes in Smalltalk often convey additional information besides the relationship between the objects contained in the collection, and the object that contains the collection. Order, sorting methods, and type of the contained objects are all problems that must be addressed.

Solution

Represent each collection in your object model (where one object class is related to another object class by a 1-N has-a relationship) by a relationship table. The table may also contain additional attributes that address the other issues.

The basic solution involves creating a table that consists of at least two columns, one, which represents the primary key (usually the OID) of the containing object (the object that holds the collection) and another which represents the primary key of the contained objects (the objects held in the collection). Each entry in the table shows a relationship

between the contained object and the containing object. The primary key of the relationship table is comprised of both columns. A third column may be needed which indicates either the class of the object or the table that the object is located in. Collections may contain objects of various classes.

Discussion

There are other possible representations of the 1-N relationships, including back-pointers. Back pointers have the drawback that it is difficult to have an object be contained in more than one collection at the same time when the two collections are contained in different instances of the same class. The simplest, and most common additional information to include in a relationship table is a column that indicates the type of the contained object. This is necessary when a Collection may be heterogeneous. If an Ordered Collection is utilized, and the order is significant, the position of the object in the collection may be stored in an additional column. It must be noted that unless a distinguishing column indicating a position or OID is added to a relation table and made part of its primary key then the basic solution represents a Set, rather than a more general collection, since the key constraint of relational databases prevent a tuple from occurring more than once in the same table.

3.7 Object Identifier (OID)

Problem

How do you represent an object's individuality in a relational database?

Forces

In object-oriented languages, objects are uniquely identifiable. In Smalltalk, an equivalence comparison (==) determines if two objects are exactly identical. This is accomplished through the comparison of their Object Pointers (OOPs) which are uniquely assigned to each object when it is instantiated.

In an environment where objects may become persistent, some way of identifying what particular persistent structure (be it a row in a relational database, or a structure in an OODBMS) corresponds to that object has to be added to the mix. OOPs are reassigned and reclaimed by the system, precluding their use as an object identifier.

Solution

Assign an identifier (an Object IDentifer or OID) to each object that is guaranteed to be unique across image invocations. This identifier will be part of the object, and will be used to identify it during query operations, and update operations.

Discussion

OID's can be generated either internally to your application, or externally. Some relational databases include a sequence number generator that can be used to generate OID's, and it is preferable to use that option when available. OID's only need be unique within a class, as long as some other way of identifying the class of an object is provided by the persistence scheme. OID's are customarily long integers.

If an OID is generated within the application, it is often common to have a table that represents the latest available OID for each class. The table will be locked, queried, updated and unlocked whenever a new OID is required. To improve performance, sometimes an entire block of OID numbers can be acquired at once.

OID's can include type information encoded into the identifier. In this case, it may be more appropriate to use a char or varchar column rather than an integer.

3.8 Foreign-Key Reference

Problem

How do you represent the fact that in an object model an object can contain not only "base datatypes" like Strings, Characters, Integers and Dates, but other objects as well?

Forces

Given that the first normal form (1NF) rule of relational databases specifically excludes a tuple from containing another tuple you must use another representation of an object that can be represented by a legal value that a column can contain.

Solution

Assign each object in your object model a unique OID (See pattern OID). Add a column for each instance variable that contains an object that is not either:

a collection object a "base datatype" In this column, store the OID of object contained in the previous object. If your database supports the feature, declare the column to be a

foreign key to the table that represents the class of object whose OID is stored in that column.

Discussion

This restriction (the 1NF rule) is both strength, and the Achilles' heel of the relational model. When this pattern is used in self-similar objects (i.e., a Person has children, who are also Persons) it is exceedingly difficult to retrieve a tree of connected objects rooted on a single object in a single SQL query.

If you find that the vast majority of columns in your database schema arise from this pattern, you may wish to reconsider the decision to use a relational database as a persistent object store.

3.9 Static Patterns (Object Side)

The previous section discussed the relationships and the definition of class properties as defined in the relational database schema. However, we must also consider the definition of the object model on the client. Foreign Key versus Direct Reference addresses how to best define the relationships of complex objects to be instantiated in the object image.

3.10 Foreign Key versus Direct Reference

Problem

In the domain object model when should you reference objects with a "foreign key" and when should you have direct reference with pointers?

Forces

In general, the object model should closely reflect the problem domain and its behavior. However, the network of objects that support this model can be complex and large. Modeling a large corporation with its numerous organizations and branches, may require hundreds of thousands of objects and multiple levels of objects of different classes.

In object models, objects usually directly reference one another. This make navigation among the object network direct and easier than via foreign-key reference.

Objects can reference other objects by using their foreign keys. When this is the case, the objects must also have methods to dereference the foreign key to get the referenced object. This makes maintaining the object relationships in the object model more complex. If foreign keys are used to reference the objects then more searches and more caches are required to support the accessing methods. However, using the foreign key makes it easier to map the domain objects to the database tables during their instantiation and passivity. Relying on foreign keys alone with the object model can result in recursive relations and may also result in extremely poor performance problems as large collections of objects are needed to represent a complex object.

In many cases, the application simply requires a list of names to peruse in order to locate the object of interest. The number of potential objects in such a list may be in the millions. This puts a heavy strain on the memory requirements of such a system. A great majority of the time the application just requires a foreign key for display and selection purposes. This means keeping the supporting application domain models "light," where they contain only those attributes necessary for display purposes.

Solution

An object model should use direct reference as much as possible. This permits fast navigation over the object structures. Build the object network piece by piece as required using Proxy objects to minimize storage. Make the associations only as complex as necessary. When dealing with large collections or a set of complex objects use foreign keys and names to represent the objects for user interface display and selection. After selection is made, instantiate the complex object depending upon memory constraints and performance.

Discussion

If each domain object maps to a single table then there is probably a table model in the domain object layer. You may be adding complexity to the whole system. If the domain objects have no behavior other then being information holders, you may consider getting them out of the way. Instead, have the application model refer directly to broker objects. This way you do not have an object cache to keep in sync with the relational tables. If domain behavior is required (which it probably will be) then you can add domain objects as required. Make the domain objects "prove" themselves. In reference to using foreign keys within the object model instead of direct references, one developer

learning Smalltalk said: "What the hell good is objects if you do not hold real objects? You might as well use PowerBuilder."

3.11 Using Patterns in Order Management Systems: A Design Patterns Experience Report

The Problem

Our particular project was for a major pharmaceutical company's IS department. Our team was tasked with building an order management system to be used by employees of the company in placing orders for consumable resources. The system is intended to allow employees to order resources by selecting the type, subtype, and vendor of the resource as well as the delivery date, and various other delivery details. The user is allowed to change orders after they are placed, to view the unconsumed resources that are still allocated to him, and to transfer unconsumed resources to other users.

The Constraints

- Our design faced several constraints:

- We had to deliver a workable application within 3 1/2 months from the inception of the project.
- There was a limited amount of OO and Smalltalk knowledge in our group. Our team consisted of me, one team member that had 6 months of OO experience (having been through a KSC Smalltalk Apprentice Program (STAP)) and three team members with only minimal Smalltalk training and no formal OOA&D training.
- We were required to use an existing Oracle database as our repository. We were free to use any appropriate object design, but it had to work with the existing database tables and Oracle Forms applications.

Solution

As the chief architect of this project, I had two things working for me as I began. The first was that we had earlier prototyped a subset of this application in an apprentice program, so we had a good feel for what objects were involved in the system. The second was that I had just finished developing a tutorial for Smalltalk Solutions '95 that

drew heavily from , so I was very familiar with those patterns. The two factors converged to let me begin the design process by picking out some appropriate patterns that I felt would be useful in this domain, and then letting the developers discover how these patterns could be applied to our specific design.

3.11.1 The Patterns

From my previous experience in this domain, I knew immediately that two patterns from would be useful; State and Memento. At the start of the design process, I described the patterns to the group and then went on to develop a solution utilizing them. Later in the design process, problems came up that were well described by Composite, Mediator, and Adapter as will be shown in sections 4.4 and 4.5. Finally, two rules-of-thumb that were used in this project were phrased (after the fact) as patterns.

3.11.2 State

One of the more common problems found in many MIS systems is the idea of a workflow. In a workflow objects move from person to person within a workgroup, with each person changing, annotating, or modifying the object before it is passed along to the next person. This project was no exception. The analysis of the project done before the design phase of the project begn described the workflow of the various types of Orders in some detail. After reviewing this, I determined that the workflow could be described as a state machine, with different submissions and modifications of the order describing the transitions between states. The states Orders can occupy are shown in the below Figure

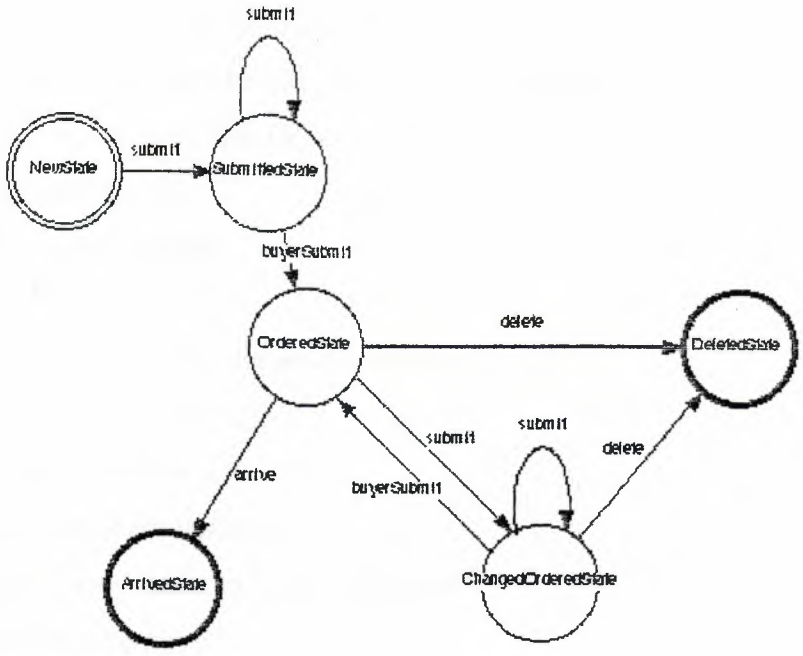


Figure 3.11.2: Order States [http://www.dbmsmag.com]

Once we had determined to represent workflow of an order as a finite state machine, the design of a significant part of our Order object "fell out" of the State pattern. We determined that an Order could be in several different states, depending upon where within the workflow it resided. We also determined that the Order should behave differently to the common messages save, delete, and notify depending upon its state. For example, when an order was in Submitted state, it was known (so far) only to the person placing the order; the buyer (the next person in the workflow) had not yet reviewed it. A delete message sent to the Order should physically remove it from the database when it is in this state. On the other hand, if an Order is in Ordered state, then a delete message should only log the fact that that particular order has been removed. A deletion in this state will necessitate the buyer resubmitting a corrected VendorOrder to the vendor.

Likewise, when an Order is in New state, the buyer should be notified (by E-Mail) when the Order receives the submit message and moves to Submitted state. A different notification should be sent out when a change is made to an Ordered order. Once an order is in ChangedOrdered state, no more notifications are necessary.

We were able to use the following design to represent the state machine portion of our domain model (see Figure 2: State design). Just as described in , we used an abstract

class State that implemented the messages **notifyWith:**, **saveWith:** and **deleteWith:**, the argument to each of these messages being the ConsumablesOrder. Each of the messages in ConsumablesOrder that differed by state were implemented by calling the corresponding message in the Order's current state. For example, let's look at the following implementation:

(ConsumablesOrder>>delete)

delete

"do whatever is appropriate for your current state"

self currentState deleteWith: self.

(SubmittedState>>deleteWith:)

deleteWith: aConsumablesOrder

"tell your consumables order to remove himself"

aConsumablesOrder remove.

(OrderedState>>deleteWith:)

deleteWith: aConsumablesOrder

"tell this consumablesOrder to become deleted (i.e. record the fact in the database)"

aConsumablesOrder becomeDeleted.

(DeletedState>>deleteWith:)

deleteWith: aConsumablesOrder

"anOrder is already deleted. Do nothing"

^self

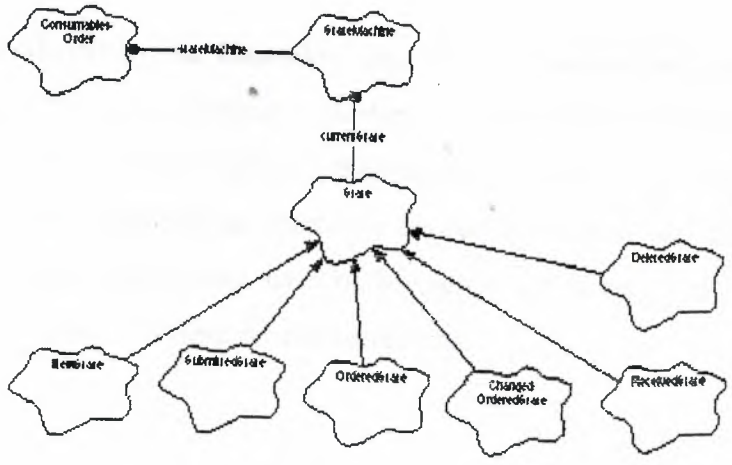


Figure 3.11.2 State design [http://www.dbmsmag.com/]

The only substantial difference from our design to the design from was the addition of a StateMachine object between the ConsumablesOrder (the Context) and the State. In retrospect, we could probably have done without this object. It was only used to aid in construction of the State connections, and for error handling in the case where a transition wasn't defined in the current state. This responsibility could easily have been absorbed into the abstract State class.

The State pattern was the big success story in this application. Its use cut through a lot of complexity in the domain that would otherwise have been handled by several conditional branches spread throughout the code. The pattern was easy to explain to the developers, and the implementation was quick and painless. It proved to be easily extensible (when we began implementing we only knew about **deleteWith:** and **notifyWith:** -- **saveWith:** was added later) and flexible.

3.11.3 Memento

Going into the design of this application, we were aware that we would need to support one-level undo for a ConsumablesOrder. For instance, if a buyer rejects a change to an Order, then the order should revert back to the state it was previously in (Ordered) and all of the changes should be erased. We felt intuitively that the correct solution would be a variant of the Memento pattern, and we discussed possible implementations during the early design sessions, starting with the design example presented in . However, in contrast to how easily we adopted the State pattern, adopting Memento proved to be more challenging.

We were a bit confused by the Caretaker object in the pattern being external to the Originator object, although reviewing it further did clarify it a bit. In our case, there were no external clients of the Originator that needed to know about the existence of a memento, and only one copy of the memento needed to exist at any time. We finally assumed that this was a degenerate case of Memento not covered in . Our resulting design is shown in Figure 3: Memento class structure.

Figure 3.11.3: Memento class structure [http://www.dbmsmag.com]

We rolled the Originator and Caretaker objects into a single object, the ConsumablesOrder. The basic flow of messages, and the structure of the classes, is the same as in once this change is made. When an outside object sends a message to a ConsumablesOrder that would change its state, it issues itself a makeMemento message. This creates the memento and sets its state appropriately (this is done all at once by a deepCopy message). Whenever an outside object sends a message to a ConsumablesOrder that would necessitate reverting back to its original state (such as cancelChanges) it sends itself the revert message, which resets the state to the stored previous state by copying all of the values of the instance variables in the memento back into the original order.

This case was unique in that the Memento pattern did no provide us the solution directly, but led us to an acceptable solution that fit our requirements. Even though the particular solution provided by the pattern's example code didn't work for us, the thought process we went

Through in trying to use the pattern did lead us to an acceptable solution.

3.11.4 Composite

After getting more deeply into our design and implementation, we realized that an unforeseen client requirement was easily solved by application of another pattern, Composite. In our initial design we identified three subclasses of the class Resource:

- OrderResource -- this represents a resource that has been ordered, but not received. It is sort of a "virtual" resource, and doesn't share many of the attributes (disposition, receivedDate, etc.) of an "actual" resource.

- IndividualResource -- this represents a specific, received resource of a certain type. Some resources are tracked individually, with specific ID numbers. A Chair, or a Forklift, or something of this sort is an IndividualResource.
- GroupedResource -- a grouped resource is a set of resources that are not uniquely identifiable. For instance, a bag of bolts might be considered a GroupedResource in that it contains several bolts, but each bolt is not important enough to represent individually. However, the entire bag is interesting enough to track.

In our original design, the user was shown a list of IndividualResources and GroupedResources that were allocated to them. In subsequent user interviews, it came to our attention that the users would prefer to see all of the IndividualResources that were ordered from the same order as a single line item in this list, then drill-down to see the Individual resources. After some consideration, we decided the easiest way to achieve this was to use the Composite pattern, and refactor the hierarchy to create some new classes. First, we divided Resource into two classes, AbstractResource, which defined a resource's protocol, but not its implementation, and ActualResource, which defined the implementation used by the preexisting Resource subclasses. We then defined one more class:

- CompositeResource -- a CompositeResource is a subclass of AbstractResource that responds to the same protocol as an ActualResource, but which is implemented quite differently. It contains a collection of IndividualResources, and implements its protocol by passing through many of its messages to a representative element of that collection. A CompositeResource can answer its type, subtype, etc. just as can an instance of a subclass of ActualResource.

The full Resource hierarchy is shown in Figure 4: Resource Hierarchy.

The great thing about using Composite was that our user interface code did not change at all when we refactored the hierarchy. Since a CompositeResource responded to the same protocol as an IndividualResource or a GroupedResource, the display logic was identical. We were able to easily add new drill-down capabilities through additional UI code that was specific to CompositeResources.

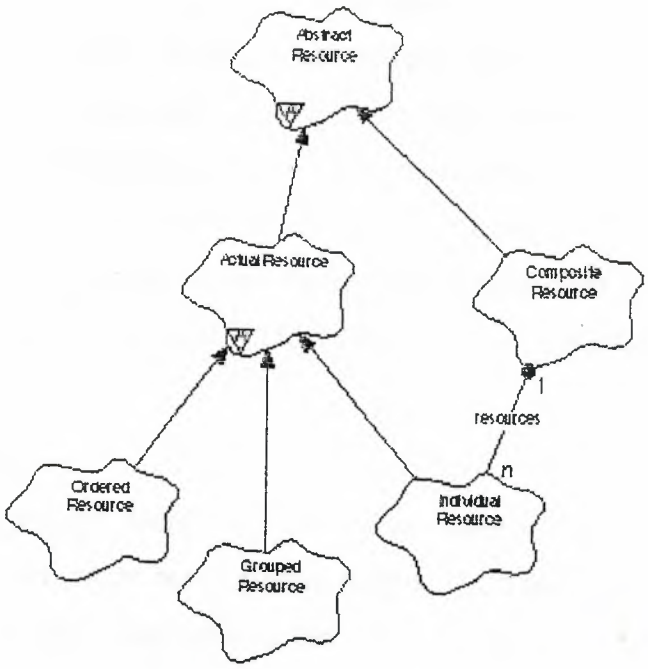


Figure 3.11.4: Resource Hierarchy [http://www.dbmsmag.com/]

An important lesson learned through the application of this pattern was that the interface of an object is different than its implementation. One of the programmers really struggled with why we were refactoring the hierarchy and separating the interface (in AbstractResource) from its implementation (in CompositeResource and ActualResource). The "light came on" in this programmer's mind after we had roughed out the first iteration of code for the new hierarchy and then started up the user interface without having modified any UI code. This was a key lesson in OO design in that for the first time the programmer realized what it meant for a class to be abstract, and why abstract superclasses were useful.

3.11.5 Mediator and Adapter

Our use of these two patterns was more simplistic than the other patterns. In our target language, Smalltalk/V, the ViewManager class provides a Mediation interface between its component SubPanes. It also serves as an Adapter between the SubPanes and the objects of the domain model . The two patterns were used more in spirit than in fact. Whenever any code was written in a ViewManager subclass it was carefully reviewed to see if it fulfilled either the role of Mediator or Adapter. Any code that attempted to do something other than coordinate SubPane display, or adapt SubPane events to domain model methods was rejected in the code review as violating our rules. As an example, at

one point a programmer was planning to place some Unit of Measure conversion code in a specific ViewManager. After a code review, she agreed that this was neither mediating between SubPanes, nor adapting to the domain model. She then developed a more general UnitOfMeasure class for handling the conversions, and wrote only enough code in the ViewManager to adapt this class to the input and output methods of the SubPanes. This allowed her to extend the UnitOfMeasure class to handle similar, but unforeseen cases later in the project without changing the ViewManager code.

3.12 Other Patterns

In developing this system, there were two more "rules of thumb" that we followed during the design, that, while not in pattern form at the time of the development, were patternizable after the fact. Each solution had all the earmarks of a pattern:

- It was a solution to a general problem within a set of constraints
- It had been used several times in other projects
- It was easily explainable in a few sentences

All that remained was for the solution to be written in pattern form. The description of the heuristics we used follows. I have since rewritten them in pattern form, and used them as part of "Crossing Chasms" a pattern language for object to relational database interface design.

3.12.1 Errors as Objects

In a previous project I had seen an interesting way of separating concerns in the ViewManager classes from domain-layer considerations with respect to errors. In this approach, domain validations (range checks, type checks, etc.) were done in the domain, and the results were passed back to the ViewManager as an ErrorSet. In this way you could distribute the responsibility for validation among several objects, with the error set being passed around and added to whenever a validation failed. This design preserved model and view separation, and allowed the user to intervene in the handling of recoverable errors.

When the top-level message returned, the ErrorSet was displayed by the UI, and each Warning (which represents a potentially recoverable error) was flagged as to whether or

not it was proceedable. The entire ErrorSet was then passed back to the domain, which used it to determine if it should allow the next action.

3.12.2 Broker

A second design heuristic that we used was the concept of a Database broker. A Broker acts as an Adapter between a persistent domain object and the classes that represent the physical database and the query language. It translates "domainish" requests into "databasish" queries and helps in mapping SQL rows and columns to objects and instance variables. It provides a needed separation of concerns that isolate the domain classes from the purely database-oriented classes. This architecture allowed us to meet our requirement that we use the existing Oracle tables, while at the same time freeing us to use a fully OO design in our domain classes.

While these solutions were not written down as patterns when we were designing our system, I nevertheless presented them to the developers just as I had presented the patterns from . This process of explaining them in this way helped immensely when I sat down to write them in pattern form later.

3.13 The Type Object Pattern

Intent

Decouple instances from their classes so that those classes can be implemented as instances of a class. Type Object allows new "classes" to be created dynamically at runtime, lets a system provide its own type-checking rules, and can lead to simpler, smaller systems.

Motivation

Sometimes a class requires not only an unknown number of instances, but an unknown number of subclasses as well. Although an object system can create new instances on demand, it usually cannot create new classes without recompilation. A design in which a class has an unknown number of subclasses can be converted to one in which the class has an unknown number of instances.

Consider a system for tracking the videotapes in a video rental store's inventory. The system will obviously require a class called "Videotape." Each instance of **Videotape** will represent one of the videotapes in the store's inventory. However, since many of the videotapes are very similar, the Videotape instances will contain a lot of redundant information. For example, all copies of *Star Wars* have the same title, rental price, MPAA rating, and so forth. This information is different for *The Terminator*, but multiple copies of *The Terminator* also have identical data. Repeating this information for all copies of *Star Wars* or all copies of *The Terminator* is redundant.

One way to solve this problem is to create a subclass of **Videotape** for each movie. Thus two of the subclasses would be **StarWarsTape** and **TerminatorTape**. The class itself would keep the information for that movie. So the information common to all copies of *Star Wars* would be stored only once. It might be hardcoded on the instance side of **StarWarsTape** or stored in variables on the class side or in an object assigned to the class for this purpose. Now **Videotape** would be an abstract class; the system would not create instances of it. Instead, when the store bought a new copy of *The Terminator* videotape and started renting it, the system would create an instance of the class for that movie, an instance of **TerminatorTape**.

This solution works, but not very well. One problem is that if the store stocks lots of different movies, Videotape could require a huge number of subclasses. Another problem is what would happen when, with the system deployed, the store starts stocking a new movie-perhaps *Independence Day*. There is no **IndependenceDayTape** class in the system. If the developer did not predict this situation, he would have to modify the code to add a new IndependenceDayTape class, recompile the system, and redeploy it. If the developer did predict this situation, he could provide a special subclass of **Videotape**--such as **UnknownTape**--and the system would create an instance of it for all videotapes of the new movie. The problem with **UnknownTape** is that it has the same lack of flexibility that **Videotape** had. Just as Videotape required subclasses, so will **UnknownTape**, so **UnknownTape** is not a very good solution.

Instead, since the number of types of videotapes is unknown, each type of videotape needs to be an instance of a class. However, each videotape needs to be an instance of a type of videotape. Class-based object languages give support for instances of classes, but they do not give support for instances of instances of classes. So to implement this

solution in a typical class-based language, you need to implement two classes: one to represent a type of videotape (Movie) and one to represent a videotape (Videotape). Each instance of **Videotape** would have a pointer to its corresponding instance of **Movie**.

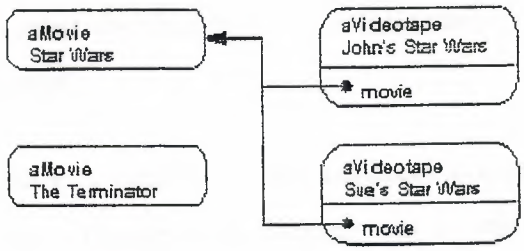


Figure 3.13 {a}

This class diagram illustrates how each instance of **Videotape** has a corresponding instance of **Movie**. It shows how properties defined by the type of videotape are separated from those which differ for each particular videotape. In this case, the movie's title and how much it costs to rent are separated from whether the tape is rented and who is currently renting it.

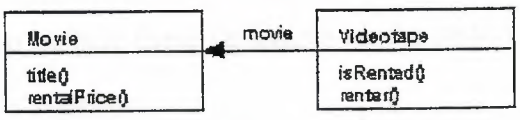


Figure: 3.13 {b}

This instance diagram shows how there is an instance of **Movie** to represent each type of videotape and an instance of **Videotape** to represent each video the store stocks. *Star Wars* and *The Terminator* are movies; videotapes are the copy of *Star Wars* that John is renting versus the one that Sue is renting. It also shows how each **Videotape** knows what type it is because of its relationship to a particular instance of **Movie**.

If a new movie, such as *Independence Day*, were to be rented to Jack, the system would create a new **Movie** and a new **Videotape** that points to the Movie. The movie is *Independence Day* and the tape is the copy of **Independence Day** that Jack ends up renting.

Videotape, **Movie**, and the is-instance-of relationship between them (a **Videotape** is an instance of a **Movie**) is an example of the Type Object pattern. It is used to create instances of a set of classes when the number of classes is unknown. It allows an application to create new "classes" at runtime because the classes are really instances of a class. The application must then maintain the relationship between the real instances and their class-like instances.

The key to the Type Object pattern is two concrete classes, one whose instances represent the application's instances and another whose instances represent types of application instances. Each application instance has a pointer to its corresponding type.

Keys

A framework that incorporates the Type Object pattern has the following features:

- Two classes, a type class and an instance class.
- The instance class has an instance variable whose type is the type class.
- The instance class delegates its type behavior to the type class via the instance variable.

The framework may also include these variations on the pattern:

- The system may maintain a list of its type class instances.
- The type class instances may maintain a list of their instances.

Applicability

- Use the Type Object pattern when:
- Instances of a class need to be grouped together according to their common attributes and/or behavior.
- The class needs a subclass for each group to implement that group's common attributes and behavior.
- The class requires a large number of subclasses and/or the total variety of subclasses that may be required is unknown.
- You want to be able to create new groupings at runtime that were not predicted during design.

- You want to be able to change an object's subclass after its been instantiated without having to mutate it to a new class.
- You want to be able to nest groupings recursively so that a group is itself an item in another group.

3.14 Structure

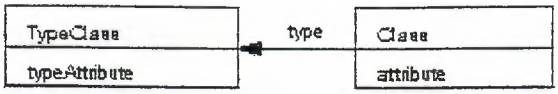


Figure: 3.14 {a}

The Type Object pattern has two concrete classes, one that represents objects and another that represents their types. Each object has a pointer to its corresponding type.

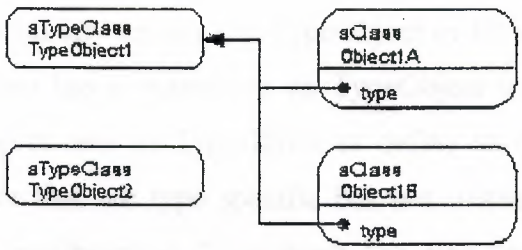


Figure: 3.14 {b}

For example, the system uses a TypeObject to represent each type in the system and an Object to represent each of the instances of those TypeObjects. Each Object has a pointer to its TypeObject.

Participants

- TypeClass (Movie)
  - is the class of TypeObject.
  - has a separate instance for each type of Object.
- TypeObject (*Star Wars, The Terminator, Independence Day*)
  - is an instance of TypeClass.

    - represents a type of Object.
    - Establishes all properties of an Object that are the same for all Objects of the same type.
- Class (**Videotape**)
    - is the class of Object.
    - represents instances of TypeClass.
- Object (John's *Star Wars*, Sue's *Star Wars*)
    - is an instance of Class.
    - represents a unique item that has a unique context.
    - Establishes all properties of that item that can differ between items of the same type.
    - has an associated TypeObject that describes its type.
    - Delegates properties defined by its type to its TypeObject.

TypeClass and Class are classes. TypeObject and Object are instances of their respective classes. As with any instance, a TypeObject or Object knows what its class is. In addition, an Object has a pointer to its TypeObject so that it knows what its TypeObject is. The Object uses its TypeObject to define its type behavior. When the Object receives requests that are type specific but not instance specific, it delegates those requests to its TypeObject. A TypeObject can also have pointers to all of its Objects.

Thus **Movie** is a TypeClass and **Videotape** is a Class. Instances of Movie like *Star Wars*, *The Terminator*, and *Independence Day* are TypeObjects. Instances of **Videotape** like John's *Star Wars* and Sue's *Star Wars* are Objects. Since an Object has a pointer to its jTypeObject, John's videotape and Sue's videotape have pointers to their corresponding Movie, which in this case is *Star Wars* for both videotapes. That is how the videotapes know that they contain *Star Wars* and not some other movie.

Collaborations

- An Object gets two categories of requests: those defined by its instance and those defined by its type. It handles the instance requests itself and delegates the type requests to its TypeObject.

- Some clients may want to interact with the TypeObjects directly. For example, rather than iterate through all of the Videotapes the store has in stock, a renter might want to browse all of the Movies that the store offers.
- If necessary, the TypeObject can have a set of pointers to its Objects. This way the system can easily retrieve an Object that fits a TypeObject's description. This would be similar to the **allInstances** message that Smalltalk classes implement. For example, once a renter finds an appealing Movie, he would then want to know which videotapes the store has that fit the description.

Consequences

The advantages of the Type Object pattern are:

- ***Runtime class creation.*** The pattern allows new "classes" to be created at runtime. These new classes are not actually classes, they are instances called TypeObjects that are created by the TypeClass just like any instance is created by its class.
- ***Avoids subclass explosion.*** The system no longer needs numerous subclasses to represent different types of Objects. Instead of numerous subclasses, the system can use one TypeClass and numerous TypeObjects.
- ***Hides separation of instance and type.*** An Object's clients does not need to be aware of the separation between Object and TypeObject. The client makes requests of the Object, and the Object in turn decides which requests to forward to the TypeObject. Clients that are aware of the TypeObjects may collaborate with them directly without going through the Objects.
- ***Dynamic type change.*** The pattern allows the Object to dynamically change its TypeObject, which has the effect of changing its class. This is simpler than mutating an object to a new class. [DeKezel96]
- ***Independent subclassing.*** TypeClass and Class can be subclassed independently.
- ***Multiple Type Objects.*** The pattern allows an Object to have multiple TypeObjects where each defines some part of the Object's type. The Object must then decide which type behavior to delegate to which TypeObject.

3.15 The disadvantages of the Type Object pattern are:

- ***Design complexity***. The pattern factors one logical object into two classes. Their relationship, a thing and its type, is difficult to understand. This is confusing for modelers and programmers alike. It is difficult to recognize or explain the relationship between a TypeObject and an Object. This confusion hurts simplicity and maintainability. In a nutshell: "Use inheritance; it's easier."
- ***Implementation complexity***. The pattern moves implementation differences out of the subclasses and into the state of the TypeObject instances. Whereas each subclass could implement a method differently, now the TypeClass can only implement the method one way and each TypeObject's state must make the instance behave differently.
- ***Reference management***. Each Object must keep a reference to its TypeObject. Just as an object knows what its class is, an Object knows what its TypeObject is. But whereas the object system or language automatically establishes and maintains the class-instance relationship, the application must itself establish and maintain the TypeObject-Object relationship.

Implementation

There are several issues that you must always address when implementing the Type Object pattern:

- ***Object references TypeObject***. Each Object has a reference to its TypeObject, and delegates some of its responsibility to the TypeObject. An Object's TypeObject must be specified when the Object is created.
- ***Object behavior vs. TypeObject behavior***. An Object's behavior can either be implemented in its class or can be delegated to its TypeObject. The TypeObject implements behavior common to the type, while the Object implements behavior that differs for each instance of a type. When the Object delegates behavior to its TypeObject, it can pass a reference to itself so that the TypeObject can access its data or behavior. The Object may decide to perform additional operations before

and after forwarding the request, similar to the way a Decorator can enhance the requests it forwards to its Component [GHJV95, page 175].

- ***TypeObject is not multiple inheritance.*** The Class-not the TypeObject-is the template for the new Object. The messages that Object understands are defined by its Class, not by its TypeObject. The Class' implementation decides which messages to forward to the TypeObject; the Object does not inherit the TypeObject's messages. Whenever you add behavior to TypeClass, you must also add a delegating method to Class before the behavior is available to the Objects.

3.16 There are other issues you may need to consider when implementing the Type Object pattern:

- ***Object creation using a TypeObject.*** Often, a new Object is created by sending a request to the appropriate TypeObject. This is notable because the TypeObject is an instance and instance creation requests are usually sent to a class, not an instance. But the TypeObject is like a class to the Object, so it often has the responsibility of creating new Objects.
- ***Multiple TypeObjects.*** An Object can have more than one TypeObject, but this is unusual. In this case, the Class would have to decide which TypeObject to delegate each request to.
- ***Changing TypeObject.*** The Type Object pattern lets an object dynamically change its "class," the type object. It is simpler for an object to change its pointer to a different type object (a different instance of the same class) than to mutate to a new class. For example, suppose that a shipment to the video store is supposed to contain three copies of *The Terminator* and two copies of *Star Wars*, so those objects are entered into the system. When the shipment arrives, it really contains two copies of *The Terminator* and three copies of *Star Wars*. So one of the three new copies of *The Terminator* in the system needs to be changed to a copy of *Star Wars*. This can easily be done by changing the videotape's Movie pointer from *The Terminator* to *Star Wars*.
- ***Subclassing Class and TypeClass.*** It is possible to subclass either Class or TypeClass. The video store could support videodisks by making another Class

called Videodisk. A new Videodisk instance would point to its Movie instance just like a **Videotape** would. If the store carried three tapes and two disks of the same movie, three Videotapes and two Videodisks would all share the same Movie.

The hard part of Type Object occurs after it has been used. There is an almost irresistible urge to make the TypeObjects more composable, and to build tools that let non-programmers specify new TypeObjects. These tools can get quite complex, and the structure of the TypeObjects can get quite complex. Avoid any complexity unless it brings a big payoff.

Sample Code

Video Store

Start with two classes, Movie and **Videotape.**

```
Object ()
  Movie (title rentalPrice rating)
  Videotape (movie isRented renter)
```

Notice how the attributes are factored between the two classes. If there are several videotapes of the same movie, some can be rented while others are not. Various copies can certainly be rented to different people. Thus the attributes isRented and renter are assigned at the **Videotape** level. On the other hand, if all of the videotapes in the group contain the same movie, they will all have the same name, will rent for the same price, and will have the same rating. Thus the attributes **title, rentalPrice,** and **rating** are assigned at the **Movie** level. This is the general technique for factoring the TypeObject out of the Object: Divide the attributes that vary for each instance from those that are the same for a given type.

You create a new **Movie** by specifying its **title.** In turn, a **Movie** knows how to create a new **Videotape.**

```
Movie class>>title: aString
    ^self new initTitle: aString
Movie>>initTitle: aString
    title := aString
Movie>>newVideotape
    ^Videotape movie: self
Videotape class>>movie: aMovie
```

```
    ^self new initMovie: aMovie
Videotape>>initMovie: aMovie
    movie := aMovie
```

Since Movie is **Videotape**'s TypeClass, **Videotape** has a movie attribute that contains a pointer to its corresponding Movie instance. This is how a **Videotape** knows what its Movie is. The movie attribute is set when the **Videotape** instance is created by **Videotape** class>>movie:.

A **Videotape** knows how to be rented. It knows whether it is already being rented. Although it does not know its price directly, it knows how to determine its price.

```
Videotape>>rentTo: aCustomer
    self checkNotRented.
    aCustomer addRental: self.
    self makeRentedTo: aCustomer
Videotape>>checkNotRented
    isRented ifTrue: [^self error]

Customer>>addRental: aVideotape
    rentals add: aVideotape.
    self chargeForRental: aVideotape rentalPrice
Videotape>>rentalPrice
    ^self movie rentalPrice
Videotape>>movie
    ^movie
Movie>>rentalPrice
    ^rentalPrice
Videotape>>makeRentedTo: aCustomer
    isRented := true.
    renter := aCustomer
```

Thus it chooses to implement its is **Rented** behavior itself but delegates its **rentalPrice** behavior to its Type Object.

When *Independence Day* is released on home video, the system creates a **Movie** for it. It gathers the appropriate information about the new movie (title, rental price, rating, etc.) via a GUI and executes the necessary code. The system then creates the new **Videotapes** using the new **Movie**.

3.17 Video Store-Nested Type Objects

The Type Object pattern can be nested recursively. For example, many video stores have categories of movies-such as New Releases (high price), General Releases (standard price), Classics (low price), and Children's (very low price). If the store wanted to raise the price on all New Release rentals from $3.00 to $3.50, it would have to iterate through all of the New Release movies and raise their rental price. It would be easier to store the rental price for a New Release in one place and have all of the New Release movies reference that one place.

Thus the system needs a **MovieCategory** class that has four instances. The **MovieCategory** would store its rental price and each Movie would delegate to its corresponding **MovieCategory** to determine its price. Thus a **MovieCategory** is the Type Object for a **Movie**, and a **Movie** is the Type Object for a **Videotape**.

A **MovieCategory** class requires refactoring Movie's behavior.

```
Object ()
    MovieCategory (name rentalPrice)
    Movie (category title rating)
    Videotape (movie isRented renter)
```

Before, **rentalPrice** was a attribute of Movie because all **videotapes** of the same movie had the same price. Now all movies in the same category will have the same price, so rentalPrice becomes an attribute of **MovieCategory**. Since **Movie** now has a type object, it has an attribute-**category**-to point to its type object.

Now behavior like **rentalPrice** gets delegated in two stages and implemented by the third.

```
Videotape>>rentalPrice
    ^self movie rentalPrice
Movie>>rentalPrice
    ^self category rentalPriceMovie
Category>>rentalPrice
    ^rentalPrice
```

This example nests the Type Object pattern recursively where each MovieCategory has Movie instances and each Movie has **Videotape** instances. The system still works primarily with Videotapes, but they delegate their type behavior to Movies, which in turn delegate their type behavior to **MovieCategorys**. **Videotape** hides from the rest of the system where each set of behavior is implemented. Each piece of information about a tape is stored in

just one place, not duplicated by various tapes. The system can easily add new **MovieCategorys**, **Movies**, and **Videotapes** when necessary by creating new instances.

3.18 Video Store-Dynamic Type Change

Once *Independence Day* is no longer a New Release, its category can easily be changed to a General Release because its category is a Type Object and not its class.

```
Movie>>changeCategoryTo: aMovieCategory
    self category removeMovie: self.
    self category: aMovieCategory.
    self category addMovie: self.
```

With the Type Object pattern, an Object can easily change its Type Object when desired.

3.19 Video Store-Independent Subclassing

The system could also support videodisks. The commonalities of videotapes and videodisks are captured in the abstract superclass **RentableItem**, where **Videotape** and **Videodisk** are subclasses. Both concrete classes delegate their type behavior to **Movie**, so **Movie** does not need to be subclassed.

```
Object ()
    MovieCategory (name rentalPrice)
    Movie (category title rating)
    RentableItem (movie isRented renter)
        Videotape (isRewound)
        Videodisk (numberOfDisks)
```

Most of **Videotape**'s behavior and implementation is moved to **RentableItem**. Now **Videodisk** inherits this code for free.

Movie may turn out to be a specific example of a more general **Title** class. **Title** might have subclasses like **Movie**, **Documentary**, and **HowTo**. Movies have ratings whereas documentary and how-to videos often do not. How-to videos often come in a series or collection that is rented all at once whereas movies and documentaries do not. Thus **Title** might also need a Composite [GHJV95, page 163] subclass such as **HowToSeries**. Movie itself might also have subclasses like **RatedMovie** for those movies that have MPAA ratings and **UnratedMovie** for movies that don't.

```
Object ()
  MovieCategory (name rentalPrice)
  Title (category title)
    Documentary ()
    HowTo ()
    Movie ()
      RatedMovie (rating)
      UnratedMovie ()
    TitleComposite (titles)
      HowToSeries ()
  RentableItem (title isRented renter)
    Videotape (isRewound)
    Videodisk (numberOfDisks)
```

The code above and the diagram below show the final set of classes in this framework.

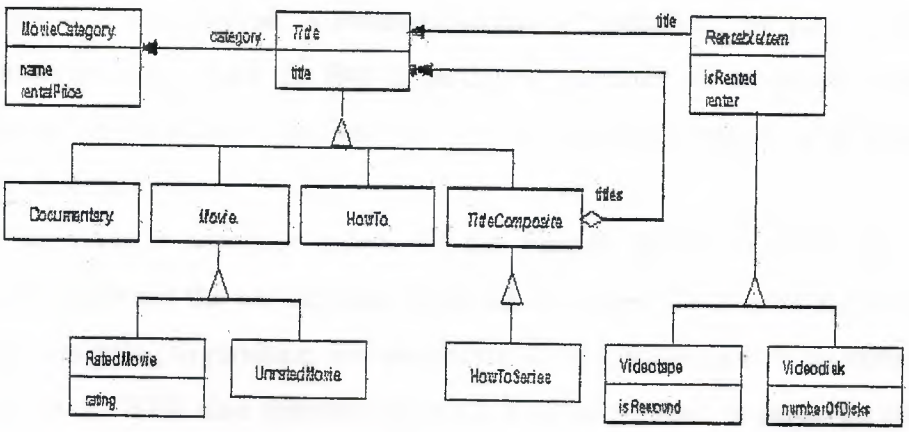


Figure 3.19 {a} [http://www.dbmsmag.com]

Movie and **Title** can be subclassed without affecting the way **RentableItem** and **Videotape** are subclassed. This ability to independently subclass **Title** and **RentableItem** would be impossible to achieve if the **videotape** object had not first been divided into **Movie** and **Videotape** components. Obviously, all of this nesting and subclassing can get complex, but it shows the flexibility the Type Object pattern can give you-flexibility that would be impossible without the pattern.

Manufacturing

Consider a factory with many different machines manufacturing many different products. Every order has to specify the kinds of products it requires. Each kind of product has a list of parts and a list of the kinds of machines needed to make it. One approach is to make a class hierarchy for the kinds of machines and the kinds of products. But this means that adding a new kind of machine or product requires programming, since you have to define a new class. Moreover, the main difference between different products is how they are made. You can probably specify a new kind of product just by specifying its parts and the sequence of machine tools that is needed to make it.

It is better to make objects that represent "kind of product" and "kind of machine." They are both examples of type objects. Thus, there will be classes such as **Machine, Product, MachineType,** and **ProductType.** A **ProductType** has a "manufacturing plan" which knows the **MachineTypes** that make it. But a particular instance of **Product** was made on a particular set of **Machines.** This lets you identify which machine is at fault when a product is defective.

Suppose we want to schedule orders for the factory. When an order comes in, the system will figure out the earliest that it can fill the order. Each order knows what kind of product it is going to produce. For simplicity, we'll assume each order consists of one kind of product. We'll also assume that each kind of product is made on one kind of machine. But that product is probably made up of other products, which will probably require many other machines. Thus, Product is an example of the Composite pattern [GHJV95, page 163] (not shown below). For example, a hammer consists of a handle and a head, which are combined at an assembly station. The wooden handle is carved at one machine, and the head is cast at another. **ProductType** and **Order** are also composites, but are not shown.

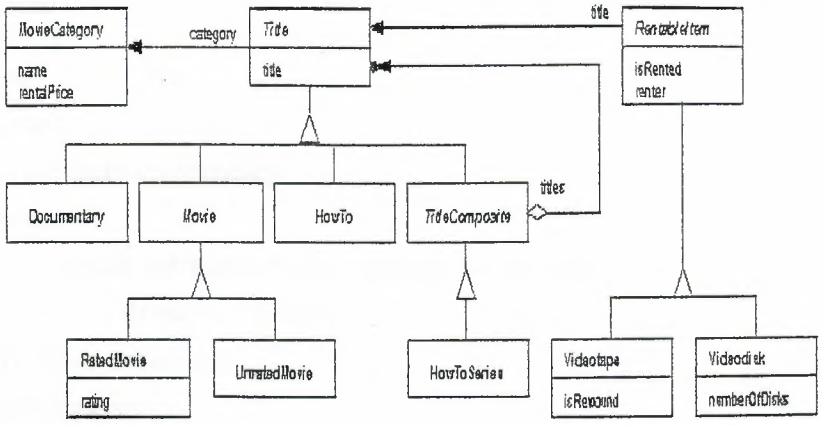


Figure 3.19 {b}[http://www.dbmsmag.com/]

There are six main classes:

Object

- MachineType (name machines)
- Machine (type location age schedule)
- ProductType (manufacturingMachine duration parts)
- Product (type creationDate manufacturedOn parts)
- Order (productType dueDate requestor parts item)
- Factory (machines orders)

We will omit all the accessing methods, since they are similar to those in the video store example. Instead, we will focus on how a factory schedules an order.

A factory acts as a Facade [GHJV95, page 185], creating the order and then scheduling it.

```
Factory>>orderProduct: aType by: aDate for: aCustomer
    | order |
    order := Order product: aType by: aDate for: aCustomer.
    order scheduleFor: self.
    ^order
Order>>scheduleFor: aFactory
    | partDate earliestDate |
    partDate := dueDate minusDays: productType duration.
    parts := productType parts collect: [:eachType |
        aFactory
            orderProduct: eachType
```

```
        by: partDate
        for: order].
    productType
      schedule: self
      between: self datePartsAreReady
      and: dueDate
ProductType>>schedule: anOrder between: startDate and: dueDate
  (startDate plusDays: duration) > dueDate
    ifTrue: [anOrder fixSchedule].
  manufacturingMachine
    schedule: anOrder
    between: startDate
    and: dueDate
```

There are at least two different subclasses of **ProductType**, one for machines that can only be used to make one product at a time, and one for assembly lines and other machines that can be pipelined and so make several products at a time. A non-pipelined machine type is scheduled by finding a machine with a schedule with enough free time open between the **startDate** and the **dueDate.**

```
NonpipelinedMachineType>>schedule: anOrder between: startDate and: dueDate
  machines do: [:each | | theDate |
  theDate := each schedule
              slotOfSize: anOrder duration
              freeBetween: startDate
              and: dueDate.
  theDate notNil ifTrue:
    [^each schedule: anOrder at: theDate]].
  anOrder fixSchedule
```

A pipelined machine type is scheduled by finding a machine with an open slot between the startDate and the dueDate.

```
PipelinedMachineType>>schedule: anOrder between: startDate and: dueDate
  machines do: [:each | | theDate |
  theDate := each schedule
              slotOfSize: 1
              freeBetween: startDate
              and: dueDate.
  theDate notNil ifTrue:
    [^each schedule: anOrder at: theDate]].
```

anOrder fixSchedule

This design lets you define new **ProductTypes** without programming. This lets product managers, who usually aren't programmers, specify a new product type. It will be possible to design a tool that product managers can use to define a new product type by specifying the manufacturing plan, defining the labor and raw materials needed, determining the price of the final product, and so on. As long as a new kind of product can be defined without subclassing Product, it will be possible for product managers to do their work without depending on programmers.

There are constraints between types. For example, the sequence of actual **MachineTools** that manufactured a Product must match the **MachineToolTypes** in the manufacturing plan of its **ProductType**. This is a form of type checking, but it can be done only at runtime. It might not be necessary to check that the types match when the sequence of **MachineTools** is assigned to a Product, because this sequence will be built by iterating over a manufacturing plan to find the available **MachineTools**. However, scheduling can be complex and errors are likely, so it is probably a good idea to double-check that a Product's sequence of **MachineTools** matches what its **ProductType** says it should be.

3.20 Known Uses

Coad

Coad's Item Description pattern is the Type Object pattern except that he only emphasized the fact that a Type holds values that all its Instances have in common. He used an "aircraft description" object as an example. [Coad92]

Hay

Hay uses Type Object in many of his data modeling patterns, and discusses it as a modeling principle, but doesn't call it a separate pattern. He uses it to define types for activities, products, assets (a supertype of product), incidents, accounts, tests, documents, and sections of a Material Safety Data Sheet. [Hay96]

Fowler

Fowler talks about the separate Object Type and Object worlds, and calls these the "knowledge level" and the "operational level." He uses Type Object to define types for

organizational units, accountability relationships, parties involved in relationships, contracts, the terms for contracts, and measurements, as well as many of the things that Hay discussed. [Fowler97]

Odell

Odell's Power Type pattern is the Type Object pattern plus the ability for subtypes (implemented as subclasses) to have different behavior. He illustrates it with the example of tree species and tree. A tree species describes a type of tree such as American elm, sugar maple, apricot, or saguaro. A tree represents a particular tree in my front yard or the one in your back yard. Each tree has a corresponding tree species that describes what kind of tree it is. [MO95]

3.21 Sample Types and Samples

The Type Object pattern has been used in the medical field to model medical samples. A sample has four independent properties:

- The system it is taken from (e.g., John Doe)
- The subsystem (e.g., blood, urine, sputum)
- The collection procedure (aspiration, drainage, scraping)
- The preservation additive (heparin, EDTA)

This is easily modeled as a **Sample** object with four attributes: system, subsystem, collection procedure, and additive. Although the system (the person who gave the sample) is different for almost all samples, the triplet (subsystem, collection procedure, and additive) is shared by a lot of samples. For example, medical technicians refer to a "blood" sample, meaning a blood/aspiration/EDTA sample. Thus the triplet attributes can be gathered into a single **SampleType** object.

A **SampleType** is responsible for creating new **Sample** objects. There are about 5,000 different triplet combinations possible, but most of them don't make any sense, so the system just provides the most common **SampleTypes**. If another **SampleType** is needed, the users can create a new one by specifying its subsystem, collection procedure, and additive. While the system tracks tens of thousands of **Samples**, it only needs to track

about one-hundred **SampleTypes**. So the **SampleTypes** are TypeObjects and the **Samples** are their Objects. [DeKezel96]

Signals and Exceptions

The Type Object pattern is more common in domain frameworks than vendor frameworks, but one vendor example is the Signal/Exception framework in VisualWorks Smalltalk. When Smalltalk code encounters an error, it can raise an **Exception**. The **Exception** records the context of where the error occurred for debugging purposes. Yet the **Exception** itself doesn't know what went wrong, just where. It delegates the what information to a **Signal**. Each **Signal** describes a potential type of problem such as user-interrupt, message-not-understood, and subscript-out-of-bounds. Thus two message-not-understood errors create two separate **Exception** instances that point to the same **Signal** instance. **Signal** is the TypeClass and **Exception** is the Class. [VW95]

Reflection

Type Object is present in most reflective systems, where a type object is often called a metaobject. The class/instance separation in Smalltalk is an example of the Type Object pattern. Programmers can manipulate classes directly, adding methods, changing the class hierarchy, and creating new classes. By far the most common use of a class is to make instances, but the other uses are part of the culture and often discussed, even if not often used. [KRB91]

Reflection has a well-deserved reputation for being hard to understand. Type Object pattern shows that it does not have to be difficult, and can be an easy entrance into the more complex world of reflective programming.

3.22 Related Patterns

3.22.1 Type Object vs. Strategy and State

The Type Object pattern is similar to the Strategy and State patterns [GHJV95, page 315 and page 305]. All three patterns break an object into pieces and the creal objectî delegates to the new object-either the Type Object, the Strategy, or the State. Strategy

and State are usually pure behavior, while a Type Object often holds a lot of shared state. States change frequently, while Type Objects rarely change. State solves the problem of an object needing to change class, whereas Type Object solves the problem of needing an unlimited number of classes. A Strategy usually has one main responsibility, while a Type Object usually has many responsibilities. So, the patterns are not exactly the same, even though their object diagrams are similar.

3.22.2 Type Object and Reflective Architecture

Any system with a Type Object is well on its way to having a Reflective Architecture [BMRSS96]. Often a Type Object holds Strategies for its instances. This is a good way to define behavior in a type.

3.22.3 Type Object vs. Bridge

A Type Object implementation can become complex enough that there are Class and Type Class hierarchies. These hierarchies look a lot like the Abstraction and Implementor hierarchies in the Bridge pattern [GHJV95, page 151], where Class is the abstraction and Type Class is the implementation. However, clients can collaborate directly with the Type Objects, an interaction that usually doesn't occur with Concrete Implementors.

3.22.4 Type Object vs. Decorator

An Object can seem to be a Decorator [GHJV95, page 175] for its Type Object. An Object and its Type Object have similar interfaces and the Object chooses which messages to forward to its Type Object and which ones to enhance. However, a Decorator does not behave like an instance of its Component.

3.22.5 Type Object vs. Flyweight

The Type Objects can seem like Flyweights [GHJV95, page 195] to their Objects. However, Type Object does not involve a Flyweight Factory that provides access to a Flyweight Pool. Nevertheless, two Objects using the same Type Object might think that they each have their own copy, but instead are sharing the same one. Thus it is important that neither Object change the intrinsic state of the Type Object.

3.22.6 Type Object and Prototype

Another way to make one object act like the type of another is with the Prototype pattern [GHJV95, page 117], when each object keeps track of its prototype and delegates requests to it that it does not know how to handle.

3.23 Pattern Language for Relational Databases and Smalltalk

Early in 1995 we (two experienced Smalltalk programmers) began a project in analysis and design that would tax our abstraction abilities to their limits. The result of this ongoing exercise is a pattern language we call Crossing Chasms. This article describes Crossing Chasms as well as exploring the thought processes that led us to write it, what we discovered in its writing, and how we have used the document since its creation.

3.24 What motivated us to write a pattern language?

The business of companies like Knowledge Systems Corporation (KSC) and The Object People is to transfer information about the process of building object systems from consultants to clients. One of the most common themes running through many of the object systems our two companies have built over the past five years is the need to integrate Smalltalk with relational database technology. We have found that the clients of our training and consulting businesses are extremely interested in this area, and often need guidance to understand how these two technologies combine.

In early 1995 we were both involved in creating new material for client-centered mentoring and classroom education. We felt the need to include some information about relational databases, but were uncertain as to how to organize that information. Each of the Smalltalk vendors (Digitalk, Parcplace, and IBM) had their own, unique class libraries for handling relational database queries. On the surface, there did not appear to be much commonality among the three.

Over the past several years we had built many systems using Smalltalk and relational databases with major corporate clients. KSC's first such effort had been with a government organization in early 1992, followed by projects for a national bank, a

major telecommunications company, a telecommunications equipment manufacturer, and a pharmaceutical company. We had learned many lessons about building this kind of system, and had found out what worked and what didn't. Although each system was unique, we felt that there were some commonalties among all of them. In fact, the design for each usually incorporated the best ideas from all the previous ones, even though none of the systems shared any code.

It was this desire to record our lessons learned, to be better equipped for future projects, and to find unity among the disparate vendor implementations, that led us to explore pattern languages as an avenue for recording this design and implementation information. A pattern language is a set of related patterns that guides a reader through a set of closely linked problems and their solutions.

The pattern is a literary form invented by the architect Christopher Alexander to describe the decisions involved in designing and building communities and buildings. The shortest way to describe the essence of a pattern is "A solution to a problem in a context". It records how the interplay of different "forces" on a particular problem can lead to their resolution in a template solution. The pattern form was introduced into the software community by Ward Cunningham and Kent Beck in the early 1980's. It has become popular in recent years due in large part to the work of Gamma, Coad, and others.

We chose to begin writing a pattern language because the pattern form seemed to best capture the spirit of the notions that we had. We felt that a pattern language that could lead readers in a non-linear fashion from one topic to the next could bring together the interconnected threads of thought that we had. It also provides a structure in which to study the issues and their solutions by naming and isolating the essence of each problem. We were also interested in exploring the issues involved in writing patterns -- in this sense Crossing Chasms was an experiment in writing a large pattern language.

3.25 How did we find our patterns?

We first wanted to identify all the issues and problems that arise in designing and building a framework marrying relational databases and Smalltalk.

In reviewing the process of building such a system it became apparent that we could split the set of problems roughly in two. The problems of defining the tables and object

models we categorized as "static" patterns. Those involved in resolving the runtime problems of object-table mapping we put in a category called "dynamic" patterns. We then realized that a number of the problems we were identifying were not so much directly related to the object-table mismatch but were really client-server issues. These problem-solution pairs were generic enough to be applicable to any client-server architecture, object-oriented or not, so we developed a third category ("client-server" patterns) for them.

Lastly we saw that the decision to go with a client-server model was just one fundamental architectural decision out of many. Many other architectural issues must also be resolved, including the modularization of functionality into application layers and the choice of the number of tiers that the system would include. These patterns we termed "architectural" patterns.

Crossing Chasms grew in size and complexity as new problems were identified. To discover the patterns we first immersed ourselves in the literature and subject area. We found our patterns in numerous places. Our own experience in building systems led us to identifying most of the major ones. Studying the documentation of existing frameworks, both commercial and proprietary, added to the list as well. Reading the OO literature that addressed the subject, (Rumbaugh, Jacobson, Gamma, and others) also contributed some patterns to the list, particularly in the static category.

Eventually after defining the basic patterns and formulating them as a pattern language we came up with some new ones based on feedback from our colleagues. This whole process followed the 3 I Paradigm of mastering a subject area. First you Immerse yourself in a field. This leads you to Imitate the solutions of others, until finally you can Innovate and come up with your own solutions.

As mentioned above, Crossing Chasm's patterns are categorized into four groups: architectural, static, dynamic and client-server. In the following sections we will introduce a few of the most important patterns in the language in their respective categories. Unfortunately, we can only present a taste of our language as a whole. Our current version of the language is over 90 pages long and very dense in text and diagrams. We have discovered almost forty patterns, of which we introduce eleven here. The presentations of the individual patterns here are by necessity very brief; the pattern language goes into much more depth in each pattern.

3.26 The Patterns of Crossing Chasms Architectural Patterns

When a project needs to use both Smalltalk and relational technology there are a group of issues at a very high architectural level that need to be addressed. Surprisingly, we did not recognize many of these issues until well after we had written the rest of the patterns in Crossing Chasms. These issues so pervaded our thinking that it took a second look at the problem to even recognize their existence.

One of the most important decisions to make about the design of a system is its overall software architecture. This decision determines the direction that development will take.

3.27 Pattern: Four-Layer Architecture

Problem:

What is the appropriate structure and grouping of classes in a Smalltalk client-server system? What architecture is most appropriate?

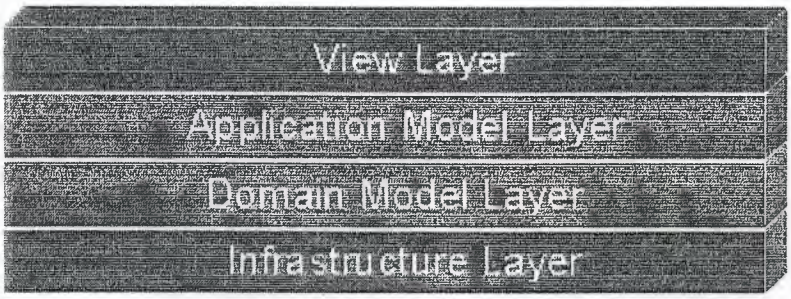


Figure 3.27 Four Layer Architecture

Solution:

Employ a four-layer architecture consisting of a view layer, an application model layer, a domain layer, and a supporting infrastructure layer (see Figure 1: Four Layer Architecture). Determine the interfaces between the layers well ahead of time and keep the communications paths well defined. Enforce the layering through design and code reviews.

Layered architectures are a well-known idea in Computer Science, but it is rare that new Smalltalk programmers see their designs in terms of well-defined layers.

Nevertheless, proper layering is important for reusability and maintainability. Brown [96] deals with this issue at length.

Another key decision that has to be made is the order in which development events must occur. It is especially difficult for first-time users of Object Technology to develop an ordered development process. After seeing several bad decisions made in projects we had observed, we recognized this pattern in retrospect.

3.28 Pattern: Table Design Time

Problem

When is the best time to develop your relational database schema? In what order do object design and schema design occur?

Solution:

Design the relational database schema based upon a first-pass object model done using a behavioral modeling technique. It may be more prudent to wait until after an architectural prototype has been built before designing the schema (see Figure 2: Development Lifecycle). Remember that an OO design is in reality a first-pass database design. Doing things in the reverse order (schema first) often lead to a poorly factored OO design with separate "function" and "data" objects.

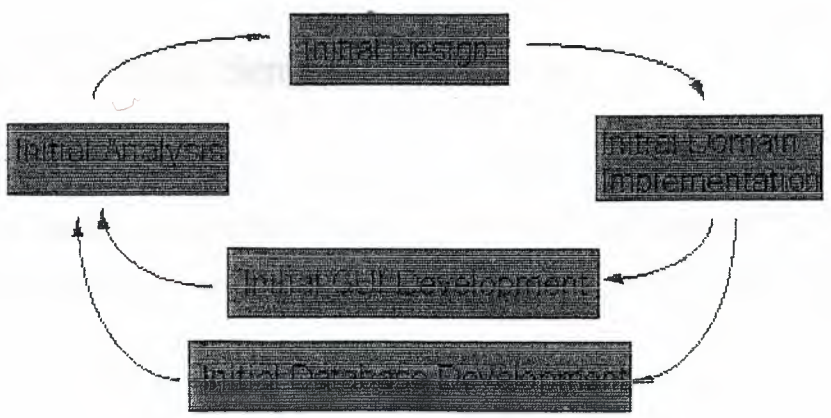


Figure 3.28 Development Lifecycle

One of the fundamental problems in developing a total enterprise solution using Object Technology is the development of relational database schemas from object models. We were lucky in finding that this is a well-represented area of research that had been covered well over the past several years. Our job in developing the static patterns was to pick the "best of breed" of the available approaches and integrate them into a complete, self-consistent method.

3.29 Pattern: Representing objects as tables

Problem:

How do you map a set of objects into a relational database schema? Considering that complex objects do not map neatly into tables, objects do not have keys, tables do not have identity, and the datatypes do not match between worlds, how do you perform a mapping?

Solution:

Start with a table for each persistent object. Determine the "type" of each instance variable and create a column for each that have "base" datatypes. Use the Representing Collections pattern to handle collections. Use the Foreign Key reference pattern to handle other non-base datatype objects. Finally, use the Object Identifier pattern.

3.30 Pattern: Object Identifier

Problem:

How do you preserve an object's identity in a relational database? Each individual object's identity must be preserved in the database and there should be no spurious duplicates.

Solution:

Assign an independent identifier (called an Object Identifier, or OID) to each persistent object. An easy way to do this is to use a sequence number generator if one is available

in your particular database. If not, an OID table can be used. OIDs are usually simply long integers that are guaranteed to be unique for a particular class of objects.

3.31 Pattern: Foreign Key Reference

Problem:

How do you represent objects that reference other objects that are not "base datatypes"? The First Normal Form Rule (1NF) excludes tuples from containing other tuples; therefore Object relationships must be represented using only legal column values.

Solution:

Assign each object a unique OID. You then add a column for each instance variable that is not a base datatype or a collection. In that column store the OID of the referenced object, then declare the column as a foreign key.

3.32 Pattern: Representing Collections

Problem:

How do you represent Smalltalk collections in a relational database? The first normal form rule of relational databases forbids tuples from containing sets of other elements. Other properties of Smalltalk collections also prove bothersome. For instance, objects may be contained in many collections (M-N relationships). Also, collections have special properties (sort order, duplicates). Finally, Smalltalk collections can be either heterogeneous or homogenous

Solution:

Create a relationship table for each collection. A relationship table maps the primary keys of the containing objects to the primary keys of the contained objects. The relationship table may store other information as well, for instance the class of contained object, or the position of object (OrderedCollection, SortedCollection). If a collection is heterogeneous, then the class of each element is also stored in that table.Other static patterns in Crossing Chasms dealt with the issues of representing

why Brokers are important. However, they are central to maintaining the integrity of the layers in a 1 Four-Layer Architecture.

As we looked back on the broker implementations we had built, we found that two more patterns occurred in the best implementations; *Query Object* and *Object Metadata*.

3.35 Pattern: Object Metadata

Problem:

How do you define the mapping between the elements of an object class and the corresponding parts of a relational schema?

Solution:

Reify the mapping into a set of Map classes that map object relationships into relational equivalents. Map objects also map column names to instance variable selectors in domain objects.

3.36 Pattern: Query Object

Problem:

How do you handle the generation and execution of common SQL statements and minimize the amount of duplicated code between broker classes?

Solution:

Write a set of generic classes that generate SQL statements from common data. A hierarchy of classes representing SQL statements can generate the appropriate SQL given a domain object and its Map object metadata representation.

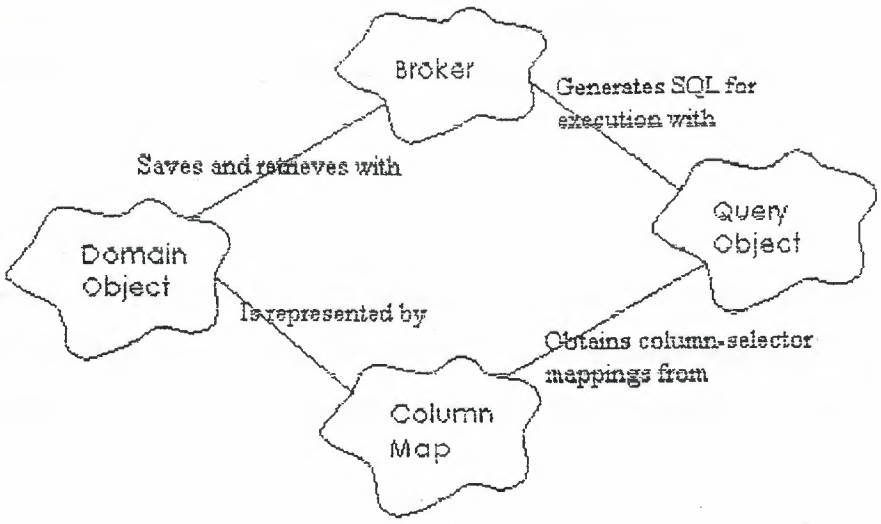


Figure 3.36: Broker Interactions [Ms Access]

The three previous patterns, when combined, make up a powerful mini-architecture. Each domain object will have a set of Map objects that represent its object relationships as metadata. The Broker classes that are responsible for saving and restoring those objects can use Query Objects to generate the appropriate SQL statements from the data held in the Maps. In this way, proper layering can be preserved since the objects in the Domain layer are not directly knowledgeable about the internals of the SQL generation, while the Brokers themselves obtain information about their domain classes only indirectly through the Map objects. A diagram of the interactions of these classes is shown in Figure 3: Broker Interactions.

While the Broker architecture worked well to allow us to move objects in and out of the database, the performance of some of our early attempts was less than adequate. In particular, early versions often spent too much time reading in data from the database that was never subsequently used. In trying to resolve this, we found that the 1 Proxy pattern from Gamma provided us with an effective solution. We could often use a Proxy as a placeholder for information that had not yet been read in from the database. When that information was needed, the Proxy would collaborate with the Broker to read it in, and then replace itself with the new object.

Other topics addressed by the dynamic patterns included handling database transactions and the order in which connected objects must be written to or restored from the database.

Client-Server Patterns

As we mentioned previously, there were many issues we discovered that were not specific to Smalltalk, or even OO in general, but were rather applicable to any client-server systems. Two of these patterns were 1 Client Synchronization and 1 Cache Management.

3.37 Pattern: Client Synchronization

Problem:

How do you handle resynchronizing the client image and database when there are errors? What do you do if you change the value of data held in the client's memory and the corresponding request to the database fails?

One solution is to just note the error to the user and flush any cached information. In this case any error is deemed to be catastrophic and you must start a new session. This is not a very robust solution, but it is a quickly implement able one.

A second solution is a playback mechanism that has a logging facility. Each change is logged in a local log. If there is a failure the cache is flushed and you replay of all the events as needed. This solution is more robust, but it is not trivial to implement.

Solution:

Mark the objects appropriately as deleted, added or updated during the session. If the update to the database succeeds then remove the mark. If it fails then retry the transaction. If it continually fails (e.g., times out) note the error and flush the cache. With the changed objects marked it is possible to recover to the original state by filing out the changed objects to local storage and performing recovery at a more propitious time.

3.38 Pattern: Cache Management

Problem:

How do you best manage the lifetime of persistent objects stored in an RDB? Caches can increase client performance, but they also increase client memory size. Caches can become out of date, necessitating frequent updates. Caching also generally increases application complexity.

Solution:

Use a Session object that has a bounded lifetime and is responsible for identity cache management of a limited set of objects. Balance speed vs. space by flushing the cache as appropriate. Use a query before write (timestamp) technique to keep caches accurate.

How have we used Crossing Chasms?

Since writing Crossing Chasms we have successfully applied its patterns in a number of different instances. It has proven to be a very useful teaching aid - we subsequently have developed several lectures for classroom use from the pattern language. The structure of the pattern language proved to be a useful framework for discussing the different concepts in object to relational connectivity. The topics of the lectures we developed from the pattern language paralleled the organization of the language. In addition, some of the patterns have been used as a basis for other lecture topics in our classroom education. We have also found that students like having the pattern language as an after-the-fact reference after seeing presentations based on it. In this way, we can present a high-level overview and then allow the students to investigate the deeper issues at their own pace.

Several companies have used the patterns in Crossing Chasms as part of their object-relational architectures as a result of our presenting them as part of our training. We have found that addressing the issues covered in Crossing Chasms early in the development process can preclude many of the missteps that first OO projects often take.

We have also developed a conference tutorial based upon the pattern language and presented it at Smalltalk Solutions '96 in New York. Again, we have had feedback that

students appreciate using the pattern language to gain deeper understanding of the issues after the presentation.

The static portion of Crossing Chasms was presented at the Plop (Pattern Languages of Programs) '95 conference in September 1995. Those patterns have been published in Brown [96].

3.39 Crossing Chasms: The Architectural PatternsPATTERN NAME: FOUR LAYER ARCHITECTURE

Problem:
When designing an object system for a client-server environment what is the most appropriate way to structure the overall application architecture?

Forces:
When designing the software architecture in a client-server system, you must come up with a way to divide the labor among team members. Your architecture must also be simple enough to be easily explainable to new team members, so they can understand where their work fits.

In looking for application architecture, many developers have looked to the pioneering MVC architecture. However, MVC is not the be-all and end-all of object design. While a proper architecture should address the concerns addressed by MVC, and may trace its descent from MVC, modern software systems must also address issues not covered by classic MVC.

MVC promoted reuse by factoring out the UI widget away from the domain objects. Modern class libraries derived from MVC have also discovered yet another set of potentially useful and reusable abstractions in separating out the aspect of mediating between views and adapting views to domain models into another set of classes.

However, this still does not address the connection of the domain to the outside world (i.e., object persistency mechanisms, network protocols, etc.). A complete architecture for client-server systems must address these issues as well. Therefore:

Solution:

Factor your application classes into four layers in the following way (see Figure 1: Four Layer Architecture):

- **The View layer**. This is the layer where the physical window and widget objects live. It may also contain Controller classes as in classical MVC. Any new user interface widgets developed for this application are put in this layer. In most cases today this layer is completely generated by a window-builder tool.

- **The Application Model layer**. This layer mediates between the various user interface components on a GUI screen and translates the messages that they understand into messages understood by the objects in the domain model. It is responsible for the flow of the application and controls navigation from window to window. This layer is often partially generated by a window-builder and partially coded by the developer.

- **The Domain Model layer**. This is the layer where most objects found in an OO analysis and design will reside. To a great extent, the objects in this layer can be application-independent. Examples of the types of objects found in this layer may be Orders, Employees, Sensors, or whatever is appropriate to the problem domain.

- **The Infrastructure layer**. This is where the objects that represent connections to entities outside the application (specifically those outside the object world) reside.

Discussion:

This choice of layers can have many beneficial effects on your application if it is applied in the proper way. First, since the architecture is so simple, it is easy to explain to team members and so demonstrate where each object's role fits into the "big picture".

If a designer is very strict about clearly defining where objects fit within the layers, and the interfaces between the layers, then the potential for reuse of many objects in the system can be greatly increased. A common problem with many object designs is that they are too tightly constrained to the limits of the particular application being built. Many novice designers tend to put too much of the logic of an application in the Application Model layer. In this case, there are few, if any, domain objects that are potentially available for reuse in other applications.

Another benefit of this layering is that it makes it easy to divide work along layer boundaries. Woolf demonstrates how a "layered and sectioned architecture" can be made the basis of a source-code control system. It is easy to assign different teams or individuals to the work of coding the layers in four-layer architectures, since the interfaces are identified and understood well in advance of coding.

Finally, a four-layer architecture makes it possible to code the bulk of your system (in the domain model and application model layers) to be independent of the choice of persistence mechanism and windowing system.

Sources:

Layering is not a new idea in computer science -- Tannenbaum mentions it in conjunction with the OSI seven-layers communications model. Shaw discusses layering as an architectural choice.

Hendley discusses the benefits in portability gained by additional layering in the View and Application Model layers in Smalltalk. Brown further investigates the reasons for applying four-layer architectures for Smalltalk.

Related Patterns:

Trim and Fit Client shows how a four-layer architecture can be used in conjunction with a 3-tier machine architecture in a distributed object environment.

3.40 PATTERN NAME: THREE-TIER ARCHITECTURE

Problem:

How do you distribute responsibility among the different machines in an enterprise to best take advantage of each platform's unique capabilities?

Forces:

Many organizations plan large-scale client-server projects by planning for a large capital purchase of desktop machines and network servers, to be purchased along with the development of new software. However, the technology used to develop the software will often change faster than the plan anticipates. Several releases of the target operating system may occur between the time a large project is started and its final delivery.

Because of the above, the client machines purchased are often not up to the final size of the software that is produced. It is not uncommon to see client-server applications in production today where the total amount of client code resident in memory at any time is 12 megabytes or higher.

It must also be kept in mind that the number of clients in a system will be from one to three orders of magnitude greater than the number of servers in a system. This multiplier will heavily weight the cost of a system towards that of the client. If the choice is to buy additional memory and a faster processor for 100 clients, or for one server, the choice is fairly obvious. Therefore:

Solution:

Utilize a machine architecture that splits responsibility into three "tiers" of computation. These tiers are:

- **The Client.** The client should be primarily responsible for the display and interpretation of information. It is the focal point for user interaction with the system as a whole. As such, the client can be optimized for display and fast network access, but may not need to have the memory and computational power available in other tiers.

- **The Departmental server.** The departmental server is usually a dedicated high-end PC-style machine or a specialized UNIX workstation. The server is capable of handling many more computations per second than the clients, and often has a much, much greater amount of physical memory and disk space. This makes it valuable as a localized cache of information shared among many clients. This relieves the burden of storage and computation on the client, and can reduce the network traffic to the Enterprise server.

- **The Enterprise server.** This is traditionally a mainframe. While mainframes have gone out of fashion in the past few years, the fact still remains that for high-volume, high-speed transaction processing, there is no better technology. Organizations have invested a great deal of time and money in these machines and their software -- it is in their interest to preserve as much of that investment as is possible, while still keeping all options open for the future.

Discussion:

A three-tier approach (see Figure 2: Three-tier architecture) gives the best solution for new development, while still supporting existing systems. If it is implemented correctly, the clients are completely de-coupled from the mainframe. Intermediate server code can be developed in such a way as to minimize dependence on the mainframe so that it can be phased out over time if that is desired.

Related Patterns:

Phase-In Tiers shows how to move from a two-tier client-server approach to a three-tier one.

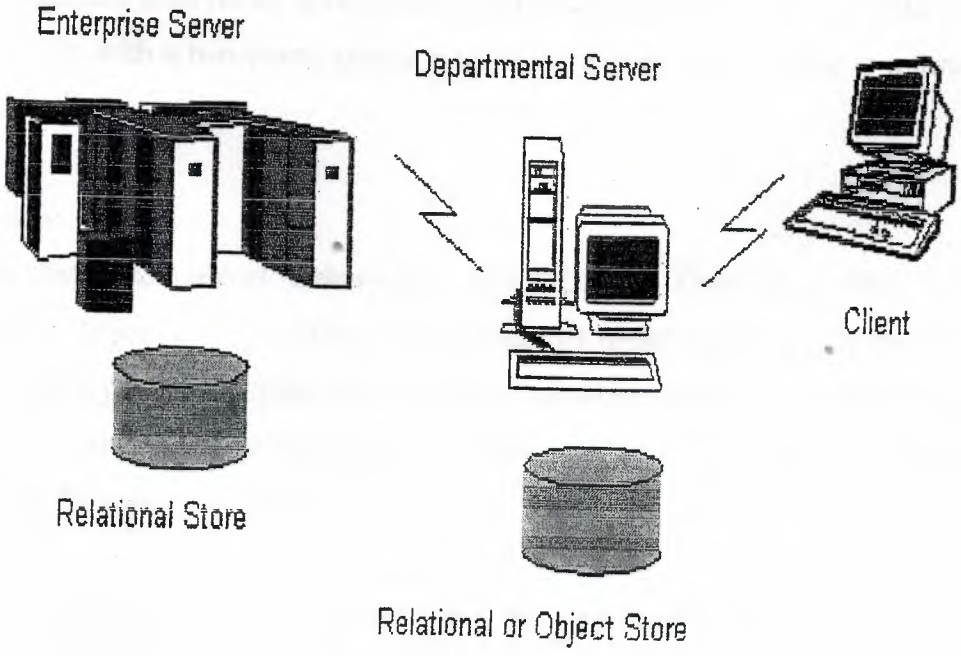


Figure 3.40: Three-tier architecture

3.41 PATTERN NAME: PHASE-IN TIERS

Problem:

You must come up with a solution that supports both your current and planned network architecture, and yet leverages your investment in object technology to produce results quickly.

Forces:

You need to best utilize existing and new computing and network resources. You would like to move to a client-server set of solutions as quickly as possible, but there is no way that everything can be replaced at once. The cost of a total redevelopment effort is prohibitive, and your staff could not complete the effort in a reasonable length of time. Therefore:

Solution:

A good approach is to begin all development on the client (sometimes resulting in a prototype "fat client") and then push the code from the bottom two layers of a four layer architecture onto a server as development progresses. In this way you can add tiers over time, starting with a two-tiered system (i.e., a "fat client") and moving to a three-tiered system later.

Discussion:

Modern distributed object technologies like CORBA, GemStone, IBM VisualAge Distributed Option, and PP-D Distributed Smalltalk make it possible (in fact, relatively easy) to move processing from client machines on to servers. Using these technologies, early releases can be made with fewer tiers than later releases without necessitating big changes in the code of a system.

Using distributed object technology does not preclude using either relational or OODBMS technology as appropriate for object persistence on the server(s) in any of the upper tiers. In fact, one common solution is to utilize an ODBMS for object persistence between tiers one and two, and an RDBMS for persistence between tiers two and three.

Related Patterns:

Trim and Fit Client discusses how to distribute behavior between client and server. Four Layer Architecture discusses how to distribute behavior among the layers of software architecture.

3.42 PATTERN NAME: TRIM AND FIT CLIENT (OR DISTRIBUTE LAYERS TWO BY TWO)

Problem:

Having seen that both "fat" and "thin" clients are not appropriate, what is the proper, "healthy" division of behavior between client and server? How do you design a system such that the system is responsive on a client's machine, and yet maintainable and architecturally sound?

Forces:

Correctly distributing behavior in a client-server system is a difficult task. When PC's were first introduced to major companies, they were most often used as terminal emulator front-ends to existing mainframe programs. These so-called "thin" clients did not take advantage of the capabilities or processing power of the new PC clients, and could not allow for complex user interaction to occur on the client side.

In a response to this, the first generation of client-server systems often overloaded the client by placing all of the domain and display logic on the client. These so-called "fat" clients were often characterized by being big, slow, and inefficient at utilizing machine resources (network traffic, CPU horsepower, etc.). In a response to this, the "thin" client model is back in vogue. This time the client consists of a Web browser that accesses dynamically generated HTML pages from a central host. This "Web 3270" approach has the same problems as the original terminal-emulator approach in that the entire processing takes place on the server end, and complex user interaction is not allowed.

The price of memory, hard-drive space and processor speed come down almost daily. However, it is apparent that the requirements placed on these resources by modern software are expanding at an even faster rate. It is not uncommon to see large-scale client-server programs that take up 15 or more megabytes of memory by themselves. At

the same time, these same advances that make client machines more powerful are also making multiprocessor servers more cost-effective. However, most current client-server systems do not take full advantage of this processing power, as the server is most often used only as a data or file server. Therefore:

Solution:

Break the system for distribution between the Application Model and Domain Model layers, or at some appropriate point inside the Domain Model layer (see Figure 3: Distributed layers). The upper two layers will reside on the client. The lower two layers will reside on the server. This will allow the code that receives the most user interaction (the upper two layers) to handle these close to the user. On the other hand, the code that handles the business logic will reside on the server. This makes it easier to design the interaction between the objects in your system if you know ahead of time where in the network these objects will reside.

Discussion:

This solution minimizes the amount of processing that must be done on a Client, and can reduce its need for memory and computational power. Note that once a system is broken up this way it does not require that the top two layers be implemented in the same language as the bottom two layers. A heterogeneous system will work if some type of object-to-object communication is provided. It would be perfectly acceptable to write the top two layers in Java, and the bottom two in C++ or Smalltalk, or have the entire system written from top to bottom in one language. So long as an object communication technology like CORBA or SOM can be used to provide intra-machine message passing, there should not be any restrictions put on the choice of language or platform.

Sources:

Texas Instrument's Control WORKS project was the first large-scale project I have seen that successfully implemented this method. Since then many companies have broken their applications up in this way -- it is in fact recommended by Gemstone as the best use of their product.

Related Patterns:

Four-layer architecture demonstrates why systems should have clear layer boundaries and how that helps make systems more manageable.

CHAPTER FOUR

DATA BASE OPTION OF THE LIFE HOSPITAL

4.1 Databases - Microsoft Access

A database, generally speaking, is a collection of information specially organized into a group. The information should be organized in a way that allows for easy retrieval. As an example, a telephone book is a non-computerized database of information. It is organized in alphabetical order and includes information such as names, addresses, and telephone numbers.

Computerized databases allow you to manipulate large amounts of data quickly and easily. They simplify have tasks such as searching for specific data, organizing and sorting data, and making corrections or changes to data.

In Microsoft Access, the data is stored in tables. Every table has a structure that provides for the collection, organization, storage, and retrieval of data within the tables.

A table consists of fields and records. Fields are categories of information. For example, in an address table, you may maintain names, addresses, cities, states, and zip codes. A set of fields which contains data for a single entry is called a record.

MS Access is a relational database application (DBMS – Data Base Management System).A relational database contains a large amount of data that is split into numerous interrelated tables. Each table is then smaller, more manageable, and, in turn, more efficient. In a relational database, each table should represent one subject, for example, Name, Address, or Zip Code. These tables can then be joined together to make them related. When tables are related, you can access information from any of the fields in the related tables. This way your reports, forms, and queries, can be based on information from any of the related data tables.

4.2 Basic Information about Tables, Forms, Reports and Queries

4.2.1 Tables

A table is a collection of data about a bout a specific topic, such as products or suppliers. Using a separate table for each topic means that you store that data only once, which makes your database more efficient, and reduces data-entry errors.

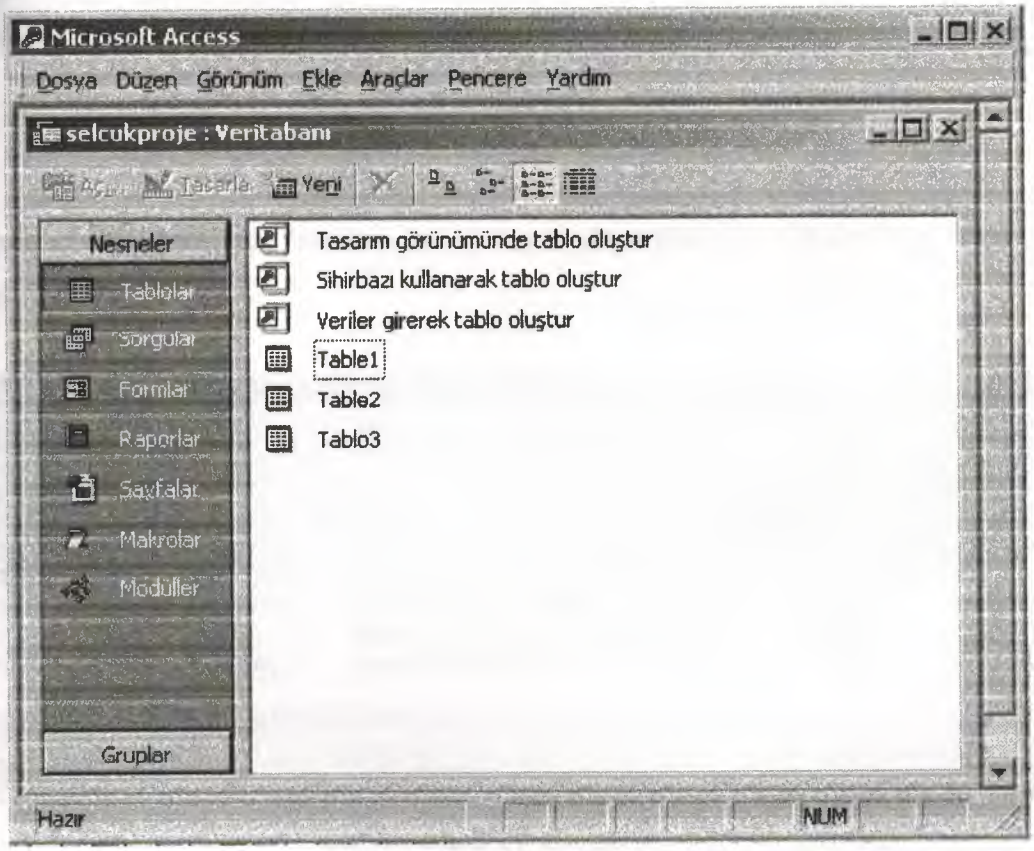
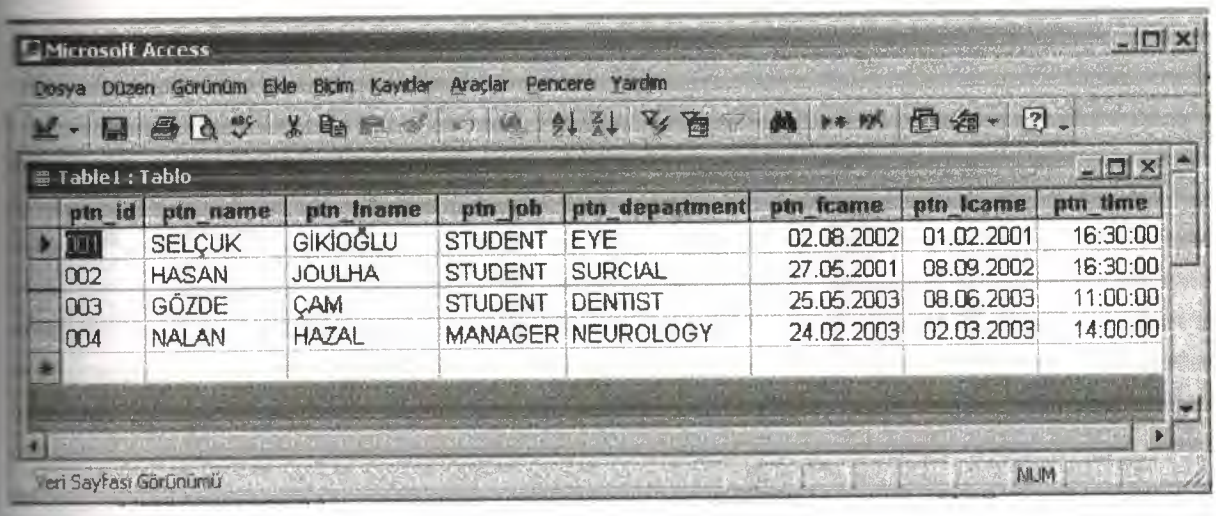


Figure:3.1 Microsoft Access



ptn_id	ptn_name	ptn_lname	ptn_job	ptn_department	ptn_fcame	ptn_lcame	ptn_time
001	SELÇUK	GİKİOĞLU	STUDENT	EYE	02.08.2002	01.02.2001	16:30:00
002	HASAN	JOULHA	STUDENT	SURCIAL	27.05.2001	08.09.2002	16:30:00
003	GÖZDE	ÇAM	STUDENT	DENTIST	25.05.2003	08.06.2003	11:00:00
004	NALAN	HAZAL	MANAGER	NEUROLOGY	24.02.2003	02.03.2003	14:00:00

Figure:3.2 Table1[Ms Access]

I use ptn_id, ptn_name, ptn_lastname, ptn_job, ptn_department, ptn_fcame, ptn_lcame, ptn_time in table1.

ptn_id	ptn_name	ptn_lastname	ptn_tel	ptn_@posta	ptn_insureno	ptn_addres	ptn_departmer
001	SELÇUK	GİKİOĞLU	05322706258	selcukg05@hot	100001	near east unive	EYE
002	HASAN	JOULHA	05338645685	hasan@yahoo.	100002	gönyeli	SURCIAL
003	GOZDE	ÇAM	05365648616	gozde@mynet.	100003	kaymaklı	DENTIST
004	NALAN	HAZAL	05425715989	nalan@yahoo.c	100004	lefkoşa	NEUROLOGY

Figure:3.3 Table2 [MS Access]

I use ptn_id, ptn_name, ptn_lastname, ptn_tel, ptn_@posta, ptn_insureno, ptn_addres, ptn_department in table2.

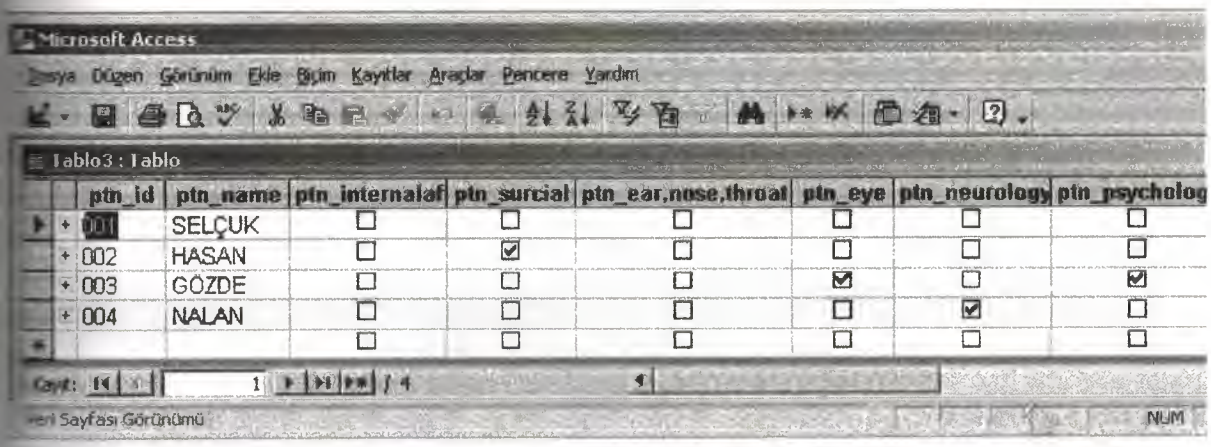


Figure: 3.4 Table3 [MS Access]

I use ptn_id, ptn_name, ptn_internalaffairs, ptn_surcial, ptn_ear,nose, throat, ptn_eye, ptn_neurology, ptn_psychology, ptn_physiotherapy, ptn_dentist in table3.

4.2.2 Forms

You can use forms for a variety of purposes. Most of the information in a form comes from an underlying record source. Other information in the form is stored in the form's design.

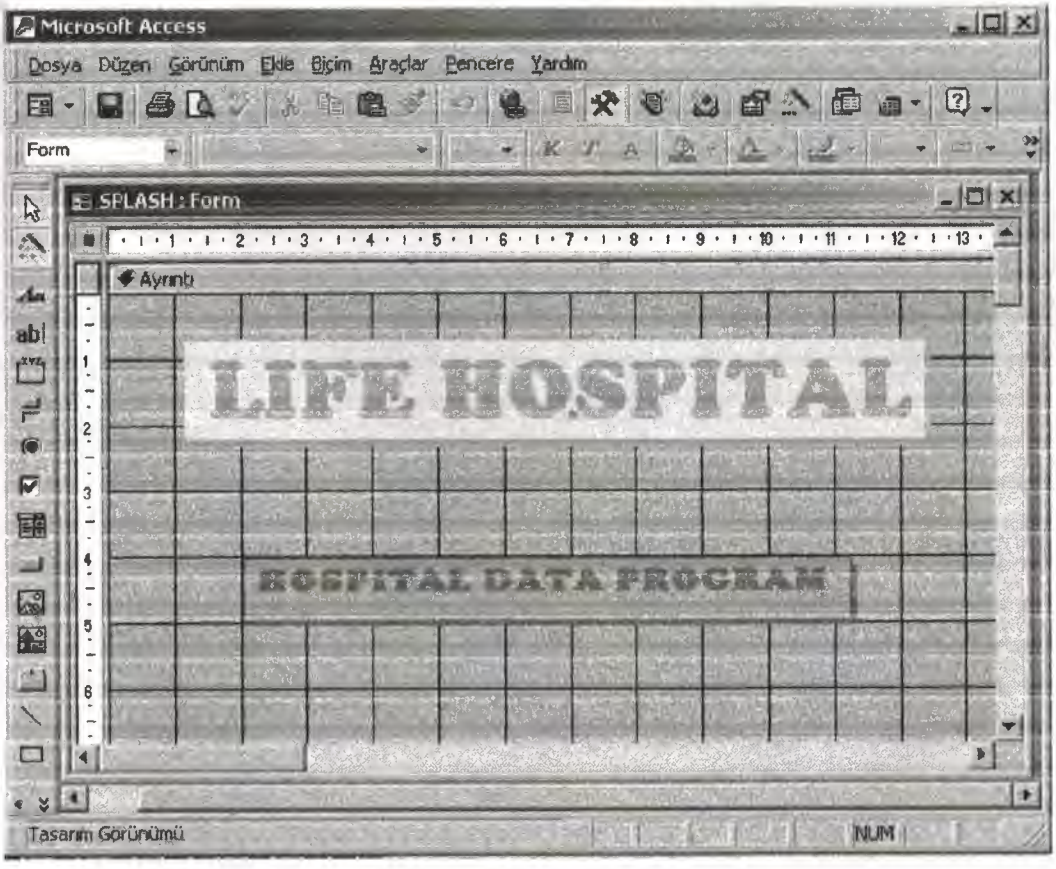


Figure3.5: Form Example[Ms Access SPLASH SCREEN]

4.2.3 Reports

A report is an effective way to present your data in a printed format. Because you have control over the size and appearance of everything on a report, you can display the information the way you want to see it.

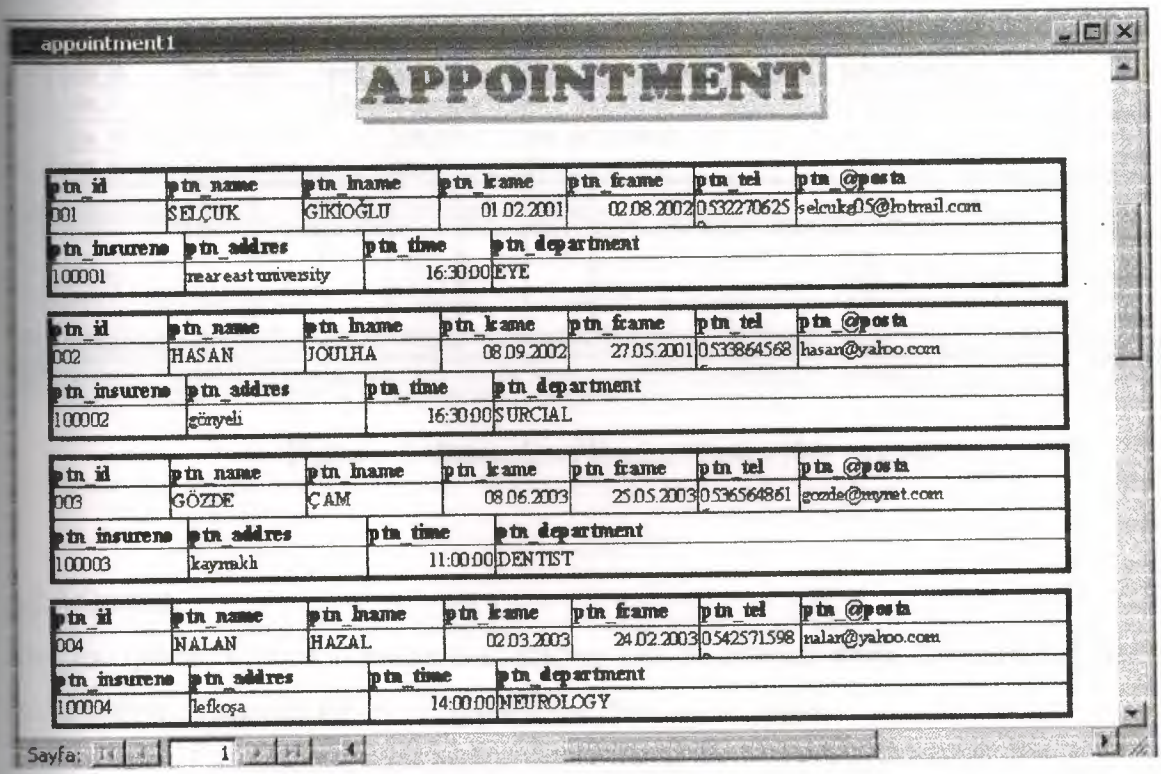


Figure 3.6: Reports Example [MS Access APPOINTMENT Report]

Most of the information in a report comes from an underlying table, query, or SQL statement, which is the source of the report's data. Other information in the report is stored in the report's design.

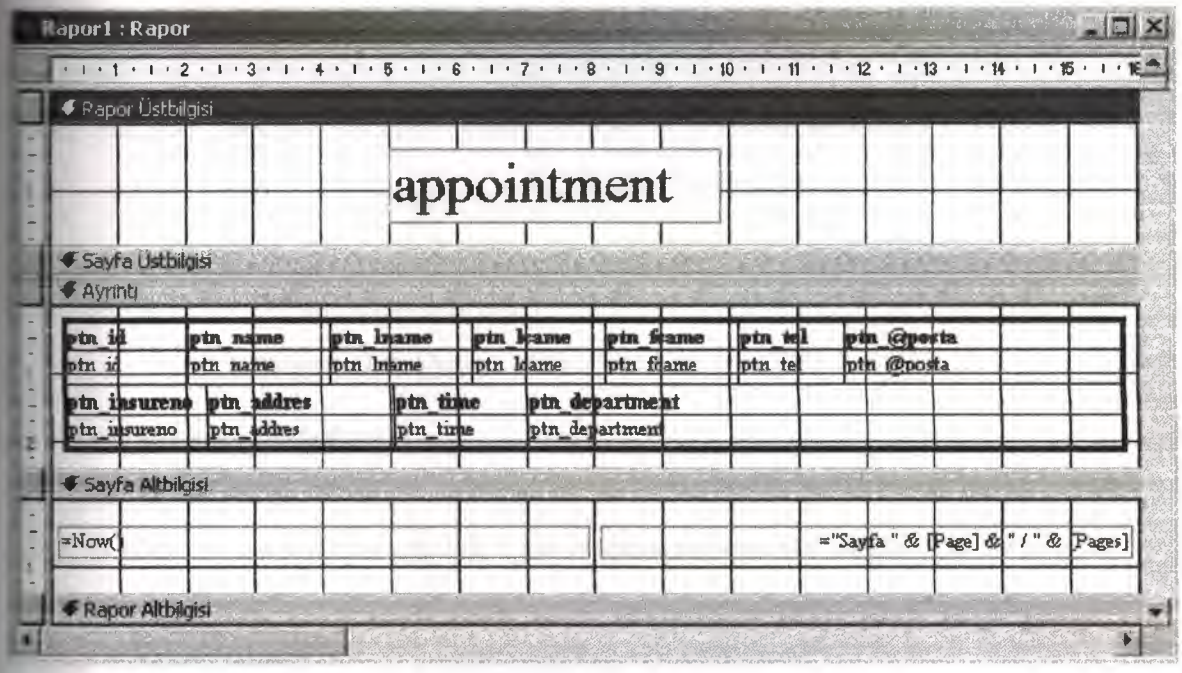


Figure 3.7: Example of Report Design [Ms Access]

4.2.4 Queries

You use queries to view, changes and analyze data in different ways. You can also use them as the source of records for froms, reports and data access pages.

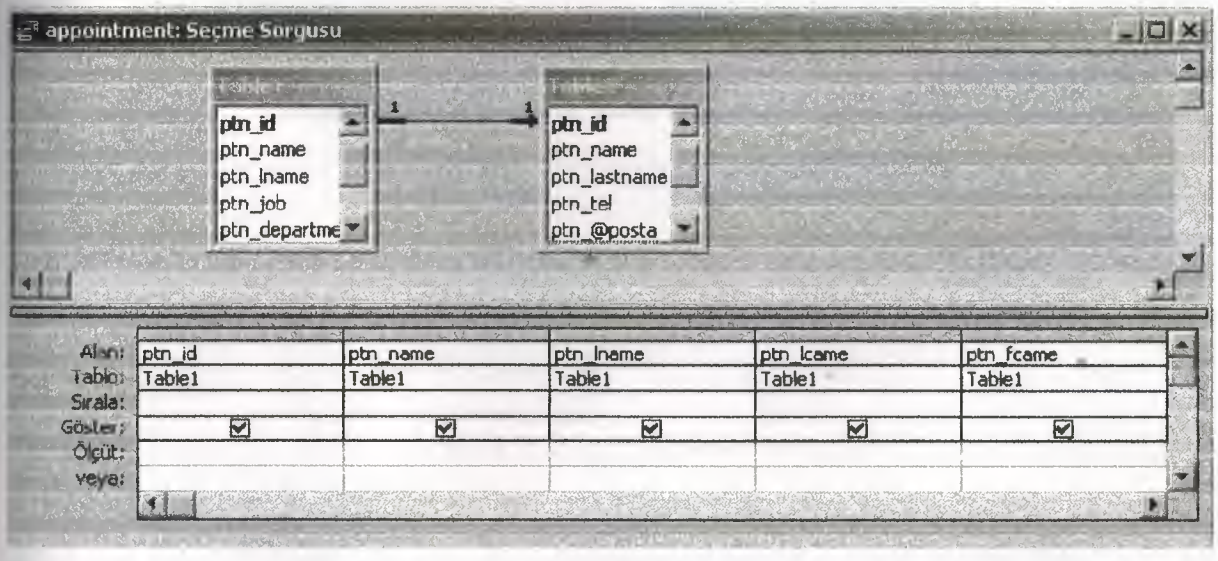


Figure 3.8: Queries Example [Ms Access appointment queries]

The most commonly type of query is a select query. A select query retrieves data from one or more tables by using criteria you specify and then displays it in the order you want.

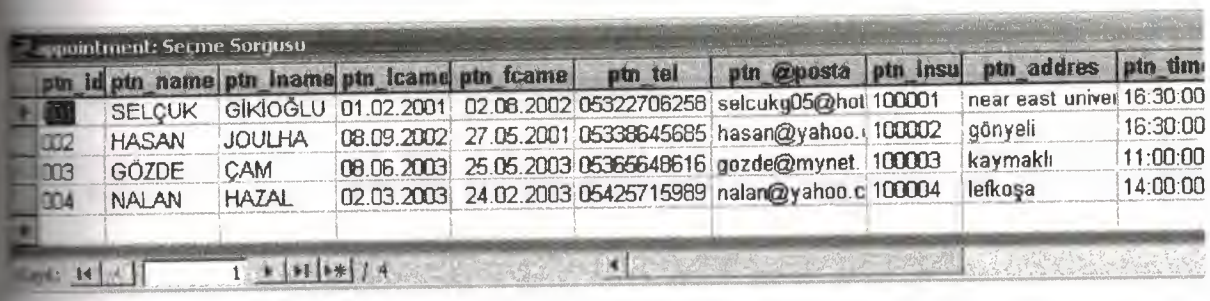


Figure 3.9: Design of Queries Example [Ms Access]

4.3 Description of the software

This software is design for those companies who deals with patient and patient information items. In addition the information about the hospital department can also be maintained in this program.

A Main Menu consist of two (2) option and they are as follows:

- Main Form Page where you can add, delete, appointment, department, search, update, address data
- Main Form Page where you can View different Report

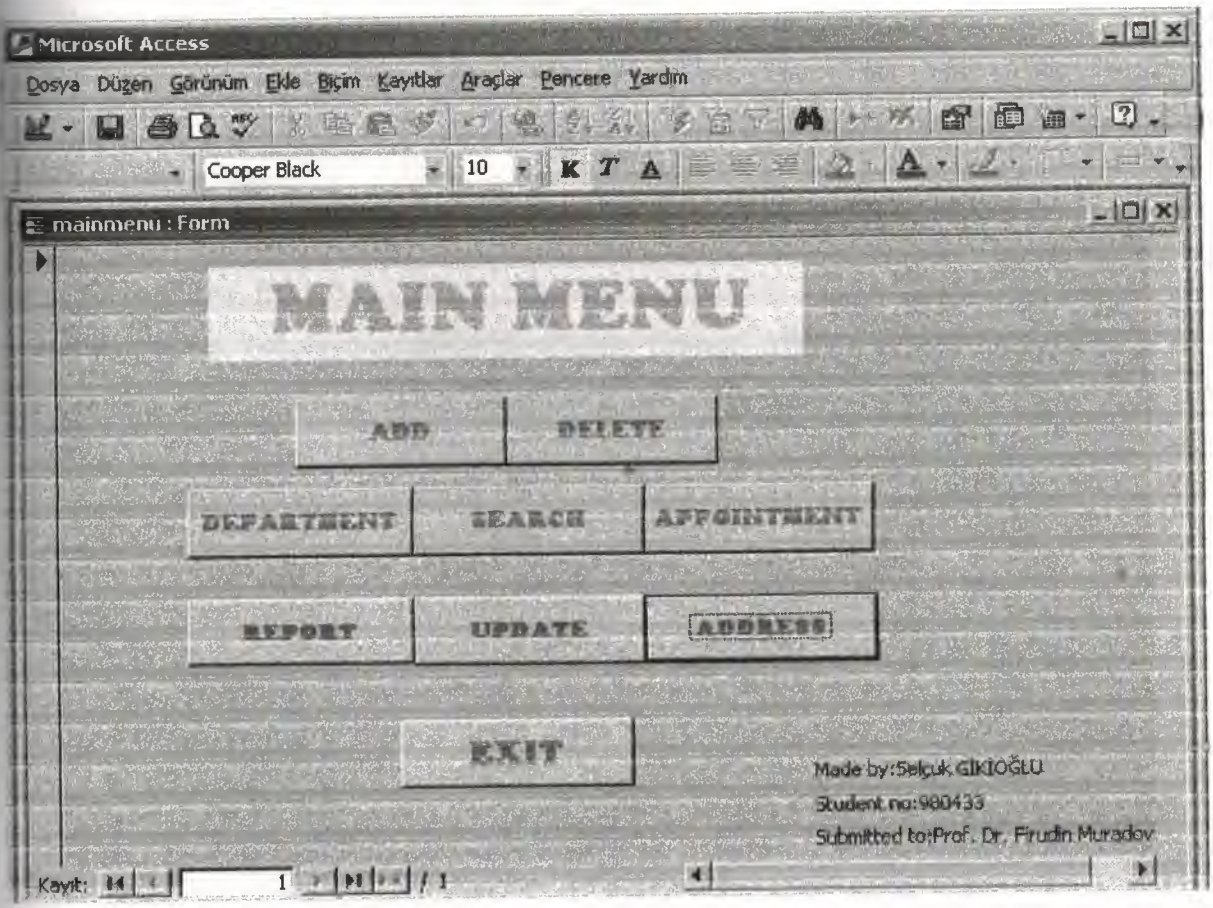


Figure 3.10: MAIN MENU [Ms Access]

4.4 Main Form Page where you can add, delete, search, update, department, appointment, address data

This is a main form page where you can add, delete, update or search the data. The main form pages consist of the following option.

1. Patient ID, Name, Lastname, Job, Department, Lastcame, Nextcome, Time(ADD)
2. Patient ID, Name, Lastname, Job, Department, Lastcame, Nextcome, Time.(DELETE)
3. Patient ID (SEARCH)
4. Patient ID, Name, Lastname, Note, Department, Lastcame, Nextcome (UPDATE)
5. Form of DEPARTMENT Patient ID, Name and Department (anyone)
6. Form of APPOINTMENT Patient ID, Name, Lastname, Insureno, Department, Lastcame, Nextcome, Time, Address, Phone
7. Form of ADDRESS Patient Name, Lastname, Phone, e-Mail, Address

4.4.1 Patient ID and Name Information

In the Hospital information you can just view the Patient ID and the Name of the Hospital, you can ADD, DELETE, UPDATE, SEARCH the information from there, as these information are Locked.

4.4.2 ADD DETAILS

In this window all information about the Patient is listed which gives the information what have been Patient. The information from this window can be add.

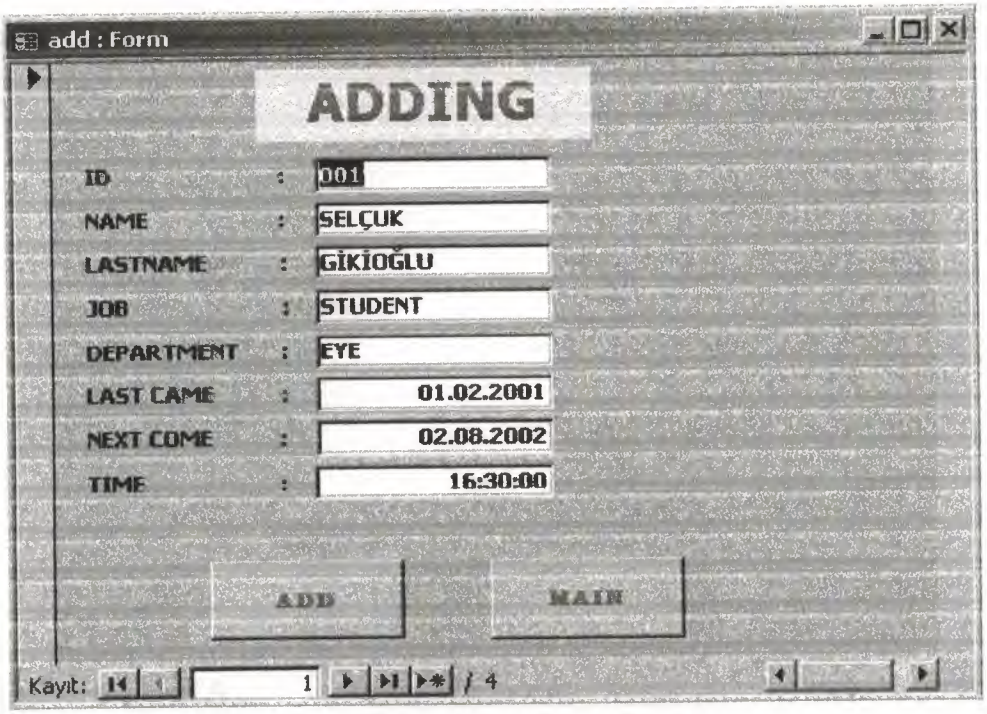


Figure 3.11: ADDING FORM

- ID : The ID of the Patient order (Auto define)
- Name : The Name of the Patient.
- Lastname : The Last Name of the Patient
- Job : The job of the Patient
- Department: The Department of the Patient
- Lastcame : When did he/she come
- Nextcome : When will he/she come.
- Time : What is the time of patient appointment
- Click (ADD): Adding record
- Click (MAIN): Go to main menu

CODES:

```
Private Sub Command7_Click()
Dim db As DAO.Database
Dim rs As DAO.Recordset
Dim s As String
s = "select * from Table1"
Set db = CurrentDb()
Set rs = db.OpenRecordset(s)
```

```
rs.AddNew
rs.Fields(0).Value = ptn_id
rs.Fields(1).Value = ptn_name
rs.Fields(2).Value = ptn_lname
rs.Fields(3).Value = ptn_note
rs.Fields(4).Value = ptn_pursuitsubject
rs.Fields(5).Value = ptn_lcame
rs.Fields(6).Value = ptn_fcame
rs.Update
End Sub
```

4.4.3 DELETE DETAILS

In this window all information about the Patient is listed which gives the information what have been Patient. The information from this window can be deleted.

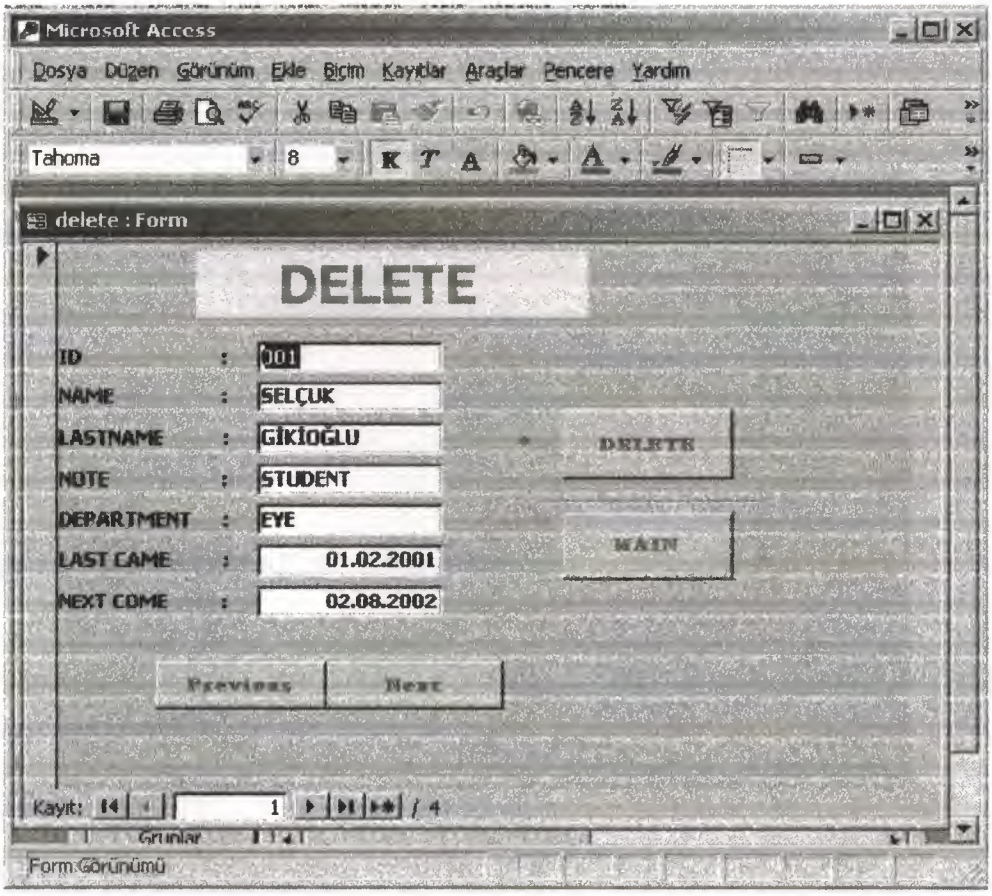


Figure 3.12: DELETE FORM

- ID : The ID of the Patient order (Auto define)
- Name : The Name of the Patient.
- Lastname : The Last Name of the Patient
- Job : The job of the Patient
- Department: The Department of the Patient
- Lastcame : When did he/she come
- Nextcome : When will he/she come.
- Click (PREVIOUS) : Go to the previous record
- Click (NEXT) : Go to the next record
- Click (DELETE) : Delete select record
- Click (MAIN) : Go to main menu

CODES:

```
Private Sub Command7_Click()
Dim db As DAO.Database
Dim rs As DAO.Recordset
Set db = CurrentDb()
Set rs = db.OpenRecordset("table1")
rs.MoveLast
rs.Delete
rs.MoveFirst
End Sub
```

4.4.4 DEPARTMENT DETAILS

In this window all information about the Patient ID and NAME is listed which gives the Department Information.

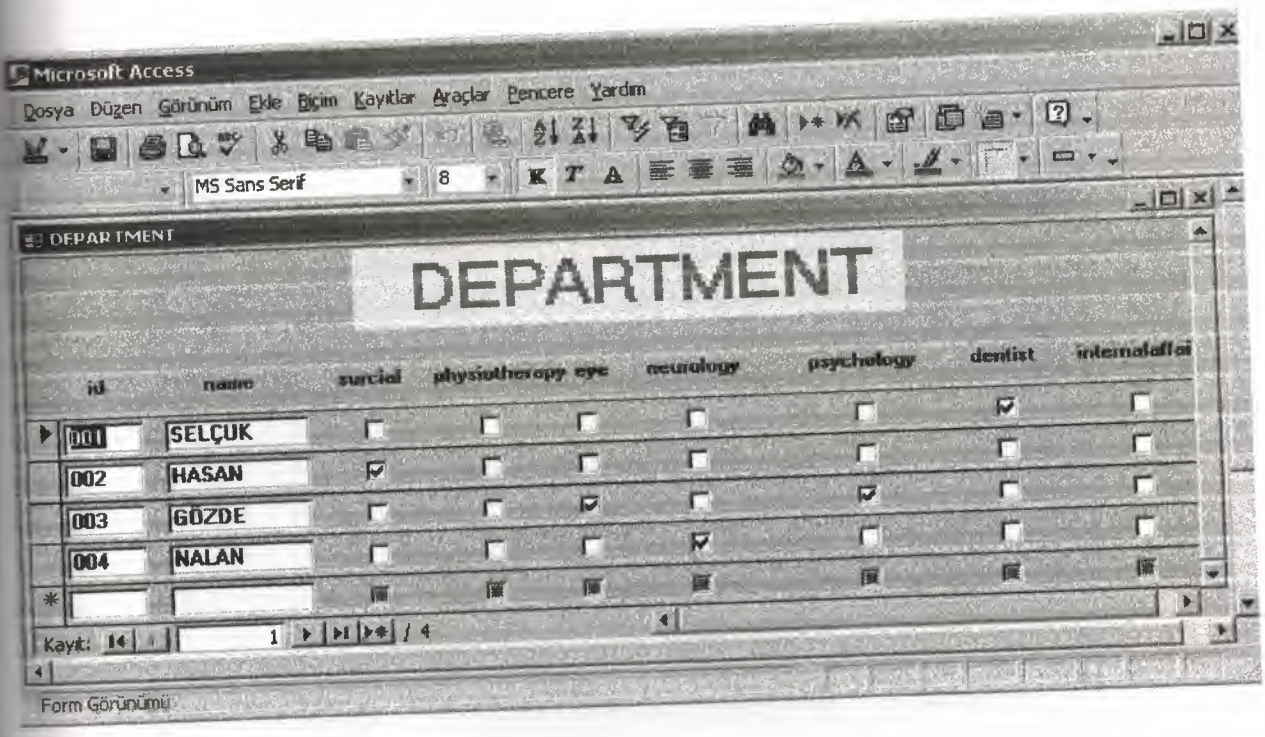


Figure 3.13 : DEPARTMENT FORM

- ID : Patient_id (Auto define)
- Name :Patient Name
- √ : If you want to any department you can tick.

4.4.5. SEARCH DETAILS

In this window you can any patient_Id and than you see patient information.

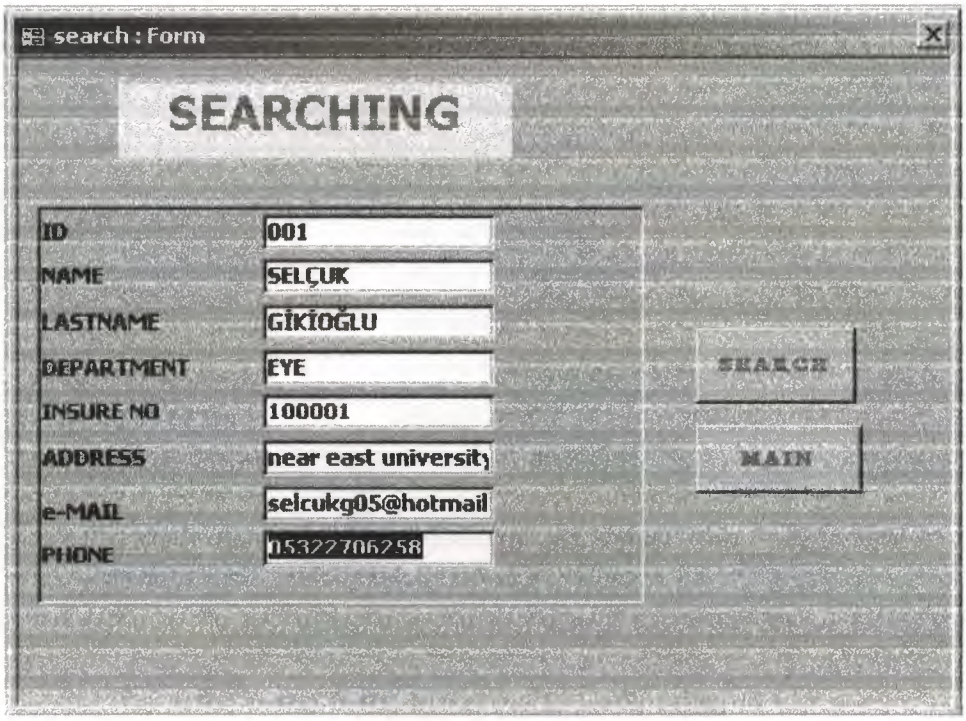


Figure 3.14 :SEARCHING FORM

- ID : Patient ID (Auto Define)
- Name : Patient Name
- Lastname: Patient Last Name
- Department: Patient came which department
- Insure No: Patient Insure No
- Address : Patient Address
- E-Mail : Patient E-Mail
- Phone : Patient Phone
- Click (SEARCH) : Search with ptn_id
- Click (MAIN) : Go to main menu

CODES:

```
Private Sub Komut9_Click()
Me.Child2.Form.Requery
End Sub
```

4.4.6 APOINTMENT DETAILS

This window is a patient appointment and information.

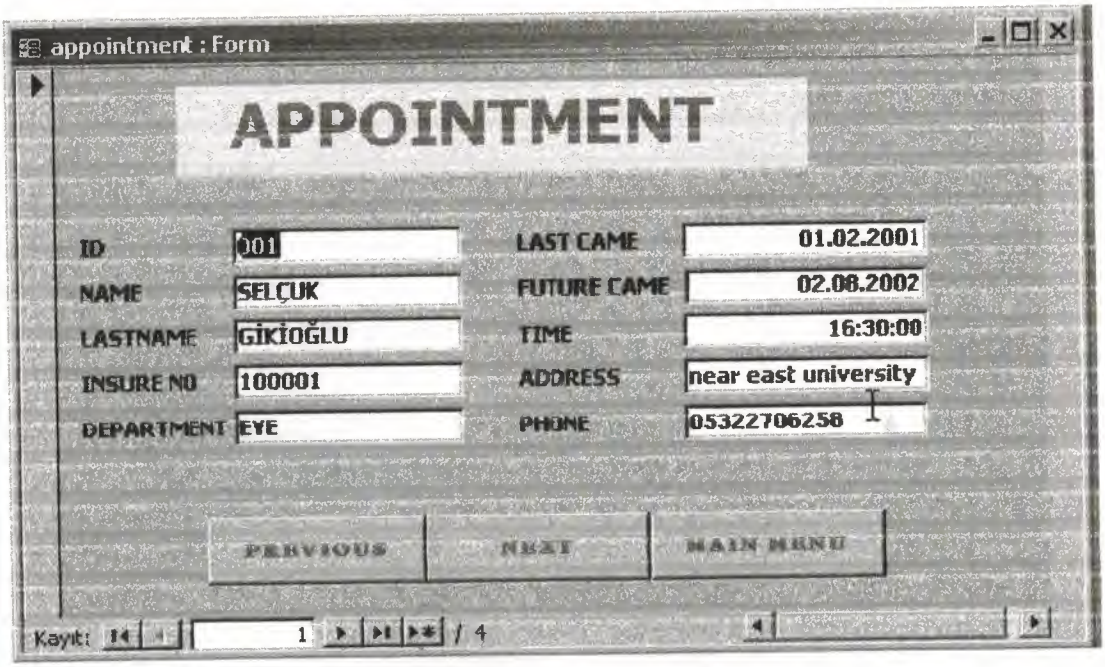


Figure 3.15: APPOINTMENT FORM

- Name : Patient Name
- Lastname: Patient Last Name
- Department: Patient came which department
- Insure No: Patient Insure No
- Department: Patient came which department
- Address : Patient Address
- Phone : Patient Phone
- Lastcame: The patient is when did he/she came
- Futurecome: The patient is when will he/she come
- Time: The appointment time
- Click (PREVIOUS) : Go to previous record
- Click (NEXT) : Go to next record

4.4.7 REPORT DETAILS

A this stage we can view all the information about the data, which has been entered in the table with the help of the form, all the form has there own reports which can be view or can be print if necessary. All the report shows date for the further information. An example how to view the reports are shown in the below figure.

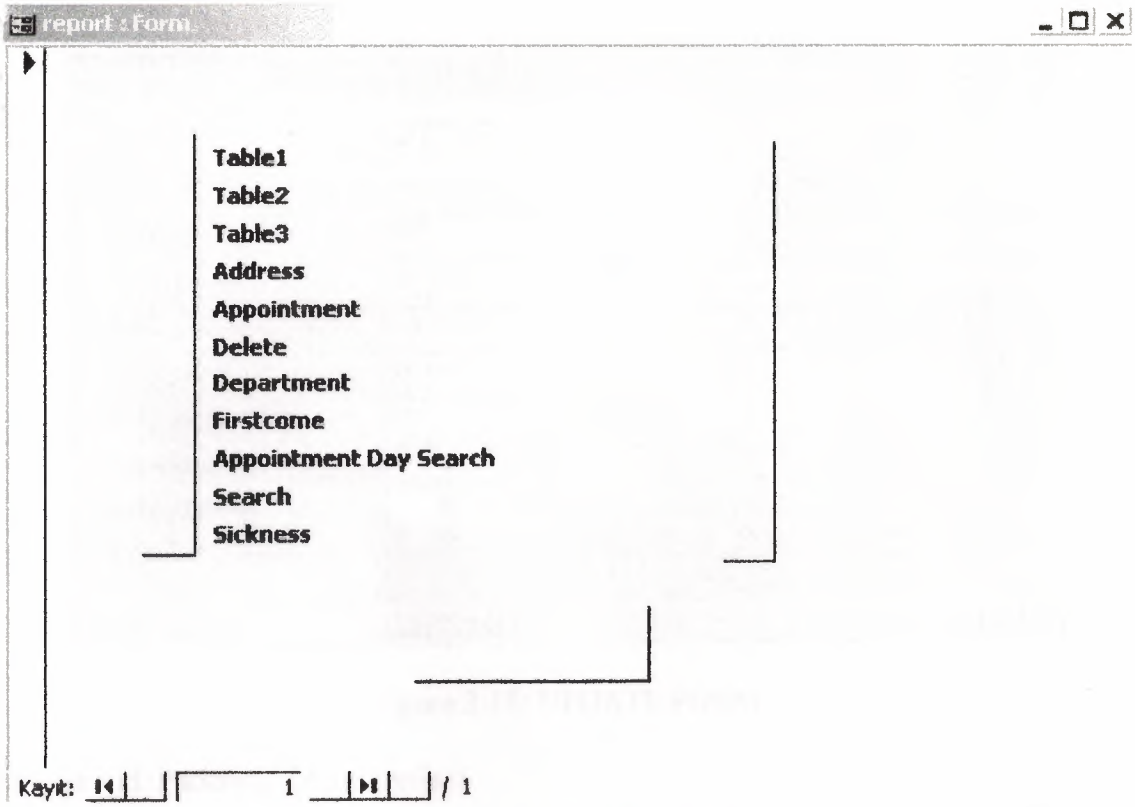


Figure 3.17: REPORTS FORM

- Click1: Table1 (See or Print)
- Click2: Table2 (See or Print)
- Click3: Table3 (See or Print)
- Click4: Address (See or Print)
- Click5: Appointment (See or Print)
- Click6: Delete (See or Print)
- Click7: Fcame (See or Print)
- Click8: Department (See or Print)
- Click9: Appointment Day with search (See or Print)
- Click10: Search (See or Print)
- Click11: Sickness (See or Print)
- Click (MAIN) :Go to main menu

4.4.8 Update Details

In this window all information is update.

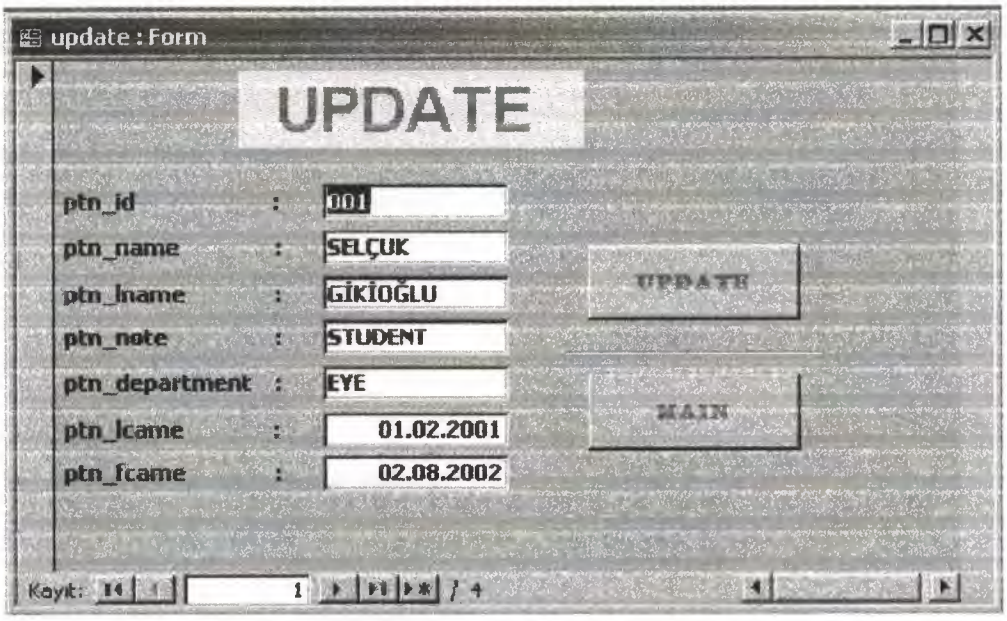


Figure 3.18: UPDATE FORM

- ptn_id :Patient Id (Auto Define)
- ptn_name :Patient Name
- ptn_lname: Patient Last Name
- ptn_note : Patient's notes
- ptn_department : Patient Department
- ptn_lcame: Patient Last Came
- ptn_fcome: Patient Future Come
- Click (UPDATE) :Update select record
- Click (MAIN) : Go to main menu

CODES:

```
Private Sub Command7_Click()
Dim db As DAO.Database
Dim rs As DAO.Recordset
Dim s  As String
Set db = CurrentDb()
s = "Table1"
Set rs = db.OpenRecordset(s)
rs.Edit
rs.Fields(0).Value = ptn_id
rs.Fields(1).Value = ptn_name
```

rs.Fields(2).Value = ptn_lname

rs.Fields(3).Value = ptn_note

rs.Fields(4).Value = ptn_pursuitsubject

rs.Fields(5).Value = ptn_lcame

rs.Fields(6).Value = ptn_fcame

rs.Update

End Sub

Click(UPDATE): These record are update

Click (MAIN) : Close this form and Open Main Menu

4.4.9 ADDRESS DETAILS

In this stage we can see Patient Address records

Figure 3.19: ADDRESS FORM

- Name : Patient Name
- Lastname: Patient Last Name
- Address : Patient Address
- E-Mail : Patient E-Mail
- Phone : Patient Phone

- Click (MAIN MENU): Close ADDRESS and open MAINMENU

4.4.10 REPORTS

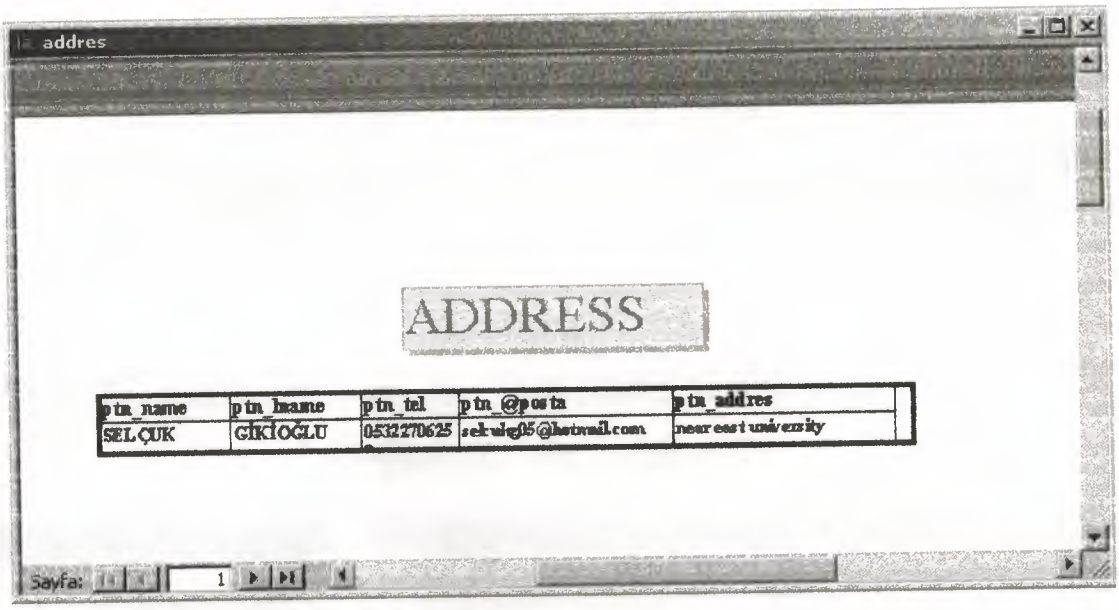


Figure 3.20: ADDRESS REPORT

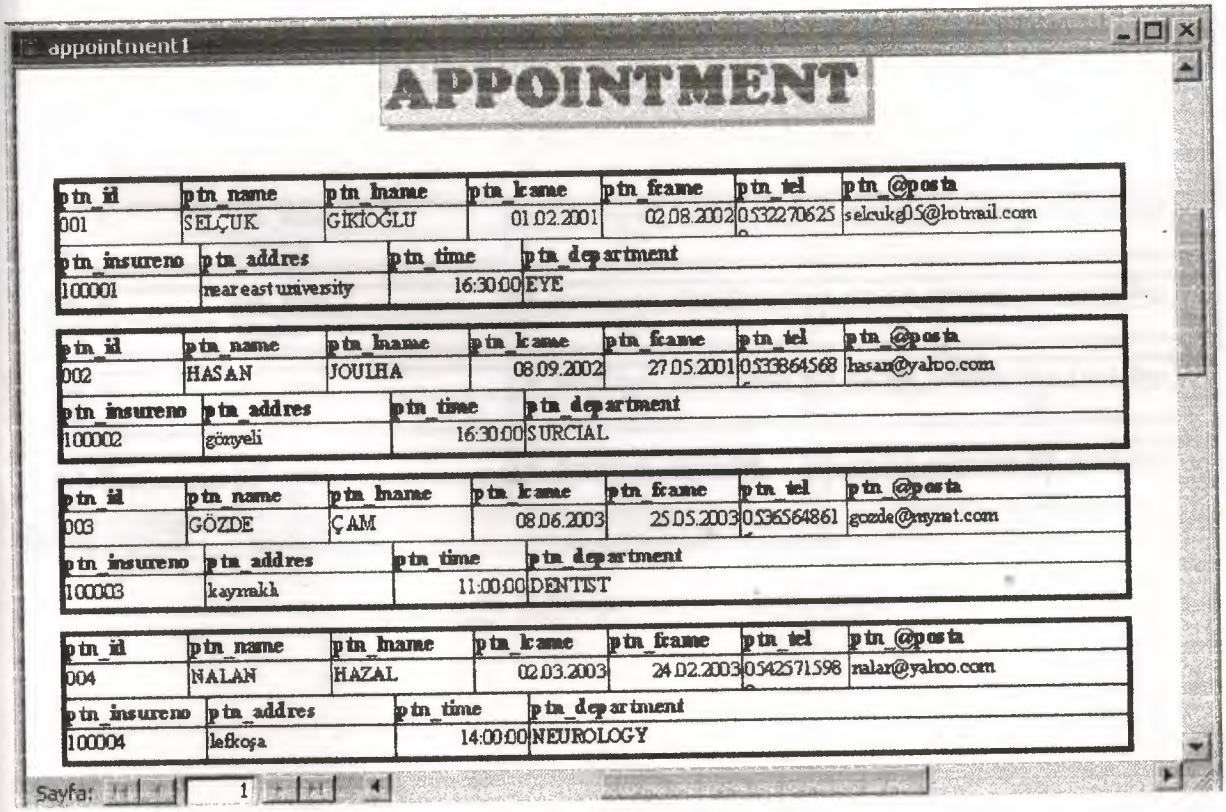


Figure 3.21: APPOINTMENT REPORT

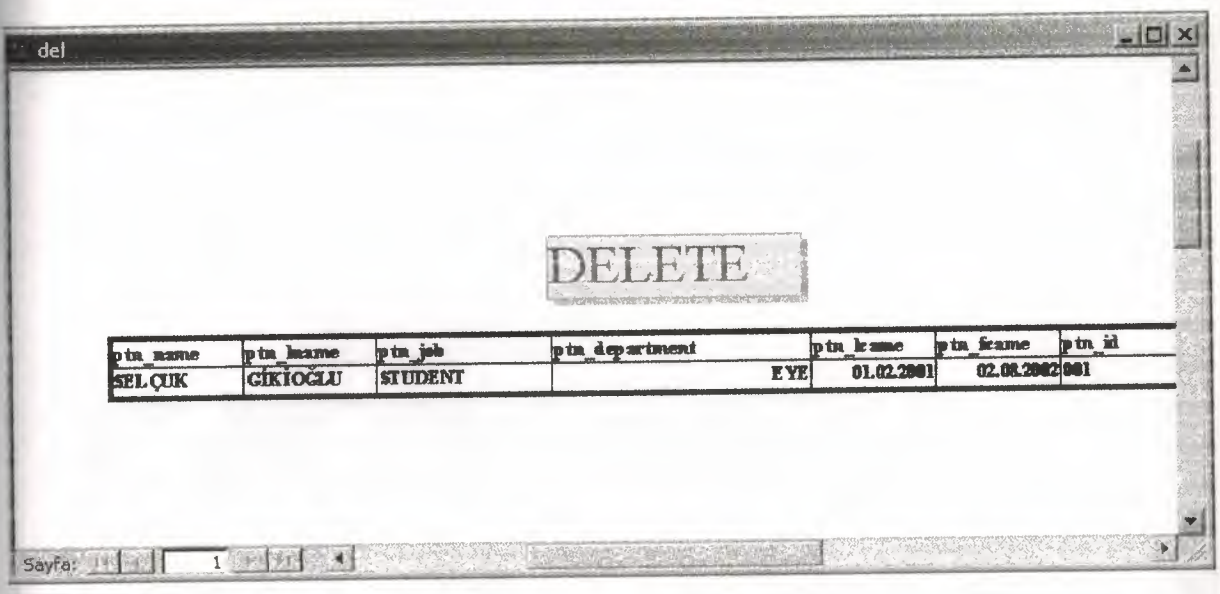


Figure 3.22: DELETE REPORT

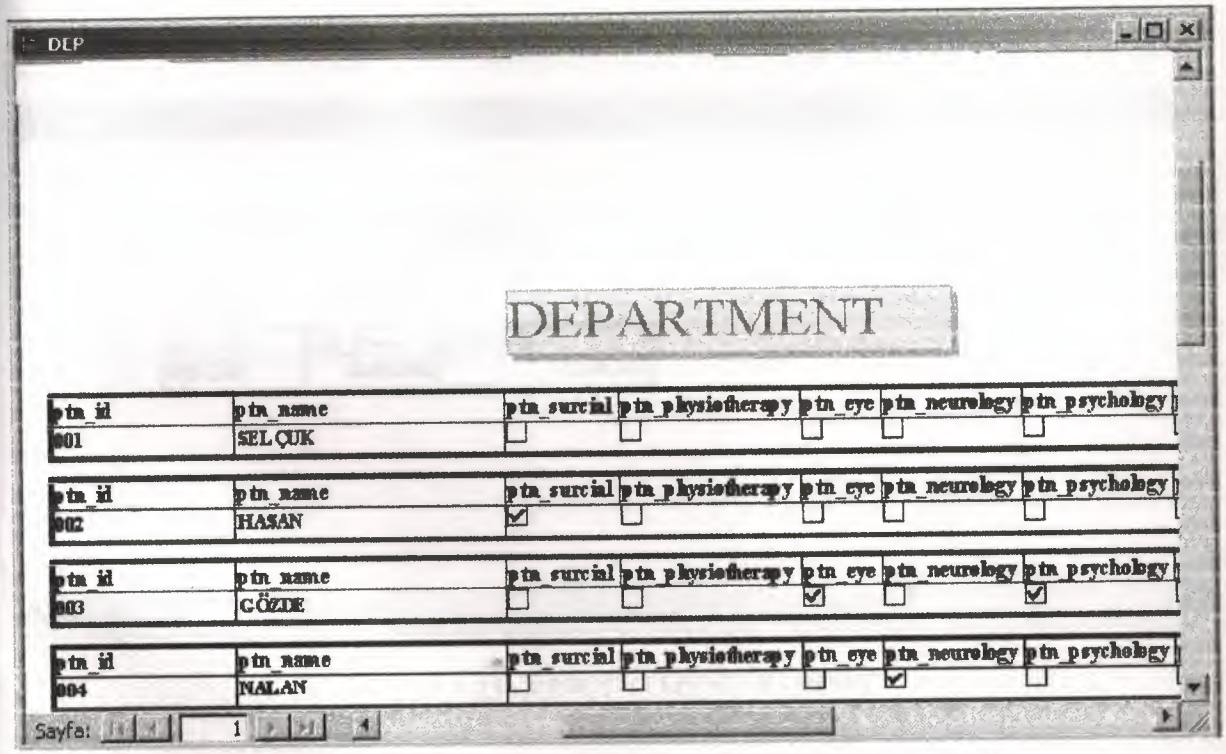


Figure 3.23 :DEPARTMENT REPORTS

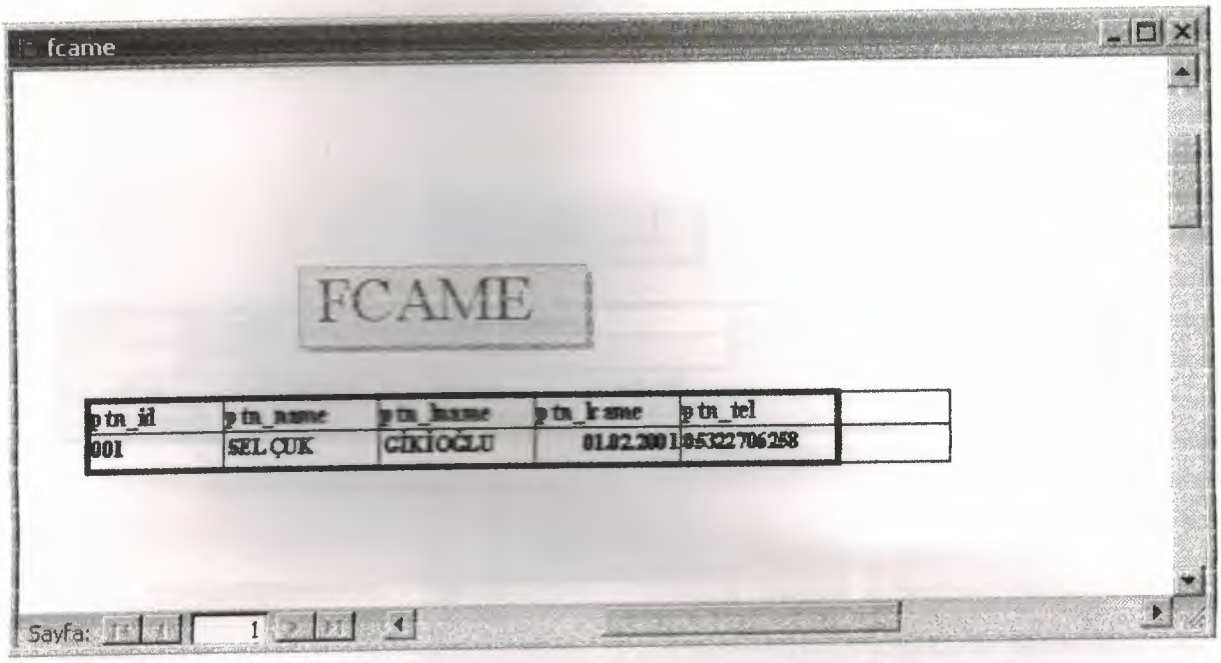


Figure 3.24: FCAME REPORT

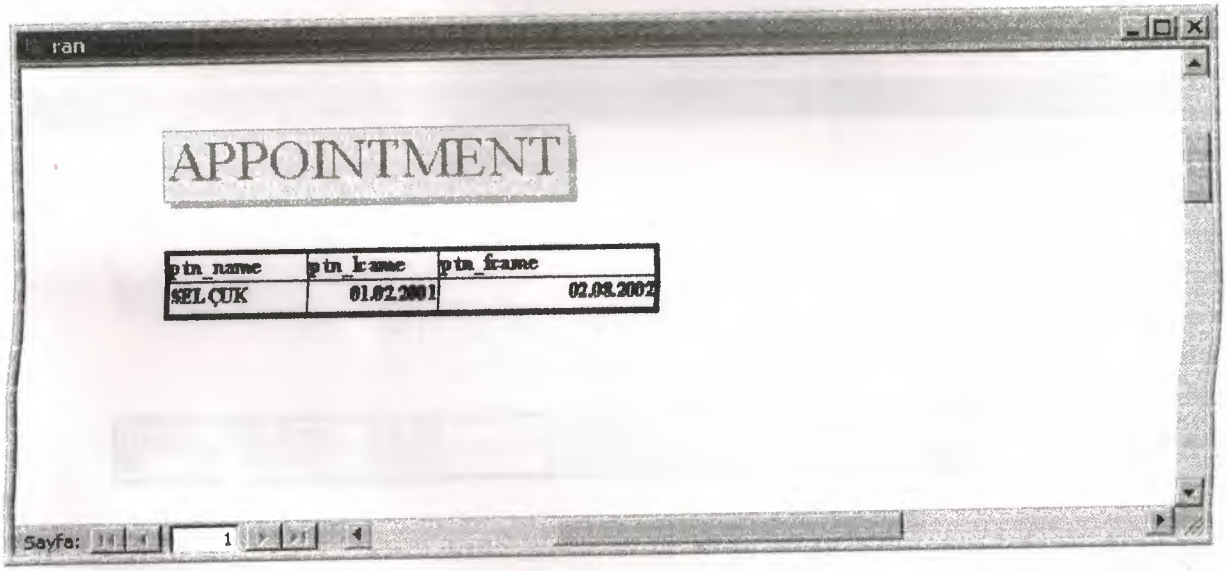


Figure 3.25: APPOINTMENT REPORT

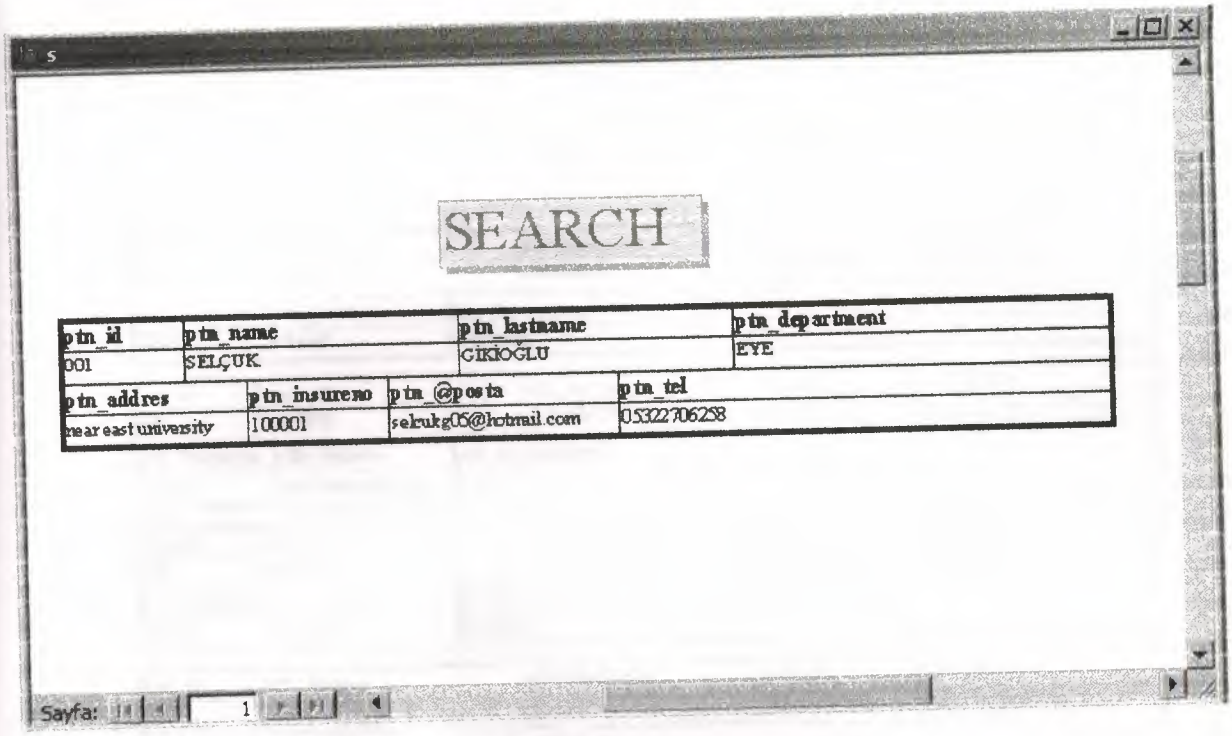


Figure 3.26: SEARCH REPORT

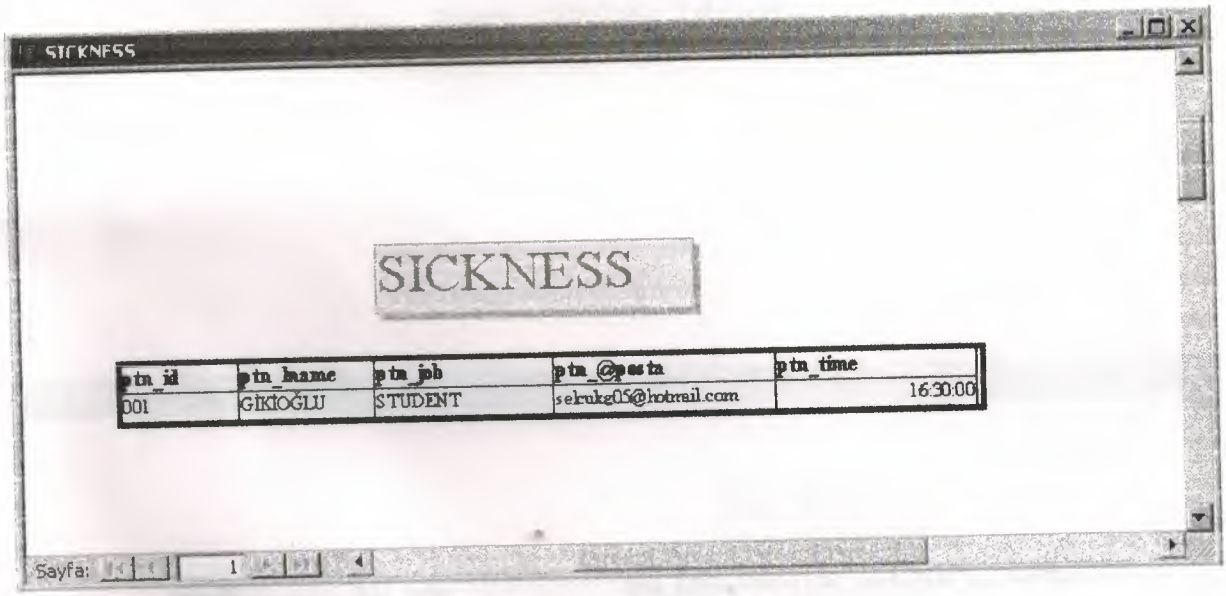


Figure 3.27: SICKNESS REPORT

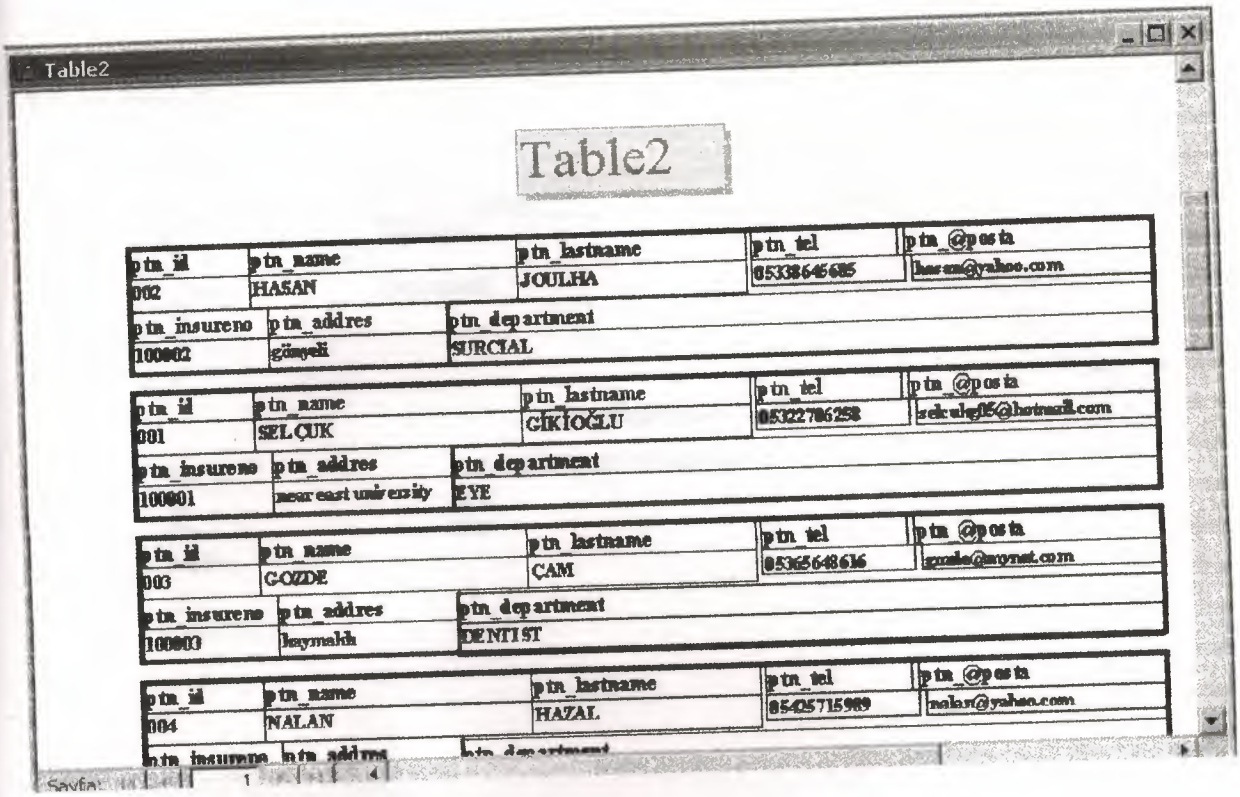


Figure 3.28:Table 2 Report

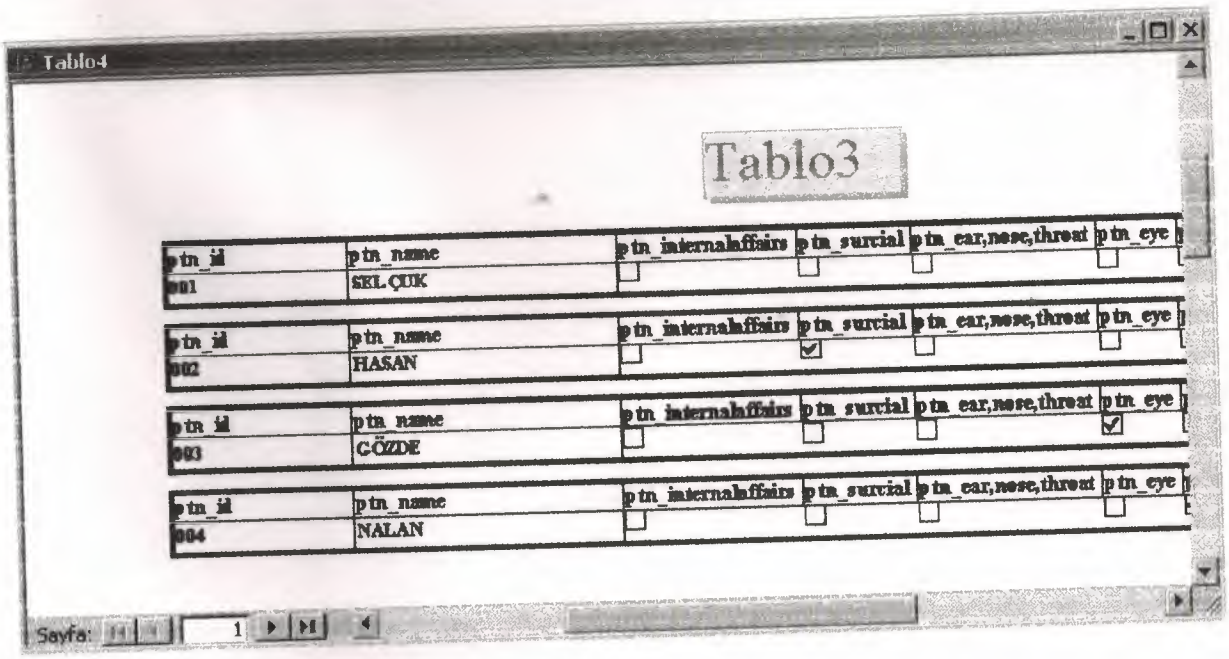


Figure 3.29: Table 3 Report

4.4.11 MACRO

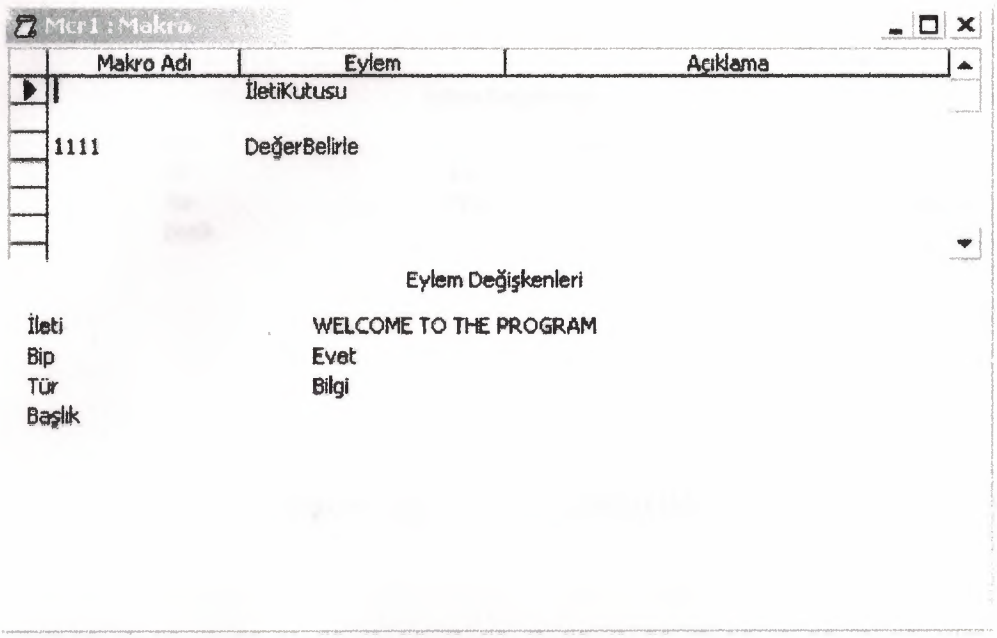


Figure3.30: MACRO DESIGN

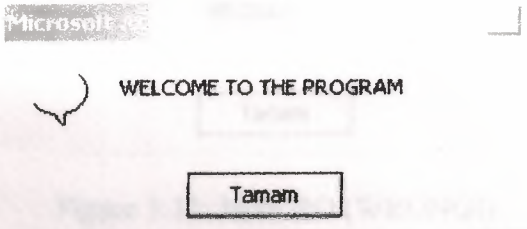


Figure 3. 31: MACRO (WELCOME TO THE PROGRAM)

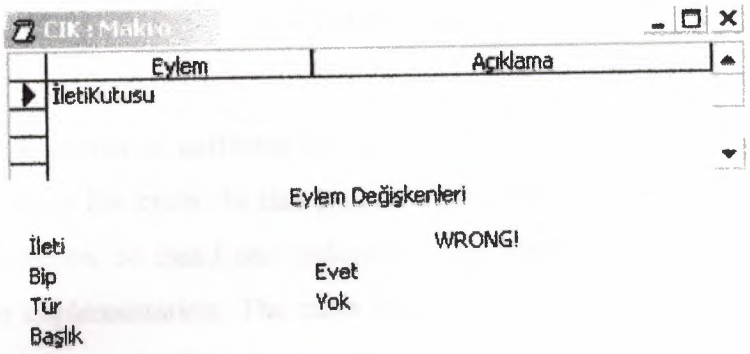


Figure 3.32: MACRO DESIGN

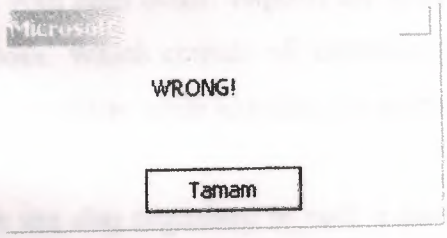


Figure 3.33: MACRO (WRONG!)

CONCLUSION

Practically implementation of software for business though it is related to any field needs a devoted and complete life cycle. In this project I personally visit two companies, which deal with patient information, so that I can understand their requirements and the problems, which may occur in the implementation. The most important think that I would like to mention, is the attitude, which has to be face during the life cycle of the company or organization. And according to my point of view the reason of most unsuccessful project is misunderstanding between the two parties.

The software was created after the deep analyst, so that all-important requirement of the company those who dealing with computer sales and purchase can be accomplished. Patient, name and Id have been added in the program to over come the mistakes, which may occurs. Plus a lock table and form has been generated which contain the entire ID with name, so by mistake it cannot be merged with each other. Reports are also generated with the help of the Queries for the update purpose. Which contain all information with dates. Help file is also written so that there will be no problem while handling the software.

The chapters of the software are also organized in such a manner so that all the information related to DBMS can be understood easily, i.e. chapter one and two are the introductory chapter, which give detail information about DBMS, chapter three contain a advance information and chapter four contain information about the software with the help file.

REFERENCES

[1] Aptech Limited, oracle 8.0

[2] Ms Access, Help

[3] DBMS and Internet Systems (*http://www.dbmsmag.com/)*
Miller Freeman, inc.

[4] *http://www.csusm.edu/hylin/cs111/notes/dbms.htm*

[5] C. Dr Julie A. McCann, 1999, secton2 "Database Application Lifecycle.

[6] C. Dr Julie A. McCann, 1999, secton2 "Relational Data Base Management Issues."