A Project Report On

"CONTACT MANAGEMENT SYSTEM"

Submitted in partial fulfilment for

the award of the Degree in

BACHELOR IN COMPUTER APPLICATIONS

By

"SAIYAM GULATI"
Enrolment No. AJU/190537

Under the guidance of
"Mr. AKASH KUMAR BHAGAT"

ARKA JAIN UNIVERSITY

JAMSHEDPUR, JHARKAHND

DEPARTMENT OF COMPUTER SCIENCE & IT

2019-2022

ARKA JAIN UNIVERSITY,
JAMSHEDPUR , JHARKHAND

ARKA JAIN UNIVERSITY

A

PROJECT REPORT

ON

CONTACT MANAGEMENT SYSTEM

IN PARTIAL FULFILLMENT OF REQUIREMENT OF

DEPARTMENT OF COMPUTER SCIENCE AND

INFORMATION TECHNOLOGY

BATCH 2019-22

GUIDED BY:

Mr. AKASH KUMAR BHAGAT

PREPARED BY:

SAIYAM GULATI

SUBMITTED TO

DEPARTMENT OF COMPUTER SCIENCE & IT

ARKA JAIN UNIVERSITY

ARKA JAIN UNIVERSITY, JHARKHAND

JAMSHEDPUR

DEPARTMENT OF COMPUTER SCIENCE & IT

CERTIFICATE

This is to certify that the project entitled, **" CONTACT MANAGEMENT SYSTEM "**, is bonafied work of " **Saiyam Gulati** " bearing **Roll No:** and **Enrollment No : AJU/190537** submitted in partial fulfillment of the requirements for the award of degree of **BACHELOR OF COMPUTER APPLICATION** from **ARKA JAIN UNIVERSITY,JHARKHAND**.

Internal Guide

Coordinator

External Examiner

Date: 25/5/22

ARKA JAIN University JHARKHAND UNIVERSITY Seal

ABSTRACT

Much of our daily communication activity involves managing interpersonal communications and relationships. Despite its importance, this activity of contact management is poorly understood. We report on field and lab studies that begin to illuminate it. A field study of business professionals confirmed the importance of contact management and revealed a major difficulty: selecting important contacts from the large set of people with whom one communicates. These interviews also showed that communication history is a key resource for this task. Informants identified several history-based criteria that they considered useful.

We conducted a lab study to test how well these criteria predict contact importance. Subjects identified important contacts from their email archives. We then analyzed their email to extract features for all contacts. Reciprocity, recency and longevity of email interaction proved to be strong predictors of contact importance. The experiment also identified another contact management problem: removing 'stale' contacts from long term archives. We discuss the design and theoretical implications of these results.

ACKNOWLEDGEMENT

I am very glad to take this opportunity to acknowledge all those who helped me in designing, developing and successful execution of my Project on “CONTACT MANAGEMENT SYSTEM”.

I would like to extend my thanks and gratitude to my Project guide Mr. Akash Kumar Bhagat for his valuable guidance and timely assistance throughout the development of this project.

I would like to thank my guide, who helped us understand the current system and for giving his valuable assistance as he would be the end user of this ' ARKA JAIN UNIVERSITY.’

I would also like to extend my thanks and gratitude to my respected H.O.D “DR.ARVIND KUMAR PANDEY” without whose help and support this project would not have been possible. Furthermore, I would also like to acknowledge with much appreciation the crucial role of faculty members on this occasion.

Last but not the least, I would like to thank friends who help me to assemble the parts and gave a suggestion about the project.

ARKA JAIN University
Jharkhand

DECLARATION

I, "**SAIYAM GULATI**", hereby declare that the Project entitled **"CONTACT MANAGEMENT SYSTEM "**done at ARKA JAIN UNIVERSITY has not been in any case duplicated to submit to any other university for the award of any degree. To the best of my knowledge other than me, no one has submitted to any other university. This project is done in partial fulfilment of the requirement for the award of degree of BACHELOR OF COMPUTER APPLICATION to be submitted as final semester project as part of our curriculum.

Saiyam

SAIYAM GULATI

AJU/190537

CONTENT

Chapter 1

1. Introduction

1.1 Introduction to system

Theorizing about asynchronous communication ha8s been dominated by comparisons with faceto-face communication . Early asynchronous theories emphasized media differences arguing that asynchronous communication differs from face-to-face communication because of the absence of non-verbal information afforded by gaze and gesture. However, the emphasis on media differences leaves other crucial aspects of asynchronous communication unexamined, particularly those that stem from its persistent nature. We explore those persistent aspects of asynchronous communication in this paper. Research on email, voicemail, and Usenet has revealed various critical features of asynchronous, technologically mediated interpersonal conversations. These conversations consist of multiple messages exchanged over a fairly extended period of time: days, weeks, or even months. This extension of conversations over time implies that people are typically engaged in multiple conversations at any given time. And each conversation often involves multiple people. These properties lead to significant problems of conversation management. People find it difficult to keep track of the content and status of their multiple conversations, as well as the identity, contact information, and expertise of all their conversational partners. Maintaining knowledge of one's contacts is a significant problem in its own right we refer to this problem as contact management. Contact management is clearly complex. A major problem is that people are exposed to an unmanageable number of potential contacts. This is exacerbated by widespread use of distribution lists . It would be both onerous and unnecessary to store detailed information about all these potential contacts. As a result, individuals must decide: (a) which of these potential contacts are important enough to retain information about; and (b) what sorts of information to retain about these chosen contacts.

1.2 Scope of the System

The system will be a Stand-Alone System for Contact Management System Unit of ARKA JAIN UNIVERSITY. This system will be designed to minimize the manual work in maintaining Contect details, Phone number and all other activities under phone contect system. It aims to maximize the productivity and provide improved managed System. This System will be easy to understand and use. More specifically, this system is designed to allow an admin to manage the records of stocks and goods. The software will facilitate creation of different Reports such as Expense and several other Reports.

1.3 Field study of contact management

We present new findings about contact management derived from a field study of workplace communication practices. Other aspects of this study have been reported elsewhere . The study had two related goals: (1) to identify the main problems informants experienced with current communication applications; and (2) to document the key strategies that users had evolved to address these problems. The study consisted of semi-structured interviews and observations of 20 business professionals. They included financial analysts, lawyers, brokers, estate agents, bankers, IT managers, academics, researchers, secretaries, administrators, marketing managers, conference organizers, and public relations specialists. They worked in a variety of settings from multinational corporations to personally owned small businesses. We asked them what communication tools they used, to explain how they used these tools, to describe the main problems with these tools, and to identify strategies they used to cope with the problems. The main tools used were email, voicemail, IM, fax, phone, and written documents. We observed the informants using their tools, also focusing on their use of communication support tools (such as address books, PDAs, and post-it notes) to manage contact information. We first describe the nature and value of contact information and the ad hoc set of tools used to manage it. We then elaborate: (a) the problems people experience in deciding whom to maintain contact information about; and (b) the onerous nature of data entry for the large number of contacts that most people possess. Finally, we document the criteria people use to decide which of this huge set of contacts to keep track of.

Chapter 2

2. Analysis

2.1 The Value of Contact Information

We observed a wide variety of tools being used to store and retrieve contact information. They included: dedicated tools such as personal address books (both digital and physical); corporate directories; organization charts; "tool-specific" address books in email and speed-dial lists for phones; business cards – either in rolodexes or kept loose; 'hotlists' – small sets of frequently called numbers placed in salient locations; pieces of paper on refrigerator doors, post-it notes, notes on calendars. A first question we put to informants was why they thought it was so important to maintain their own personal contact information, when much of the information they stored was publicly available. This is particularly true for employees of large corporations, who have access to corporate directories and organization charts. Three features of current business practice led people to keep personal contact information: (1) Informants often worked with partners or clients from other organizations, and they did not have access to corporate directories for these people; (2) They often needed access to contact information while on the move. It is much easier to take one's contact information along in a PDA or filofax than to access a corporate database from a hotel room or client's office; (3) Corporate databases do not contain the esoteric, personal information needed to maintain a relationship with a contact (birthdays, universities, sports team allegiances, number of children, and so on). Informants were unanimous about the value of their contact information. This was evident not only from their comments, but also from the time they invested in creating and maintaining contact archives. As one informant, Mary, a freelance researcher, pointed out, her personal contact list was a resource that pervaded all of her work.

2.2 Problems:

Contact Selection, Data Entry, Tool Diversity However, contact management has a number of associated difficulties. At first glance, the main problem informants had was the number of contacts they needed to manage. We estimate that this number varied from a low of several hundred to well into the thousands, although reliable estimation was hard given the large number of contact management tools people typically used, and the fact that there was often duplication between these. Upon further examination, though, deeper problems concerned: (a) the need to make an explicit decision that someone was a valuable contact; (b) the diversity of tools used; and (c) data entry.

2.3 Contact Selection:

When someone calls you on the phone, leaves you voicemail, sends you email, or hands you a business card, what do you do? Do you record their contact information or not? The difficulty is that it is hard to anticipate whether, and to what extent, you will need to communicate with that person in the future. Whether someone is an “important contact” only becomes clear over time. The ease of electronic communication, especially the ability to broadcast messages to large numbers of people at little cost, exacerbates this problem: you may be cc’ed on messages, get email from various distribution lists, or receive mass mailings. To be safe, our informants often “over-saved” information, leading to huge rolodexes, overflowing booklets of business cards, and faded post-it notes scattered around their work areas. Despite this strategy, participants were still exposed to many more contacts than they recorded information about. One reason for this was the laborious nature of recording contact information.

2.4 Data Entry:

Informants made it clear that contact information is costly to acquire and especially hard to maintain. They often wanted to record various types of addressing information for a particular contact: work, home, and mobile phone numbers, fax number, email address, postal address, instant messaging alias, as well as the IM system it was good for, and so on. And, as we mentioned earlier, some people found it important to include detailed personal and social information that was useful in maintaining an effective relationship with that contact.

2.5 Diversity of Tools:

All the informants used ad hoc combinations of tools, with some people evolving highly complex and idiosyncratic systems. For example, Mary, the freelance researcher, had over 1000 people in her email address book, a 60 page Word document containing over 1200 people, over 400 people in her PDA, as well as miscellaneous people in Christmas card lists. Ollie, a corporate research scientist, kept 7 different address books, using 2 PDAs, Microsoft Outlook, and 4 independent email address books. He also wrote key work numbers on his office blackboard. One reason why these complex systems evolved was that informants seldom ‘cleaned up’ their contact information. People were loath to delete any contact information. This seemed to be motivated both by the effort of data entry, along with the belief that even little used contact information may be relevant at some future time.

2.6 Criteria for Determining:

Contact Importance Returning to the basic decision people face – is this an important contact? – we sought to find out how our informants dealt with this issue. Informants responded with a surprising consensus. Since they could not make this decision at first exposure, they relied largely on the history of their prior interactions. Further individual factors, such as communication style seemed to affect the number and type of contacts selected. In our interviews, we probed informants to identify specific aspects of interaction history and communication style that were critical in determining important contacts. We asked people to walk us through their contact management tools and explain why particular contacts had been included.

Chapter 3

3. Experimental Study of factors underlying perceptions of contact importance

The experiment examined the criteria underlying user's judgments of contact importance. We presented people with sets of contacts automatically extracted from their email archive. The archives included messages sent by the user to others. They excluded messages that users had received but deleted, as we had no way of accessing these. For each extracted contact we asked users whether they wanted to include that contact in their contact management system in order to keep in touch with them. For each email contact, we also recorded header information from the email archive about each message involving the contact. From this data, we can compute quantitative characteristics of that contact's communication history involving the user, including the frequency, recency, reciprocity and longevity of their exchanges. We can therefore determine the extent to which the decision to select a particular contact correlated with these aspects of communication history. The second part of the study examines individual differences in communication style on contact selection. We considered an alternative experimental design, where instead of having users select contacts extracted from email, we simply looked at the contacts already in their email address books. However our field study research suggested using existing contact entries was problematic: address books often contained out-of-date contacts who had never been removed, or contacts who had been added in anticipation of interactions that never materialized. We wanted instead to collect information about currently important contacts.

3.1 Hypotheses:

The hypotheses are derived from the criteria suggested by our users in the field study.

Communication .The communication hypotheses concern frequency, reciprocity, recency and longevity. First we expected important contacts to interact more frequently with the user. Frequency is defined as the total number of messages exchanged between contact and user, divided by the longevity of their relationship.

3.2 Frequency:

Important contacts should have more frequent exchanges with the user than unimportant ones. We also expected important contacts to show greater reciprocity, so that messages exchanged with important contacts should contain roughly equal numbers of sent and received messages. Reciprocity is defined as (number of messages sent)/(number of messages sent + number of messages received). This definition gives a high reciprocity score to a user sending multiple messages to a contact, but receiving few in return. This situation demonstrates a high investment on the part of the user in maintaining the communication, which we would expect to be reflected in a high perceived value for the contact. Other possible definitions of reciprocity involve the use of message replies (re:). However the header logs that we collected did not contain the message subject lines needed to extract this information.

3.3 Reciprocity:

Important contacts should demonstrate greater communication reciprocity than unimportant ones We also made a more specific prediction about unsolicited communication, which is a specific instance of (lack of) reciprocity. We define unsolicited communication as cases where a contact sends messages to the user, but there is never any communication from the user to the contact. While this definition is simple, it may however, overestimate unsolicited communication by including people who have sent messages that the user intends to respond to. Unsolicited communication: Contacts who send messages to the user, but never receive any communication from the user should be more likely to be classified as unimportant. The next two hypotheses concern the temporal aspects of the communication history, longevity and recency. Longevity is defined as the total number of days between the dates of the first and last messages exchanged by contact and user. Recency is the number of days since the last message exchanged between user and contact. Longevity: Important contacts should interact over longer periods than unimportant ones.

3.4 Recency:

Important contacts should have interacted with the user more recently than unimportant ones. Individual differences in communication style The next hypothesis concerned individual differences between users in terms of their communication style. We classified all users into high and low frequency communicators based on whether they exchanged more messages with contacts than the overall sample mean. We expected more intense communicators to select more contacts because of the greater effort they invested in communication.

3.5 Method:

Users Seventeen users from a large corporate research laboratory took part in the experiment. They included researchers, managers, secretaries, computer support staff and marketing managers. Participants had been using their email system for an average of 3.0 years (standard deviation 1.8 years), and so all had substantial numbers of messages in their archives. Task We presented users with an on-line list of extracted contacts. For each contact we showed contact name (e.g. Abhi , jaiswal), email name (e.g. ajaiswal), domain name (gmaiil.com), the number of messages received by the user from that contact, the number of message sent by the user to that contact, the date of the first message exchanged by user and contact, the date of the last message exchanged. This information was presented in a spreadsheet-like table. The columns could be sorted, making it possible to order contacts by the number of messages they sent to the user, or by the domain name of the contact, and so on. This allowed users to examine and order the extracted contacts in multiple ways, while making their choices. One concern is that the columns in the table may have biased users to focus on particular contact characteristics. However, pilot studies showed that without techniques to systematically sort and view data, users quickly became overwhelmed by the task of judging hundreds of contacts. We asked users to select important contacts for inclusion in their contact management system. They were told to choose contacts based on whether ‘you might want to be in contact with them again’. Users could make three possible judgments. They could decide that contacts: (a) should be added to their contact management system, i.e. that they were worth keeping in touch with; (b) should be excluded from the system, i.e. they were not worth keeping in touch with; and (c) that they were unsure of the status of the contact.

3.6 Results:

Characteristics of Extracted Contacts and the Selection Process Before testing our hypotheses, we present some general observations about the characteristics of the original archives and the set of contacts our users rated as important. We also present some observations about the selection process.

Chapter 4

4. Comprison of field and lab studies

Our field data suggested a significant, but currently under researched problem, that of contact management. People are exposed to large numbers of potential contacts, but the onerous nature of data entry means that they end up being conservative about who they add to their contact management systems. Despite this, people have a large number of contacts that they have to manage, but end up using a variety of ad hoc tools for this purpose. Our experimental results confirm the interview data in two important respects. First, consistent with the interview data, people are exposed to a large number of contacts (859 on average), only 19% of whom they judge as important. This supports the idea that people are exposed to many more contacts than they want to keep in touch with. This in turn suggests that contact selection is an important process. Second, the experiment confirmed the criteria that our interviewees suggested for identifying important contacts. We found evidence that a contact's communication history, and communication style were important determinants of whether a contact was selected. Frequency, reciprocity, longevity, and recency predicted subjective importance, as did contact affiliation and the style of the user's communication.

Chapter 5

5. Design and theory implications

Several design suggestions follow from these results. First, our regression analysis is a model for identifying important contacts in email, and this could be implemented directly as an algorithm. The ability to automatically identify important contacts from communication archives might be used in a number of applications, allowing us to improve messaging applications, support reminding and provide social recommendation. Messaging applications are currently poorly integrated with contact management tools, but future systems could exploit information about important contacts in a variety of ways. These might include alerting, filtering and prioritization of incoming email or voicemail messages based on the sender's importance. Tighter integration of contact information with messaging logs could be also used to manage relationships with contacts, e.g. reminding the user when they haven't talked to an important contact in a long time. We have implemented contact-based alerting and reminding in a social network-based user interface to communication and information . Finally social recommendation systems might be able to exploit information about a register of important contacts to either direct a user query or guide information access. Other design implications concern contacts management tools directly. We could improve address book utility by using our algorithm to automatically recommend that a potentially important contact should be added to the address book, based on their communication history. But even if we provide ways to better identify significant contacts, data entry is still a major problem. One possible way to address this would be to identify contact information from other sources, such as Internet home pages containing addresses. We may also be able to mine other types of records such as phone and voicemail logs or use reverse lookup to provide detailed addresses for contacts. Having general techniques for populating address books is clearly important.

Chapter 6

6.Hardware and Software specifications

Processor	Intel Pentium iv
Processor Speed	1GHz to 2GHz
RAM	4GB TO 16 GB
Hard Disk	8GB to 1TB
Keyboard	104 keys
Language	Python
Database	PytonDB (SQLite databases)
Operating System	Windows 7 /10

Chapter 7

7. About Python

Python is a high-level, interpreted, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.

Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

Python is a multi-paradigm programming language. Object-oriented programming and structured programming are fully supported, and many of its features support functional programming and aspect-oriented programming (including by metaprogramming and metaobjects [magic methods]). Many other paradigms are supported via extensions, including design by contract and logic programming.

Python uses dynamic typing, and a combination of reference counting and a cycle-detecting garbage collector for memory management. It uses dynamic name resolution (late binding), which binds method and variable names during program execution.

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Its design offers some support for functional programming in the Lisp tradition. It has filter,mapandreduce functions; list comprehensions, dictionaries, sets, and generator expressions. The standard library has two modules (itertools and functools) that implement functional tools borrowed

from Haskell and Standard ML.

Its core philosophy is summarized in the document The Zen of Python (PEP 20), which includes aphorisms such as

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Readability counts.

Rather than building all of its functionality into its core, Python was designed to be highly extensible via modules. This compact modularity has made it particularly popular as a means of adding programmable interfaces to existing applications. Van Rossum's vision of a small core language with a large standard library and easily extensible interpreter stemmed from his frustrations with ABC, which espoused the opposite approach.

Python strives for a simpler, less-cluttered syntax and grammar while giving developers a choice in their coding methodology. In contrast to Perl's "there is more than one way to do it" motto, Python embraces a "there should be one—and preferably only one—obvious way to do it" philosophy. Alex Martelli, a Fellow at the Python Software Foundation and Python book author, wrote: "To describe something as 'clever' is not considered a compliment in the Python culture."

Python's developers strive to avoid premature optimization and reject patches to non-critical parts of the CPython reference implementation that would offer marginal increases in speed at the cost of clarity. When speed is important, a Python programmer can move time-critical functions to extension modules written in languages such as C; or use PyPy, a just-in-time compiler. Cython is also available, which translates a Python script into C and makes direct C-level API calls into the Python interpreter.

Python's developers aim for it to be fun to use. This is reflected in its name—a tribute to the British comedy group Monty Python—and in occasionally playful approaches to tutorials and reference materials, such as examples that refer to spam and eggs (a reference to a Monty Python sketch) instead of the standard foo and bar.

A common neologism in the Python community is pythonic, which has a wide range of meanings related to program style. "Pythonic" code may use Python idioms well, be natural or show fluency in the language, or conform with Python's minimalist philosophy and emphasis on readability. Code that is difficult to understand or reads like a rough transcription from another programming language is called unpythonic.

Python users and admirers, especially those considered knowledgeable or experienced, are often referred to as Pythonistas

Methods

Methods on objects are functions attached to the object's class; the syntax instance.method(argument) is, for normal methods and functions, syntactic sugar for Class.method(instance, argument). Python methods have an explicit self parameter to access instance data, in contrast to the implicit self (or this) in some other object-oriented programming languages (e.g., C++, Java, Objective-C, Ruby). Python also provides methods, often called dunder methods (due to their names beginning and ending with double-underscores), to allow user-defined classes to modify how they are handled by native operations including length, comparison, in arithmetic operations and type conversion.

Typing

The standard type hierarchy in Python 3

Python uses duck typing and has typed objects but untyped variable names. Type constraints are not checked at compile time; rather, operations on an object may fail, signifying that it is not of a suitable type. Despite being dynamically-typed, Python is strongly-typed, forbidding operations that are not well-defined (for example, adding a number to a string) rather than silently attempting to make sense of them.

Python allows programmers to define their own types using classes, most often used for object-oriented programming. New instances of classes are constructed by calling the class (for example, SpamClass() or EggsClass()), and the classes are instances of the metaclass type (itself an instance of itself), allowing metaprogramming and reflection.

Before version 3.0, Python had two kinds of classes (both using the same syntax): old-style and new-style,current Python versions only support the semantics new style.

The long-term plan is to support gradual typing. Python's syntax allows specifying static types, but they are not checked in the default implementation, CPython. An experimental optional static type-checker, mypy, supports compile-time type checking.

Chapter 8

8. SQLite databases

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by PEP 249, and requires SQLite 3.7.15 or newer.

To use the module, start by creating a Connection object that represents the database. Here the data will be stored in the example.db file:

```
import sqlite3

con = sqlite3.connect('example.db')

cur = con.cursor()
```

Design

Unlike client–server database management systems, the SQLite engine has no standalone processes with which the application program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the application program. Linking may be static or dynamic. The application program uses SQLite's functionality through simple function calls, which reduce latency in database access: function calls within a single process are more efficient than inter-process communication.

SQLite stores the entire database (definitions, tables, indices, and the data itself) as a single cross-platform file on a host machine. It implements this simple design by locking the entire database file during writing. SQLite read operations can be multitasked, though writes can only be performed sequentially.

Due to the server-less design, SQLite applications require less configuration than client–server databases. SQLite is called zero-conf because it does not require service management (such as startup scripts) or access control based on GRANT and passwords. Access control is handled by means of file-system permissions given to the database file itself. Databases in client–server systems use file-system permissions that give access to the database files only to the daemon process.

Another implication of the serverless design is that several processes may not be able to write to the database file. In server-based databases, several writers will all connect to the same daemon, which is able to handle its locks internally. SQLite, on the other hand, has to rely on file-system locks. It has less knowledge of the other processes that are accessing the database at the same time. Therefore, SQLite is not the preferred choice for write-intensive deployments.However, for

simple queries with little concurrency, SQLite performance profits from avoiding the overhead of passing its data to another process.

SQLite uses PostgreSQL as a reference platform. "What would PostgreSQL do" is used to make sense of the SQL standard. One major deviation is that, with the exception of primary keys, SQLite does not enforce type checking; the type of a value is dynamic and not strictly constrained by the schema (although the schema will trigger a conversion when storing, if such a conversion is potentially reversible). SQLite strives to follow Postel's rule.

Features

SQLite implements most of the SQL-92 standard for SQL, but lacks some features. For example, it only partially provides triggers and cannot write to views (however, it provides INSTEAD OF triggers that provide this functionality). Its support of ALTER TABLE statements is limited.

SQLite uses an unusual type system for a SQL-compatible DBMS: instead of assigning a type to a column as in most SQL database systems, types are assigned to individual values; in language terms it is dynamically typed. Moreover, it is weakly typed in some of the same ways that Perl is: one can insert a string into an integer column (although SQLite will try to convert the string to an integer first, if the column's preferred type is integer). This adds flexibility to columns, especially when bound to a dynamically typed scripting language. However, the technique is not portable to other SQL products. A common criticism is that SQLite's type system lacks the data integrity mechanism provided by statically typed columns in other products. The SQLite web site describes a "strict affinity" mode, but this feature has not yet been added. However, it can be implemented with constraints like CHECK(typeof(x)='integer').

Tables normally include a hidden rowid index column, which gives faster access. If a database includes an Integer Primary Key column, SQLite will typically optimize it by treating it as an alias for rowid, causing the contents to be stored as a strictly typed 64-bit signed integer and changing its behavior to be somewhat like an auto-incrementing column. Future[when?] versions of SQLite may include a command to introspect whether a column has behavior like that of rowid to differentiate these columns from weakly typed, non-autoincrementing Integer Primary Keys.[failed verification]

Full support for Unicode case-conversions can be optionally be enabled through an extension.

Several computer processes or threads may access the same database concurrently. Several read accesses can be satisfied in parallel. A write access can only be satisfied if no other accesses are currently being serviced. Otherwise, the write access fails with an error code (or can automatically be retried until a configurable timeout expires). This concurrent access situation would change when dealing with temporary tables. This restriction is relaxed in version 3.7 when write-ahead logging (WAL) is turned on, enabling concurrent reads and writes.

Version 3.6.19 released on October 14, 2009 added support for foreign key constraints.

SQLite version 3.7.4 first saw the addition of the FTS4 (full-text search) module, which features enhancements over the older FTS3 module.FTS4 allows users to perform full-text searches on documents similar to how search engines search webpages. Version 3.8.2 added support for creating tables without rowid,[29] which may provide space and performance improvements. Common table expressions support was added to SQLite in version 3.8.3.[31] 3.8.11 added a newer search module called FTS5, the more radical (compared to FTS4) changes requiring a bump in version.

In 2015, with the json1 extension and new subtype interfaces, SQLite version 3.9 introduced JSON content managing.

As of version 3.33.0, the maximum supported database size is 281 TB.

Development and distribution

SQLite's code is hosted with Fossil, a distributed version control system that is itself built upon an SQLite database.

A standalone command-line program is provided in SQLite's distribution. It can be used to create a database, define tables, insert and change rows, run queries and manage an SQLite database file. It also serves as an example for writing applications that use the SQLite library.

SQLite uses automated regression testing prior to each release. Over 2 million tests are run as part of a release's verification. Starting with the August 10, 2009 release of SQLite 3.6.17, SQLite releases have 100% branch test coverage, one of the components of code coverage. The tests and test harnesses are partially public-domain and partially proprietary.

Chapter 9

9. Software development process

In software engineering, a software development process is a process of dividing software development work into smaller, parallel, or sequential steps or sub-processes to improve design, product management. It is also known as a software development life cycle (SDLC). The methodology may include the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application.

Most modern development processes can be vaguely described as agile. Other methodologies include waterfall, prototyping, iterative and incremental development, spiral development, rapid application development, and extreme programming.

A life-cycle "model" is sometimes considered a more general term for a category of methodologies and a software development "process" a more specific term to refer to a specific process chosen by a specific organization.[citation needed] For example, there are many specific software development processes that fit the spiral life-cycle model. The field is often considered a subset of the systems development life cycle.

Prototyping

Software prototyping is about creating prototypes, i.e. incomplete versions of the software program being developed.

The basic principles are:

Prototyping is not a standalone, complete development methodology, but rather an approach to try out particular features in the context of a full methodology (such as incremental, spiral, or rapid application development (RAD)).

Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

The client is involved throughout the development process, which increases the likelihood of client acceptance of the final implementation.

While some prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.

A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problems, but this is true for all software methodologies.

Methodologies

Agile development

Main article: Agile software development

"Agile software development" refers to a group of software development frameworks based on iterative development, where requirements and solutions evolve via collaboration between self-

organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto was formulated.

Agile software development uses iterative development as a basis but advocates a lighter and more people-centric viewpoint than traditional approaches. Agile processes fundamentally incorporate iteration and the continuous feedback that it provides to successively refine and deliver a software system.

The Agile model also includes the following software development processes:

Dynamic systems development method (DSDM)

Kanban

Scrum

Crystal

Atern

Lean software development

Continuous integration

Main article: Continuous integration

Continuous integration is the practice of merging all developer working copies to a shared mainline several times a day.[5] Grady Booch first named and proposed CI in his 1991 method, although he did not advocate integrating several times a day. Extreme programming (XP) adopted the concept of CI and did advocate integrating more than once per day – perhaps as many as tens of times per day.

Incremental development

Main article: Iterative and incremental development

Various methods are acceptable for combining linear and iterative systems development methodologies, with the primary objective of each being to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

There are three main variants of incremental development:[1]

A series of mini-Waterfalls are performed, where all phases of the Waterfall are completed for a small part of a system, before proceeding to the next increment, or

Overall requirements are defined before proceeding to evolutionary, mini-Waterfall development of individual increments of a system, or

The initial software concept, requirements analysis, and design of architecture and system core are defined via Waterfall, followed by incremental implementation, which culminates in installing the final version, a working system.

Rapid application development

Main article: Rapid application development

Rapid Application Development (RAD) Model

Rapid application development (RAD) is a software development methodology, which favors iterative development and the rapid construction of prototypes instead of large amounts of up-front planning. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

The rapid development process starts with the development of preliminary data models and business process models using structured techniques. In the next stage, requirements are verified using prototyping, eventually to refine the data and process models. These stages are repeated iteratively; further development results in "a combined business requirements and technical design statement to be used for constructing new systems".

The term was first used to describe a software development process introduced by James Martin in 1991. According to Whitten (2003), it is a merger of various structured techniques, especially data-driven information technology engineering, with prototyping techniques to accelerate software systems development.

The basic principles of rapid application development are:

Key objective is for fast development and delivery of a high quality system at a relatively low investment cost.
Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
Aims to produce high quality systems quickly, primarily via iterative Prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include Graphical User Interface (GUI) builders, Computer Aided Software Engineering (CASE) tools, Database Management Systems (DBMS), fourth-generation programming languages, code generators, and object-oriented techniques.
Key emphasis is on fulfilling the business need, while technological or engineering excellence is of lesser importance.

Project control involves prioritizing development and defining delivery deadlines or “timeboxes”.

If the project starts to slip, emphasis is on reducing requirements to fit the timebox, not in increasing the deadline.

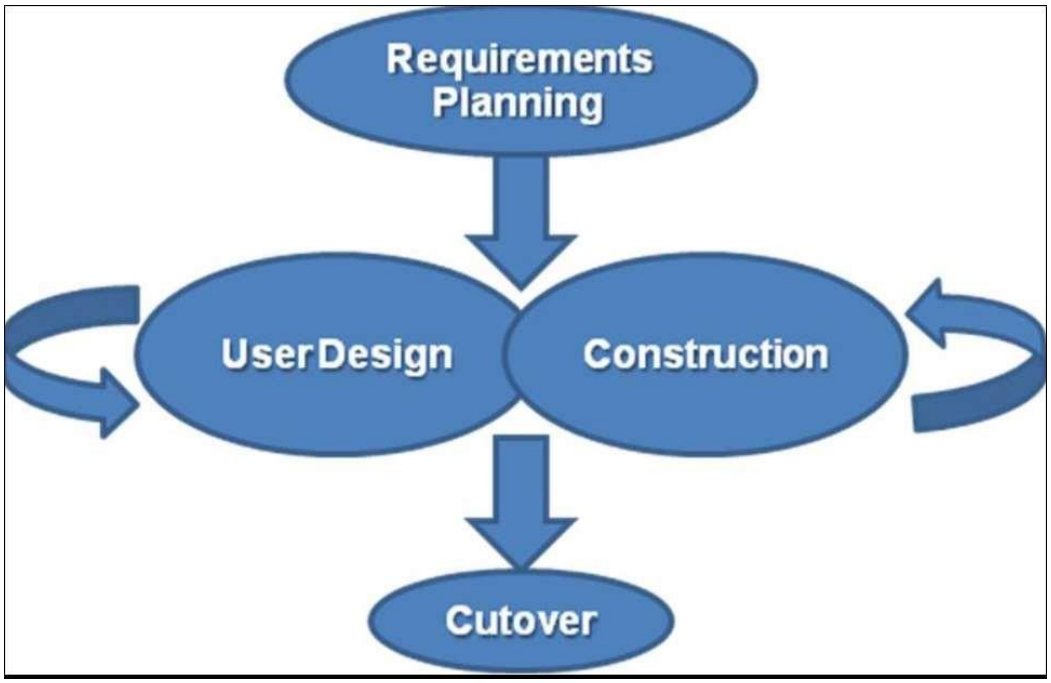
Generally includes joint application design (JAD), where users are intensely involved in system design, via consensus building in either structured workshops, or electronically facilitated interaction.

Active user involvement is imperative.

Iteratively produces production software, as opposed to a throwaway prototype.

Produces documentation necessary to facilitate future development and maintenance.

Standard systems analysis and design methods can be fitted into this framework.



Waterfall development

The waterfall model is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through several phases, typically:

Requirements analysis resulting in a software requirements specification

Software design

Implementation

Testing

Integration, if there are multiple subsystems

Deployment (or Installation)

Maintenance

The first formal description of the method is often cited as an article published by Winston W. Royce[8] in 1970, although Royce did not use the term "waterfall" in this article. Royce presented this model as an example of a flawed, non-working model.[9]

The basic principles are:

The Project is divided into sequential phases, with some overlap and splash back acceptable between phases.

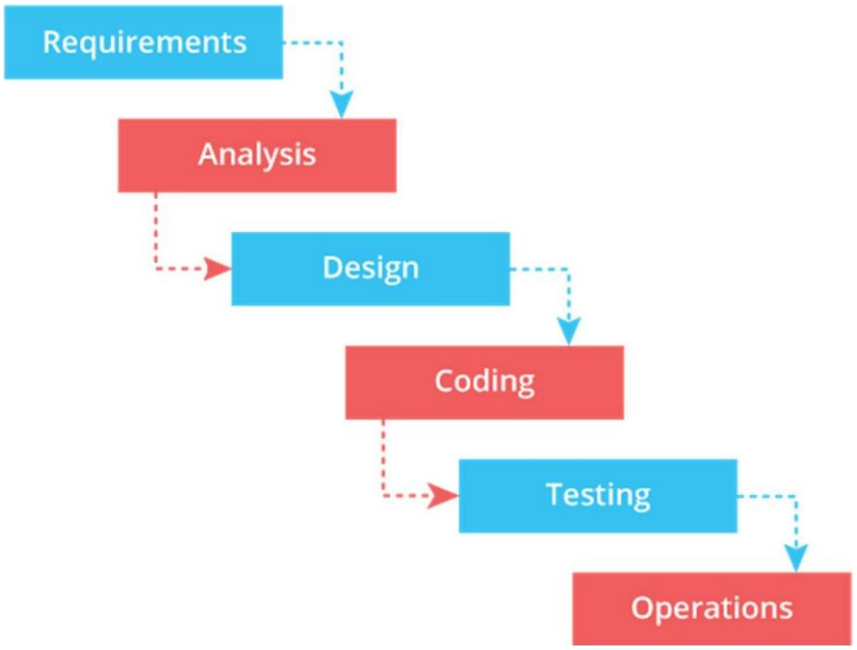
Emphasis is on planning, time schedules, target dates, budgets, and implementation of an entire system at one time.

Tight control is maintained over the life of the project via extensive written documentation, formal reviews, and approval/signoff by the user and information technology management occurring at the end of most phases before beginning the next phase. Written documentation is an explicit deliverable of each phase.

The waterfall model is a traditional engineering approach applied to software engineering. A

strict waterfall approach discourages revisiting and revising any prior phase once it is complete.[according to whom?] This "inflexibility" in a pure waterfall model has been a source of criticism by supporters of other more "flexible" models. It has been widely blamed for several large-scale government projects running over budget, over time and sometimes failing to deliver on requirements due to the Big Design Up Front approach.[according to whom?] Except when contractually required, the waterfall model has been largely superseded by more flexible and versatile methodologies developed specifically for software development.[according to whom?] See Criticism of Waterfall model

.



SOFTWARE DESIGN

Software design is the process by which an agent creates a specification of a software artifact intended to accomplish goals, using a set of primitive components and subject to constraints.Software design may refer to either "all the activity involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems" or "the activity following requirements specification and before programming, as ... [in] a stylized software engineering process."

Software design usually involves problem-solving and planning a software solution. This includes both a low-level component and algorithm design and a high-level, architecture design.

Software design is the process of envisioning and defining software solutions to one or more sets of problems. One of the main components of software design is the software requirements analysis (SRA). SRA is a part of the software development process that lists specifications used in software engineering. If the software is "semi-automated" or user centered, software design may involve user experience design yielding a storyboard to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case, some documentation of the plan is usually the product of the design. Furthermore, a software design may be platform-independent or platform-specific, depending upon the availability of the technology used for the design.

The main difference between software analysis and design is that the output of a software analysis consists of smaller problems to solve. Additionally, the analysis should not be designed very differently across different team members or groups. In contrast, the design focuses on capabilities, and thus multiple designs for the same problem can and will exist. Depending on the environment, the design often varies, whether it is created from reliable frameworks or implemented with suitable design patterns. Design examples include operation systems, webpages, mobile devices or even the new cloud computing paradigm.

Software design is both a process and a model. The design process is a sequence of steps that enables the designer to describe all aspects of the software for building. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are examples of critical success factors for a competent design. It is important to note, however, that the design process is not always a straightforward procedure; the design model can be compared to an architect's plans for a house. It begins by representing the totality of the thing that is to be built (e.g., a three-dimensional rendering of the house); slowly, the thing is refined to provide guidance for constructing each detail (e.g., the plumbing lay). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process. Davis[3] suggests a set of principles for software design, which have been adapted and extended in the following list:

The design process should not suffer from "tunnel vision." A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.

The design should be traceable to the analysis model. Because a single element of the design model can often be traced back to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.

The design should not reinvent the wheel. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited; design time should be invested in representing (truly new) ideas by integrating patterns that already exist (when applicable).

The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world. That is, the structure of the software design should, whenever possible, mimic the structure of the problem domain.

The design should exhibit uniformity and integration. A design is uniform if it appears fully coherent. In order to achieve this outcome, rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

The design should be structured to accommodate change. The design concepts discussed in the next section enable a design to achieve this principle.

The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. Well-designed software should never "bomb"; it should be designed to accommodate unusual circumstances, and if it must terminate processing, it should do so in a graceful manner.

Design is not coding, coding is not design. Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than the source code. The only design decisions made at the coding level should address the

small implementation details that enable the procedural design to be coded.

The design should be assessed for quality as it is being created, not after the fact. A variety of design concepts and design measures are available to assist the designer in assessing quality throughout the development process.

The design should be reviewed to minimize conceptual (semantic) errors. There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

Chapter 10

10. Data Flow Diagram (DFD)

DFD is an important tool used by system analysis. A data flow diagram model, a system using external entities from which data flows to a process which transforms the data and create output data transforms which go to other processes or external entities such as files. The main merit of DFD is that it can provide anoverview of what data a system would process.

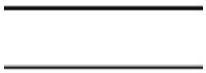
SYMBOLS

- A Circle represents a process that transforms incoming data flow into outgoing data flows
- A Square defines a source or destination of system data
- An Arrow identifies data flow direction. It is the pipeline through which the information flows.
- An Open Rectangle is a data store, data at rest or a temporary repository of data.

Data flow diagram symbol

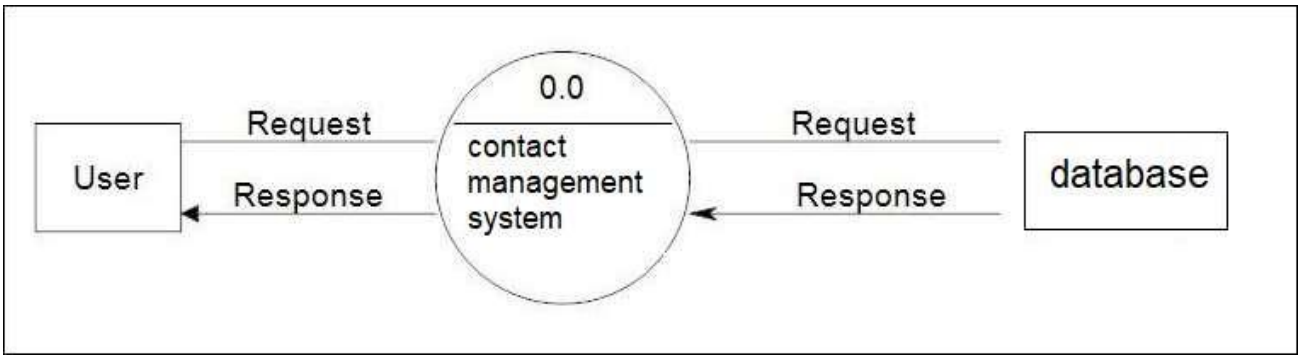
 Data Flow – Data flow are pipelines through the packets of information flow.  Process : A Process or task performed by the system.

 Entity : Entity are object of the system. A source or destination data of a system.

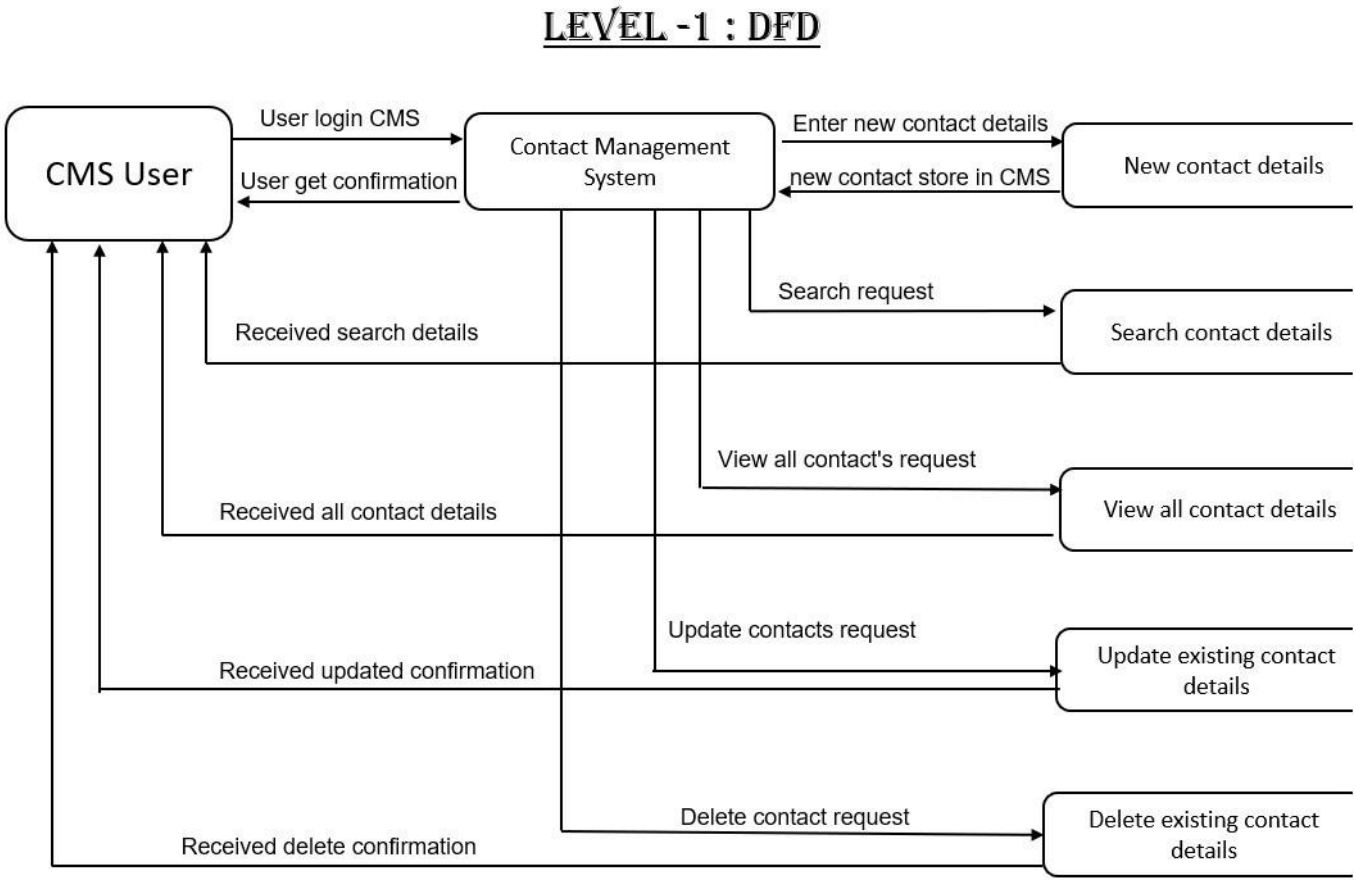
 Data Store : A place where data to be stored.

10.1. Context level DFD – 0 level

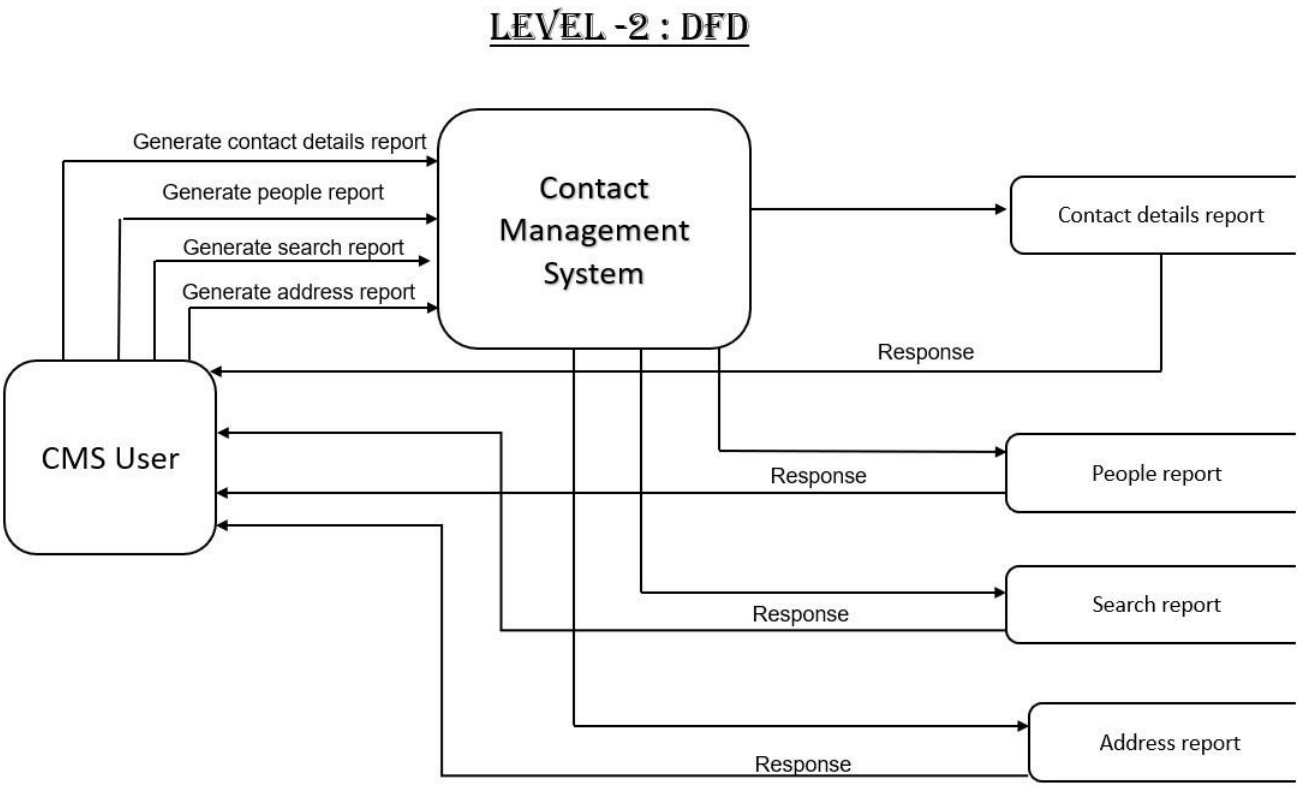
The context level data flow diagram (dfd) is describing the whole system. The (o) level dfd describe the all-user module who operate the system. Below data flow diagram of contact management system application shows the workflow of the process.



10.2. DFD 1 level



10.3. DFD 2 level

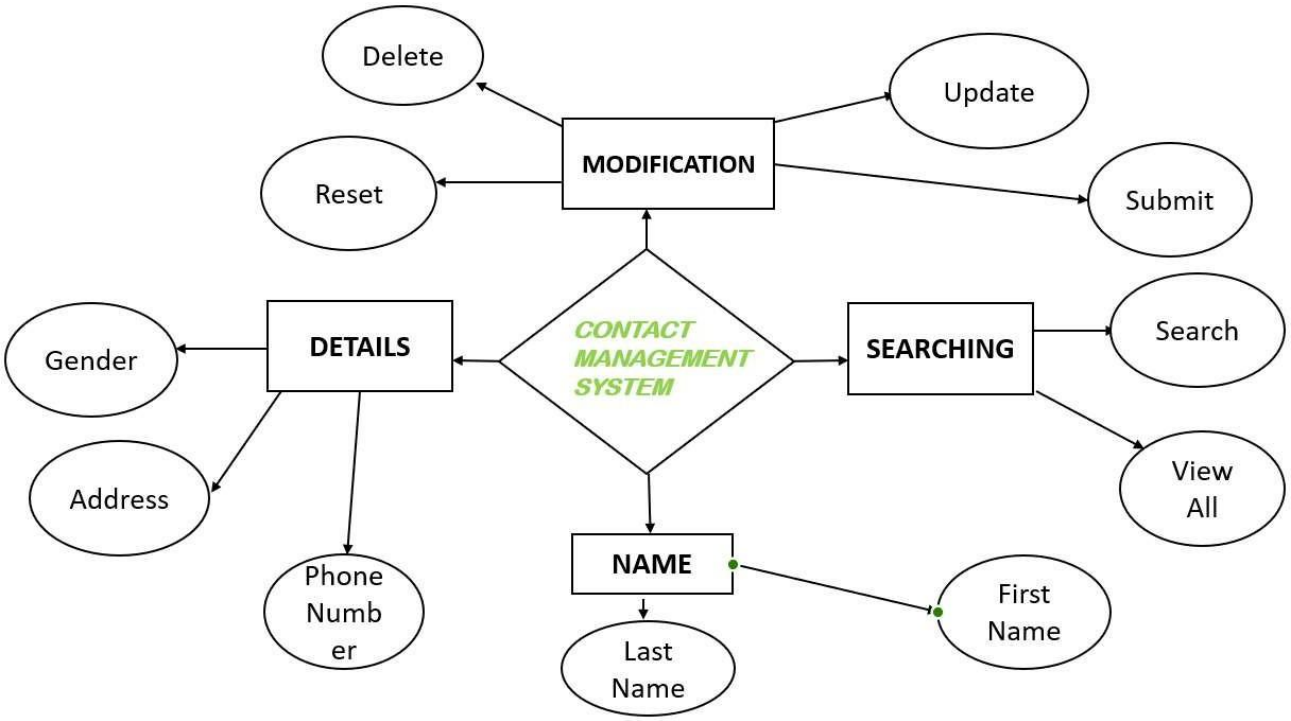


10.4. ENTITY-RELATIONSHIP Diagram

A flowchart is a type of diagram that represents a workflow or process. A flowchart can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task. The flowchart showsthe steps as boxes of various kinds, and their order by connecting the boxes with arrows.

Flow Chart Symbol	Meaning	Explanation
	Start and end	The symbol denoting the beginning and end of the flow chart.
	Step	This symbol shows that the user performs a task. (Note: In many flow charts steps and actions are interchangeable.)
	Decision	This symbol represents a point where a decision is made.
	Action	This symbol means that the user performs an action. (Note: In many flow charts steps and actions are interchangeable.)
	Flow line	A line that connects the various symbols in an ordered way.

10.5. ER Diagram



Chapter 11

11. Program Code and Testing

```
#import libraries
from tkinter import *
import tkinter.ttk as ttk
import tkinter.messagebox as tkMessageBox
import sqlite3
import string
import phonenumbers
#function to define database


def  Database():
    global conn, cursor
    #creating contact database
    conn = sqlite3.connect("contactdb.db")
    cursor = conn.cursor()
    #creating REGISTRATION table
    cursor.execute(
        "CREATE TABLE IF NOT EXISTS REGISTRATION (RID INTEGER PRIMARY
KEY AUTOINCREMENT NOT NULL, FNAME TEXT, LNAME TEXT, GENDER TEXT,
ADDRESS TEXT, CONTACT TEXT)")
      # "CREATE TABLE IF NOT EXISTS REGISTRATION_New (RID INTEGER
PRIMARY KEY AUTOINCREMENT NOT NULL, FNAME TEXT, LNAME TEXT,
GENDER TEXT, ADDRESS TEXT, CONTACT TEXT)")
```

```
    #defining function for creating GUI Layout
def DisplayForm():
    #creating window
    display_screen = Tk()
    #setting width and height for window
    display_screen.geometry("1200x500")
    #setting title for window
    display_screen.title("Developed by Sudhanshu Kumar (AJU/190323) ")
    global tree
    global btn_submit
    global SEARCH
    global fname,lname,gender,address,contact
    btn_submit= StringVar()
    SEARCH = StringVar()
    fname  =  StringVar()
    lname = StringVar()
    gender = StringVar()
    address = StringVar()
    contact = StringVar()
    #creating frames for layout
    #topview frame for heading
    TopViewForm = Frame(display_screen, width=600, bd=1, relief=SOLID)
    TopViewForm.pack(side=TOP, fill=X)
    #first left frame for registration from
    LFrom = Frame(display_screen, width="350",bg="#15244C")
    LFrom.pack(side=LEFT, fill=Y)
    #seconf left frame for search form
```

```
    LeftViewForm = Frame(display_screen, width=500,bg="#0B4670")
    LeftViewForm.pack(side=LEFT, fill=Y)
    #mid frame for displaying lnames record
    MidViewForm = Frame(display_screen, width=600)
    MidViewForm.pack(side=RIGHT)
    #label for heading
    lbl_text = Label(TopViewForm, text="Contact Management System", font=('verdana',
18), width=600,bg="cyan")
    lbl_text.pack(fill=X)
    #creating registration form in first left frame
    Label(LFrom, text="First Name  ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    Entry(LFrom,font=("Arial",10,"bold"),textvariable=fname).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Last Name ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    Entry(LFrom, font=("Arial", 10, "bold"),textvariable=lname).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Gender ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    #Entry(LFrom, font=("Arial", 10, "bold"),textvariable=gender).pack(side=TOP, padx=10,
fill=X)
    gender.set("Select Gender")
    content={'Male','Female'}
    OptionMenu(LFrom,gender,*content).pack(side=TOP, padx=10, fill=X)
```

```
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Address ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    Entry(LFrom, font=("Arial", 10, "bold"),textvariable=address).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Contact Number ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    Entry(LFrom, font=("Arial", 10, "bold"),textvariable=contact).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    btn_submit = Button(LFrom,text="Submit",font=("Arial", 10,
"bold"),command=register,bg="cyan")
    btn_submit.pack(side=TOP, padx=10,pady=5, fill=X)

    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="*All fields are required", font=("Arial", 8), bg="#15244C",
fg="white").pack(side=TOP)

    #creating search label and entry in second frame
    lbl_txtsearch = Label(LeftViewForm, text="Enter first name to Search", font=('verdana',
8),bg="#0B4670",fg="white")
    lbl_txtsearch.pack()
    #creating search entry
    search = Entry(LeftViewForm, textvariable=SEARCH, font=('verdana', 15), width=10)
    search.pack(side=TOP, padx=10, fill=X)
```

```
        #creating search button
        btn_search = Button(LeftViewForm, text="Search", command=SearchRecord,bg="cyan")
        btn_search.pack(side=TOP, padx=10, pady=10, fill=X)
        #creating view button
        btn_view = Button(LeftViewForm, text="View All", command=DisplayData,bg="cyan")
        btn_view.pack(side=TOP, padx=10, pady=10, fill=X)
        #creating reset button
        btn_reset = Button(LeftViewForm, text="Reset", command=Reset,bg="cyan")
        btn_reset.pack(side=TOP, padx=10, pady=10, fill=X)
        #creating delete button
        btn_delete = Button(LeftViewForm, text="Delete", command=Delete,bg="cyan")
        btn_delete.pack(side=TOP, padx=10, pady=10, fill=X)
        #create update button
        btn_delete = Button(LeftViewForm, text="Update", command=Update,bg="cyan")
        btn_delete.pack(side=TOP, padx=10, pady=10, fill=X)
        #setting scrollbar
        scrollbarx = Scrollbar(MidViewForm, orient=HORIZONTAL)
        scrollbary = Scrollbar(MidViewForm, orient=VERTICAL)
        tree = ttk.Treeview(MidViewForm,columns=("Contact Id", "FName", "LName",
"Gender","Address","Contact"),
                            selectmode="extended", height=100, yscrollcommand=scrollbary.set,
xscrollcommand=scrollbarx.set)
        scrollbary.config(command=tree.yview)
        scrollbary.pack(side=RIGHT, fill=Y)
        scrollbarx.config(command=tree.xview)
        scrollbarx.pack(side=BOTTOM, fill=X)
        #setting headings for the columns
```

```
    tree.heading('Contact Id', text="Contact Id", anchor=W)
    tree.heading('FName', text="FirstName", anchor=W)
    tree.heading('LName', text="LastName", anchor=W)
    tree.heading('Gender', text="Gender", anchor=W)
    tree.heading('Address', text="Address", anchor=W)
    tree.heading('Contact', text="Contact", anchor=W)
    #setting width of the columns
    tree.column('#0', stretch=NO, minwidth=0, width=0)
    tree.column('#1', stretch=NO, minwidth=0, width=80)
    tree.column('#2', stretch=NO, minwidth=0, width=150)
    tree.column('#3', stretch=NO, minwidth=0, width=150)
    tree.column('#4', stretch=NO, minwidth=0, width=80)
    tree.pack()
    DisplayData()
#function to update data into database
def Update():
    Database()
    #getting form data
    fname1=fname.get()
    lname1=lname.get()
    gender1=gender.get()
    address1=address.get()
    contact1=contact.get()
    #applying empty validation
    if fname1=='' or lname1=="or gender1=='' or address1=="or contact1=='':
        tkMessageBox.showinfo("Warning","fill the empty field!!!")
    else:
```

```
        if phone(contact1) == 'false':
            tkMessageBox.showinfo("Warning",
                                  "The phone number must be number with containing country code
'+91 ....."")
        else:
            #getting selected data
            phone(contact1)
            curItem = tree.focus()
            contents = (tree.item(curItem))
            selecteditem = contents['values']
            #update query
            conn.execute('UPDATE REGISTRATION SET
FNAME=?,LNAME=?,GENDER=?,ADDRESS=?,CONTACT=? WHERE RID =
?',(fname1,lname1,gender1,address1,contact1, selecteditem[0]))
            conn.commit()
            tkMessageBox.showinfo("Message","Updated successfully")
            #reset form
            Reset()
            #refresh table data
            DisplayData()
            conn.close()


  def register():
      Database()
      #getting form data
      fname1=fname.get()
      lname1=lname.get()
```

```
    gender1=gender.get()
    address1=address.get()
    contact1=contact.get()
    #applying empty validation
    if fname1==" or lname1=="or gender1=='Select Gender' or address1=="or contact1==":
        tkMessageBox.showinfo("Warning","fill the empty field!!!")
    else:
        if phone(contact1) == 'false':
            tkMessageBox.showinfo("Warning", "The phone number must be number with
containing country code '+91..... "')


        else:
            #execute query
            conn.execute('INSERT INTO REGISTRATION
(FNAME,LNAME,GENDER,ADDRESS,CONTACT) \
              VALUES (?,?,?,?,?)',(fname1,lname1,gender1,address1,contact1));
            conn.commit()
            tkMessageBox.showinfo("Message","Thank you, your contact has been stored
successfully")
            #refresh table data
            DisplayData()
            conn.close()
  def phone(phone):
    print
    "Error. I am in phone function ....."
    string_phone_number =phone
    try:
```

```
            phone_number = phonenumbers.parse(string_phone_number)
            returnvalue=phonenumbers.is_possible_number(phone_number)
        except Exception as e:
            print(e)
            return 'false'


    def Reset():
        #clear current data from table
        tree.delete(*tree.get_children())
        #refresh table data
        DisplayData()
        #clear search text
        SEARCH.set("")
        fname.set("")
        lname.set("")
        gender.set("")
        address.set("")
        contact.set("")
        btn_submit['state'] = 'normal'
    def Delete():
        #open database
        Database()
        if not tree.selection():
            tkMessageBox.showwarning("Warning","Select data to delete")
        else:
            result = tkMessageBox.askquestion('Confirm', 'Are you sure you want to delete this
record?',
```

```
                                    icon="warning")
        if result == 'yes':
            curItem = tree.focus()
            contents = (tree.item(curItem))
            selecteditem = contents['values']
            tree.delete(curItem)
            #cursor=conn.execute("DELETE * FROM REGISTRATION")
            cursor = conn.execute("DELETE FROM REGISTRATION WHERE RID = %d" %
selecteditem[0])
            conn.commit()
            cursor.close()
            conn.close()

  #function to search data
  def SearchRecord():
      #open database
      Database()

      #checking search text is empty or not
      if SEARCH.get() == "":
          tkMessageBox.showinfo("Warning", "Please enter first name in required search field
!!")
      else:
          #clearing current display data
          tree.delete(*tree.get_children())
          #select query with where clause
```

```
        cursor=conn.execute("SELECT * FROM REGISTRATION WHERE FNAME LIKE
?", ('%' + str(SEARCH.get()) + '%',))
        #fetch all matching records

        fetch = cursor.fetchall()

        #loop for displaying all records into GUI
        for data in fetch:
            tree.insert('', 'end', values=(data))
        cursor.close()
        conn.close()
  #defining function to access data from SQLite database
  def DisplayData():
     #open database
     Database()
     #clear current data
     tree.delete(*tree.get_children())
     #select query
     cursor=conn.execute("SELECT * FROM REGISTRATION")
     #fetch all data from database
     fetch = cursor.fetchall()
     #loop for displaying all data in GUI
     for data in fetch:
        tree.insert('', 'end', values=(data))
        tree.bind("<Double-1>",OnDoubleClick)
     cursor.close()
     conn.close()
```

```
def OnDoubleClick(self):
    #getting focused item from treeview
    curItem = tree.focus()
    contents = (tree.item(curItem))
    selecteditem = contents['values']
    #set values in the fields
    fname.set(selecteditem[1])
    lname.set(selecteditem[2])
    gender.set(selecteditem[3])
    address.set(selecteditem[4])
    contact.set(selecteditem[5])
    btn_submit['state'] = 'disabled'

#calling function
DisplayForm()
if __name__=='__main__':
    #Running Application
    mainloop()



#import libraries
from tkinter import *
import tkinter.ttk as ttk
import tkinter.messagebox as tkMessageBox
import sqlite3
import string
import phonenumbers
```

```
#function to define database


def  Database():
    global conn, cursor
    #creating contact database
    conn = sqlite3.connect("contactdb.db")
    cursor = conn.cursor()
    #creating REGISTRATION table
    cursor.execute(
       "CREATE TABLE IF NOT EXISTS REGISTRATION (RID INTEGER PRIMARY
KEY AUTOINCREMENT NOT NULL, FNAME TEXT, LNAME TEXT, GENDER TEXT,
ADDRESS TEXT, CONTACT TEXT)")
      # "CREATE TABLE IF NOT EXISTS REGISTRATION_New (RID INTEGER
PRIMARY KEY AUTOINCREMENT NOT NULL, FNAME TEXT, LNAME TEXT,
GENDER TEXT, ADDRESS TEXT, CONTACT TEXT)")


    #defining function for creating GUI Layout
def DisplayForm():
    #creating window
    display_screen = Tk()
    #setting width and height for window
    display_screen.geometry("1200x500")
    #setting title for window
    display_screen.title("Developed by Sudhanshu Kumar (AJU/190323) ")
    global tree
    global btn_submit
    global SEARCH
```

```
    global fname,lname,gender,address,contact
    btn_submit= StringVar()
    SEARCH = StringVar()
    fname  =  StringVar()
    lname = StringVar()
    gender = StringVar()
    address = StringVar()
    contact = StringVar()
    #creating frames for layout
    #topview frame for heading
    TopViewForm = Frame(display_screen, width=600, bd=1, relief=SOLID)
    TopViewForm.pack(side=TOP, fill=X)
    #first left frame for registration from
    LFrom = Frame(display_screen, width="350",bg="#15244C")
    LFrom.pack(side=LEFT, fill=Y)
    #seconf left frame for search form
    LeftViewForm = Frame(display_screen, width=500,bg="#0B4670")
    LeftViewForm.pack(side=LEFT, fill=Y)
    #mid frame for displaying lnames record
    MidViewForm = Frame(display_screen, width=600)
    MidViewForm.pack(side=RIGHT)
    #label for heading
    lbl_text = Label(TopViewForm, text="Contact Management System", font=('verdana',
18), width=600,bg="cyan")
    lbl_text.pack(fill=X)
    #creating registration form in first left frame
```

```
    Label(LFrom, text="First Name ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    Entry(LFrom,font=("Arial",10,"bold"),textvariable=fname).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Last Name ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    Entry(LFrom, font=("Arial", 10, "bold"),textvariable=lname).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Gender ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    #Entry(LFrom, font=("Arial", 10, "bold"),textvariable=gender).pack(side=TOP, padx=10,
fill=X)
    gender.set("Select Gender")
    content={'Male','Female'}
    OptionMenu(LFrom,gender,*content).pack(side=TOP, padx=10, fill=X)

    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Address ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
    Entry(LFrom, font=("Arial", 10, "bold"),textvariable=address).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="Contact Number ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
```

```
    Entry(LFrom, font=("Arial", 10, "bold"),textvariable=contact).pack(side=TOP, padx=10,
fill=X)
    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    btn_submit = Button(LFrom,text="Submit",font=("Arial", 10,
"bold"),command=register,bg="cyan")
    btn_submit.pack(side=TOP, padx=10,pady=5, fill=X)

    Label(LFrom, text="", font=("Arial", 12), bg="#15244C", fg="white").pack(side=TOP)
    Label(LFrom, text="*All fields are required", font=("Arial", 8), bg="#15244C",
fg="white").pack(side=TOP)

    #creating search label and entry in second frame
    lbl_txtsearch = Label(LeftViewForm, text="Enter first name to Search", font=('verdana',
8),bg="#0B4670",fg="white")
    lbl_txtsearch.pack()
    #creating search entry
    search = Entry(LeftViewForm, textvariable=SEARCH, font=('verdana', 15), width=10)
    search.pack(side=TOP, padx=10, fill=X)
    #creating search button
    btn_search = Button(LeftViewForm, text="Search", command=SearchRecord,bg="cyan")
    btn_search.pack(side=TOP, padx=10, pady=10, fill=X)
    #creating view button
    btn_view = Button(LeftViewForm, text="View All", command=DisplayData,bg="cyan")
    btn_view.pack(side=TOP, padx=10, pady=10, fill=X)
    #creating reset button
    btn_reset = Button(LeftViewForm, text="Reset", command=Reset,bg="cyan")
    btn_reset.pack(side=TOP, padx=10, pady=10, fill=X)
```

```
    #creating delete button
    btn_delete = Button(LeftViewForm, text="Delete", command=Delete,bg="cyan")
    btn_delete.pack(side=TOP, padx=10, pady=10, fill=X)
    #create update button
    btn_delete = Button(LeftViewForm, text="Update", command=Update,bg="cyan")
    btn_delete.pack(side=TOP, padx=10, pady=10, fill=X)
    #setting scrollbar
    scrollbarx = Scrollbar(MidViewForm, orient=HORIZONTAL)
    scrollbary = Scrollbar(MidViewForm, orient=VERTICAL)
    tree = ttk.Treeview(MidViewForm,columns=("Contact Id", "FName", "LName",
"Gender","Address","Contact"),
                        selectmode="extended", height=100, yscrollcommand=scrollbary.set,
xscrollcommand=scrollbarx.set)
    scrollbary.config(command=tree.yview)
    scrollbary.pack(side=RIGHT, fill=Y)
    scrollbarx.config(command=tree.xview)
    scrollbarx.pack(side=BOTTOM, fill=X)
    #setting headings for the columns
    tree.heading('Contact Id', text="Contact Id", anchor=W)
    tree.heading('FName', text="FirstName", anchor=W)
    tree.heading('LName', text="LastName", anchor=W)
    tree.heading('Gender', text="Gender", anchor=W)
    tree.heading('Address', text="Address", anchor=W)
    tree.heading('Contact', text="Contact", anchor=W)
    #setting width of the columns
    tree.column('#0', stretch=NO, minwidth=0, width=0)
    tree.column('#1', stretch=NO, minwidth=0, width=80)
```

```
    tree.column('#2', stretch=NO, minwidth=0, width=150)
    tree.column('#3', stretch=NO, minwidth=0, width=150)
    tree.column('#4', stretch=NO, minwidth=0, width=80)
    tree.pack()
    DisplayData()
#function to update data into database
def Update():
    Database()
    #getting form data
    fname1=fname.get()
    lname1=lname.get()
    gender1=gender.get()
    address1=address.get()
    contact1=contact.get()
    #applying empty validation
    if fname1==" or lname1=="or gender1==" or address1=="or contact1==":
        tkMessageBox.showinfo("Warning","fill the empty field!!!")
    else:
        if phone(contact1) == 'false':
            tkMessageBox.showinfo("Warning",
                                  "The phone number must be number with containing country code
'+91 ..... '")
        else:
            #getting selected data
            phone(contact1)
            curItem = tree.focus()
            contents = (tree.item(curItem))
```

```
            selecteditem = contents['values']
            #update query
            conn.execute('UPDATE REGISTRATION SET
FNAME=?,LNAME=?,GENDER=?,ADDRESS=?,CONTACT=? WHERE RID =
?',(fname1,lname1,gender1,address1,contact1, selecteditem[0]))
            conn.commit()
            tkMessageBox.showinfo("Message","Updated successfully")
            #reset form
            Reset()
            #refresh table data
            DisplayData()
            conn.close()

  def register():
     Database()
     #getting form data
     fname1=fname.get()
     lname1=lname.get()
     gender1=gender.get()
     address1=address.get()
     contact1=contact.get()
     #applying empty validation
     if fname1=='' or lname1==''or gender1=='Select Gender' or address1==''or contact1=='':
        tkMessageBox.showinfo("Warning","fill the empty field!!!")
     else:
        if phone(contact1) == 'false':
```

```
            tkMessageBox.showinfo("Warning", "The phone number must be number with
containing country code '+91..... '")


        else:
            #execute query
            conn.execute('INSERT INTO REGISTRATION
(FNAME,LNAME,GENDER,ADDRESS,CONTACT) \
              VALUES (?,?,?,?,?)',(fname1,lname1,gender1,address1,contact1));
            conn.commit()
            tkMessageBox.showinfo("Message","Thank you, your contact has been stored
successfully")
            #refresh table data
            DisplayData()
            conn.close()
  def phone(phone):
     print
     "Error. I am in phone function ....."
     string_phone_number =phone
     try:
        phone_number = phonenumbers.parse(string_phone_number)
        returnvalue=phonenumbers.is_possible_number(phone_number)
     except Exception as e:
        print(e)
        return 'false'


  def Reset():
     #clear current data from table
```

```
        tree.delete(*tree.get_children())
        #refresh table data
        DisplayData()
        #clear search text
        SEARCH.set("")
        fname.set("")
        lname.set("")
        gender.set("")
        address.set("")
        contact.set("")
        btn_submit['state'] = 'normal'
    def Delete():
        #open database
        Database()
        if not tree.selection():
            tkMessageBox.showwarning("Warning","Select data to delete")
        else:
            result = tkMessageBox.askquestion('Confirm', 'Are you sure you want to delete this
record?',
                                              icon="warning")
            if result == 'yes':
                curItem = tree.focus()
                contents = (tree.item(curItem))
                selecteditem = contents['values']
                tree.delete(curItem)
                #cursor=conn.execute("DELETE * FROM REGISTRATION")
```

```
            cursor = conn.execute("DELETE FROM REGISTRATION WHERE RID = %d" %
selecteditem[0])
            conn.commit()
            cursor.close()
            conn.close()

  #function to search data
  def SearchRecord():
     #open database
     Database()

     #checking search text is empty or not
     if SEARCH.get() == "":
        tkMessageBox.showinfo("Warning", "Please enter first name in required search field
!!")
     else:
        #clearing current display data
        tree.delete(*tree.get_children())
        #select query with where clause
        cursor=conn.execute("SELECT * FROM REGISTRATION WHERE FNAME LIKE
?", ('%' + str(SEARCH.get()) + '%',))
        #fetch all matching records

        fetch = cursor.fetchall()

        #loop for displaying all records into GUI
        for data in fetch:
```

```
            tree.insert('', 'end', values=(data))
        cursor.close()
        conn.close()
#defining function to access data from SQLite database
def DisplayData():
    #open database
    Database()
    #clear current data
    tree.delete(*tree.get_children())
    #select query
    cursor=conn.execute("SELECT * FROM REGISTRATION")
    #fetch all data from database
    fetch = cursor.fetchall()
    #loop for displaying all data in GUI
    for data in fetch:
        tree.insert('', 'end', values=(data))
        tree.bind("<Double-1>",OnDoubleClick)
    cursor.close()
    conn.close()




def OnDoubleClick(self):
    #getting focused item from treeview
    curItem = tree.focus()
    contents = (tree.item(curItem))
```

```
    selecteditem = contents['values']
    #set values in the fields
    fname.set(selecteditem[1])
    lname.set(selecteditem[2])
    gender.set(selecteditem[3])
    address.set(selecteditem[4])
    contact.set(selecteditem[5])
    btn_submit['state'] = 'disabled'

#calling function
DisplayForm()
if __name_=='_main_':
    #Running Application
    mainloop()
```

Calling function

```
DisplayForm()
if __name_=='_main_':
    #Running Application
    mainloop()
```

function to define database

```
def  Database():
    global conn, cursor
    #creating contact database
    conn = sqlite3.connect("contactdb.db")
    cursor = conn.cursor()
    #creating REGISTRATION table
    cursor.execute(
        "CREATE TABLE IF NOT EXISTS REGISTRATION (RID INTEGER PRIMARY
KEY AUTOINCREMENT NOT NULL, FNAME TEXT, LNAME TEXT, GENDER TEXT,
ADDRESS TEXT, CONTACT TEXT)")
```

Defining function for creating GUI Layout

```
def DisplayForm():
    #creating window
    display_screen = Tk()
    #setting width and height for window
    display_screen.geometry("1200x500")
    #setting title for window
    display_screen.title("Developed by Sudhanshu Kumar (AJU/190323) ")
    global tree
    global btn_submit
```

```
        global SEARCH
        global fname,lname,gender,address,contact
        btn_submit= StringVar()
        SEARCH = StringVar()
        fname  =  StringVar()
        lname = StringVar()
        gender = StringVar()
        address = StringVar()
        contact = StringVar()
        #creating frames for layout
        #topview frame for heading
        TopViewForm = Frame(display_screen, width=600, bd=1, relief=SOLID)
        TopViewForm.pack(side=TOP, fill=X)
        #first left frame for registration from
        LFrom = Frame(display_screen, width="350",bg="#15244C")
        LFrom.pack(side=LEFT, fill=Y)
        #seconf left frame for search form
        LeftViewForm = Frame(display_screen, width=500,bg="#0B4670")
        LeftViewForm.pack(side=LEFT, fill=Y)
        #mid frame for displaying lnames record
        MidViewForm = Frame(display_screen, width=600)
        MidViewForm.pack(side=RIGHT)
        #label for heading
        lbl_text = Label(TopViewForm, text="Contact Management System", font=('verdana',
18), width=600,bg="cyan")
        lbl_text.pack(fill=X)
        #creating registration form in first left frame
```

```
        Label(LFrom, text="First Name ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
        Entry(LFrom,font=("Arial",10,"bold"),textvariable=fname).pack(side=TOP, padx=10,
fill=X)
        Label(LFrom, text="", font=("Arial", 12), bg="#15244C",
fg="white").pack(side=TOP)
        Label(LFrom, text="Last Name ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
        Entry(LFrom, font=("Arial", 10, "bold"),textvariable=lname).pack(side=TOP,
padx=10, fill=X)
        Label(LFrom, text="", font=("Arial", 12), bg="#15244C",
fg="white").pack(side=TOP)
        Label(LFrom, text="Gender ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
        #Entry(LFrom, font=("Arial", 10, "bold"),textvariable=gender).pack(side=TOP,
padx=10, fill=X)
        gender.set("Select Gender")
        content={'Male','Female'}
        OptionMenu(LFrom,gender,*content).pack(side=TOP, padx=10, fill=X)

        Label(LFrom, text="", font=("Arial", 12), bg="#15244C",
fg="white").pack(side=TOP)
        Label(LFrom, text="Address ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
        Entry(LFrom, font=("Arial", 10, "bold"),textvariable=address).pack(side=TOP,
padx=10, fill=X)
```

```
        Label(LFrom, text="", font=("Arial", 12), bg="#15244C",
fg="white").pack(side=TOP)
        Label(LFrom, text="Contact Number ", font=("Arial",
12),bg="#15244C",fg="white").pack(side=TOP)
        Entry(LFrom, font=("Arial", 10, "bold"),textvariable=contact).pack(side=TOP,
padx=10, fill=X)
        Label(LFrom, text="", font=("Arial", 12), bg="#15244C",
fg="white").pack(side=TOP)
        btn_submit = Button(LFrom,text="Submit",font=("Arial", 10,
"bold"),command=register,bg="cyan")
        btn_submit.pack(side=TOP, padx=10,pady=5, fill=X)
        Label(LFrom, text="", font=("Arial", 12), bg="#15244C",
fg="white").pack(side=TOP)
        Label(LFrom, text="*All fields are required", font=("Arial", 8), bg="#15244C",
fg="white").pack(side=TOP)

        #creating search label and entry in second frame
        lbl_txtsearch = Label(LeftViewForm, text="Enter first name to Search",
font=('verdana', 8),bg="#0B4670",fg="white")
        lbl_txtsearch.pack()
        #creating search entry
        search = Entry(LeftViewForm, textvariable=SEARCH, font=('verdana', 15), width=10)
        search.pack(side=TOP, padx=10, fill=X)
        #creating search button
        btn_search = Button(LeftViewForm, text="Search",
command=SearchRecord,bg="cyan")
        btn_search.pack(side=TOP, padx=10, pady=10, fill=X)
```

```
        #creating view button
        btn_view = Button(LeftViewForm, text="View All",
command=DisplayData,bg="cyan")
        btn_view.pack(side=TOP, padx=10, pady=10, fill=X)
        #creating reset button
        btn_reset = Button(LeftViewForm, text="Reset", command=Reset,bg="cyan")
        btn_reset.pack(side=TOP, padx=10, pady=10, fill=X)
        #creating delete button
        btn_delete = Button(LeftViewForm, text="Delete", command=Delete,bg="cyan")
        btn_delete.pack(side=TOP, padx=10, pady=10, fill=X)
        #create update button
        btn_delete = Button(LeftViewForm, text="Update", command=Update,bg="cyan")
        btn_delete.pack(side=TOP, padx=10, pady=10, fill=X)
        #setting scrollbar
        scrollbarx = Scrollbar(MidViewForm, orient=HORIZONTAL)
        scrollbary = Scrollbar(MidViewForm, orient=VERTICAL)
        tree = ttk.Treeview(MidViewForm,columns=("Contact Id", "FName", "LName",
"Gender","Address","Contact"),
                            selectmode="extended", height=100, yscrollcommand=scrollbary.set,
xscrollcommand=scrollbarx.set)
        scrollbary.config(command=tree.yview)
        scrollbary.pack(side=RIGHT, fill=Y)
        scrollbarx.config(command=tree.xview)
        scrollbarx.pack(side=BOTTOM, fill=X)
        #setting headings for the columns
        tree.heading('Contact Id', text="Contact Id", anchor=W)
        tree.heading('FName', text="FirstName", anchor=W)
```

```
    tree.heading('LName', text="LastName", anchor=W)
    tree.heading('Gender', text="Gender", anchor=W)
    tree.heading('Address', text="Address", anchor=W)
    tree.heading('Contact', text="Contact", anchor=W)
    #setting width of the columns
    tree.column('#0', stretch=NO, minwidth=0, width=0)
    tree.column('#1', stretch=NO, minwidth=0, width=80)
    tree.column('#2', stretch=NO, minwidth=0, width=150)
    tree.column('#3', stretch=NO, minwidth=0, width=150)
    tree.column('#4', stretch=NO, minwidth=0, width=80)
    tree.pack()
    DisplayData()
```

Function to update data into database

```
def Update():
    Database()
    #getting form data
    fname1=fname.get()
    lname1=lname.get()
    gender1=gender.get()
    address1=address.get()
    contact1=contact.get()
    #applying empty validation
```

```
    if fname1=='' or lname1=="or gender1=='' or address1=="or contact1=='':
        tkMessageBox.showinfo("Warning","fill the empty field!!!")
    else:
        if phone(contact1) == 'false':
            tkMessageBox.showinfo("Warning",
                                  "The phone number must be number with containing country code
'+91..... '")
        else:
            #getting selected data
            phone(contact1)
            curItem = tree.focus()
            contents = (tree.item(curItem))
            selecteditem = contents['values']
            #update query
            conn.execute('UPDATE REGISTRATION SET
FNAME=?,LNAME=?,GENDER=?,ADDRESS=?,CONTACT=? WHERE RID =
?',(fname1,lname1,gender1,address1,contact1, selecteditem[0]))
            conn.commit()
            tkMessageBox.showinfo("Message","Updated successfully")
            #reset form
            Reset()
            #refresh table data
            DisplayData()
            conn.close()
```

Function for new entry

```
def register():
    Database()
    #getting form data
    fname1=fname.get()
    lname1=lname.get()
    gender1=gender.get()
    address1=address.get()
    contact1=contact.get()
    #applying empty validation
    if fname1==" or lname1=="or gender1=='Select Gender' or address1=="or contact1==":
        tkMessageBox.showinfo("Warning","fill the empty field!!!")
    else:
        if phone(contact1) == 'false':
            tkMessageBox.showinfo("Warning", "The phone number must be number with containing
country code '+91..... '")


        else:
            #execute query
            conn.execute('INSERT INTO REGISTRATION
(FNAME,LNAME,GENDER,ADDRESS,CONTACT) \
              VALUES (?,?,?,?,?)',(fname1,lname1,gender1,address1,contact1));
            conn.commit()
            tkMessageBox.showinfo("Message","Thank you, your contact has been stored
successfully")
```

```
        #refresh table data
        DisplayData()
        conn.close()
```

Clear current data from table

```
    tree.delete(*tree.get_children())
#refresh table data
DisplayData()
#clear search text
SEARCH.set("")
fname.set("")
lname.set("")
gender.set("")
address.set("")
contact.set("")
btn_submit['state'] = 'normal'
```

Delete selected data from table

```
def Delete():
    #open database
    Database()
    if not tree.selection():
        tkMessageBox.showwarning("Warning","Select data to delete")
    else:
        result = tkMessageBox.askquestion('Confirm', 'Are you sure you want to delete this record?',
                                          icon="warning")
        if result == 'yes':
            curItem = tree.focus()
            contents = (tree.item(curItem))
            selecteditem = contents['values']
            tree.delete(curItem)
            #cursor=conn.execute("DELETE * FROM REGISTRATION")
            cursor = conn.execute("DELETE FROM REGISTRATION WHERE RID = %d" %
selecteditem[0])
            conn.commit()
            cursor.close()
            conn.close()
```

Function to search data

```
def SearchRecord():
    #open database
    Database()

    #checking search text is empty or not
    if SEARCH.get() == "":
        tkMessageBox.showinfo("Warning", "Please enter first name in required search field !!")
    else:
        #clearing current display data
        tree.delete(*tree.get_children())
        #select query with where clause
        cursor=conn.execute("SELECT * FROM REGISTRATION WHERE FNAME LIKE ?", ('%' +
str(SEARCH.get()) + '%',))
        #fetch all matching records

        fetch = cursor.fetchall()
        #loop for displaying all records into GUI
        for data in fetch:
            tree.insert('', 'end', values=(data))
        cursor.close()
        conn.close()
```

Defining function to access data from SQLite database

```
def DisplayData():
    #open database
    Database()
    #clear current data
    tree.delete(*tree.get_children())
    #select query
    cursor=conn.execute("SELECT * FROM REGISTRATION")
    #fetch all data from database
    fetch = cursor.fetchall()
    #loop for displaying all data in GUI
    for data in fetch:
        tree.insert('', 'end', values=(data))
        tree.bind("<Double-1>",OnDoubleClick)
    cursor.close()
    conn.close()
```

DoubleClick function

```
def OnDoubleClick(self):
    #getting focused item from treeview
    curItem = tree.focus()
    contents = (tree.item(curItem))
    selecteditem = contents['values']
    #set values in the fields
    fname.set(selecteditem[1])
        lname.set(selecteditem[2])
       gender.set(selecteditem[3])
      address.set(selecteditem[4])
       contact.set(selecteditem[5])
  btn_submit['state'] = 'disabled'
```

Chapter 13

13. Testing Approach

Software testing is the act of examining the artifacts and the behavior of the software under test by validation and verification. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but not necessarily limited to:

analyzing the product requirements for completeness and correctness in various contexts like industry perspective, business perspective, feasibility and viability of implementation, usability, performance, security, infrastructure considerations, etc.

reviewing the product architecture and the overall design of the product

working with product developers on improvement in coding techniques, design patterns, tests that can be written as part of code based on various techniques like boundary conditions, etc.

executing a program or application with the intent of examining behavior

reviewing the deployment infrastructure and associated scripts & automation

take part in production activities by using monitoring & observability techniques

Software testing can provide objective, independent information about the quality of software and risk of its failure to users or sponsors.

To build up our project we use software testing process for executing a program with the intent of findingerrors that is uncovering errors in a program makes it a feasible task and also trying to find the error in a program as it is destructive process.

Although software testing can determine the correctness of software under the assumption of some specific hypotheses (see the hierarchy of testing difficulty below), testing cannot identify all the failures within the software. Instead, it furnishes a criticism

or comparison that compares the state and behavior of the product against test oracles — principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions, but only that it does not function properly under specific conditions.[4] The scope of software testing may include the examination of code as well as the execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing assists in making this assessment.

TESTING APPROACH

Static, dynamic, and passive testing

There are many approaches available in software testing. Reviews, walkthroughs, or inspections are referred to as static testing, whereas executing programmed code with a given set of test cases is referred to as dynamic testing.

Static testing is often implicit, like proofreading, plus when programming tools/text editors check source code structure or compilers (pre-compilers) check syntax and data flow as static program analysis. Dynamic testing takes place when the program itself is run. Dynamic testing may begin before the program is 100% complete in order to test particular sections of code and are applied to discrete functions or modules.Typical techniques for these are either using stubs/drivers or execution from a debugger environment.

Static testing involves verification, whereas dynamic testing also involves validation.

Passive testing means verifying the system behavior without any interaction with the software product. Contrary to active testing, testers do not provide any test data but look at system logs and traces. They mine for patterns and specific behavior in order to make some kind of decisions.This is related to offline runtime verification and log analysis.

Exploratory approach

Exploratory testing is an approach to software testing that is concisely described as simultaneous learning, test design, and test execution. Cem Kaner, who coined the term in 1984,[18]:2 defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."

The "box" approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that the tester takes when designing test cases. A hybrid approach called grey-box testing may also be applied to software testing methodology. With the concept of grey-box testing—which develops tests from specific design elements—gaining prominence, this "arbitrary distinction" between black- and white-box testing has faded somewhat.

White-box testing

Main article: White-box testing

White Box Testing Diagram

White Box Testing Diagram

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) verifies the internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box

testing, an internal perspective of the system (the source code), as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g., in-circuit testing (ICT).

While white-box testing can be applied at the unit, integration, and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

Techniques used in white-box testing include:

API testing – testing of the application using public and private APIs (application programming interfaces)
Code coverage – creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
Fault injection methods – intentionally introducing faults to gauge the efficacy of testing strategies
Mutation testing methods
Static testing methods
Code coverage tools can evaluate the completeness of a test suite that was created with any method, including black-box testing. This allows the software team to

examine parts of a system that are rarely tested and ensures that the most important function points have been tested. Code coverage as a software metric can be reported as a percentage for:

Function coverage, which reports on functions executed
Statement coverage, which reports on the number of lines executed to complete the test
Decision coverage, which reports on whether both the True and the False branch of a given test has been executed
100% statement coverage ensures that all code paths or branches (in terms of control flow) are executed at least once. This is helpful in ensuring correct functionality, but not sufficient since the same code may process different inputs correctly or incorrectly. Pseudo-tested functions and methods are those that are covered but not specified (it is possible to remove their body without breaking any test case).

Black-Box Testing

Black-box testing (also known as functional testing) treats the software as a "black box," examining functionality without any knowledge of internal implementation, without seeing the source code. The testers are only aware of what the software is supposed to do, not how it does it.[27] Black-box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, state transition tables, decision table testing, fuzz testing, model-based testing, use case testing, exploratory testing, and specification-based testing

Specification-based testing aims to test the functionality of software according to the applicable requirements.This level of testing usually requires thorough test cases to be

provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. It uses external descriptions of the software, including specifications, requirements, and designs to derive test cases. These tests can be functional or non-functional, though usually functional.

Specification-based testing may be necessary to assure correct functionality, but it is insufficient to guard against complex or high-risk situations.

One advantage of the black box technique is that no programming knowledge is required. Whatever biases the programmers may have had, the tester likely has a different set and may emphasize different areas of functionality. On the other hand, black-box testing has been said to be "like a walk in a dark labyrinth without a flashlight." Because they do not examine the source code, there are situations when a tester writes many test cases to check something that could have been tested by only one test case or leaves some parts of the program untested.

This method of test can be applied to all levels of software testing: unit, integration, system and acceptance. It typically comprises most if not all testing at higher levels, but can also dominate unit testing as well.

Component interface testing

Component interface testing is a variation of black-box testing, with the focus on the data values beyond just the related actions of a subsystem component. The practice of component interface testing can be used to check the handling of data passed between various units, or subsystem components, beyond full integration testing between those units. The data being

passed can be considered as "message packets" and the range or data types can be checked, for data generated from one unit, and tested for validity before being passed into another unit. One option for interface testing is to keep a separate log file of data items being passed, often with a timestamp logged to allow analysis of thousands of cases of data passed between units for days or weeks. Tests can include checking the handling of some extreme data values while other interface variables are passed as normal values. Unusual data values in an interface can help explain unexpected performance in the next unit.

Visual testing

The aim of visual testing is to provide developers with the ability to examine what was happening at the point of software failure by presenting the data in such a way that the developer can easily find the information she or he requires, and the information is expressed clearly.

At the core of visual testing is the idea that showing someone a problem (or a test failure), rather than just describing it, greatly increases clarity and understanding. Visual testing, therefore, requires the recording of the entire test process – capturing everything that occurs on the test system in video format. Output videos are supplemented by real-time tester input via picture-in-a-picture webcam and audio commentary from microphones.

Visual testing provides a number of advantages. The quality of communication is increased drastically because testers can show the problem (and the events leading up to it) to the developer as opposed to just describing it and the need to replicate test failures will cease to exist in many cases. The developer will have all the evidence she or he requires of a test

failure and can instead focus on the cause of the fault and how it should be fixed.

Ad hoc testing and exploratory testing are important methodologies for checking software integrity, because they require less preparation time to implement, while the important bugs can be found quickly. In ad hoc testing, where testing takes place in an improvised, impromptu way, the ability of the tester(s) to base testing off documented methods and then improvise variations of those tests can result in more rigorous examination of defect fixes. However, unless strict documentation of the procedures are maintained, one of the limits of ad hoc testing is lack of repeatability.

Further information: Graphical user interface testing

Grey-box testing

Main article: Gray box testing

Grey-box testing (American spelling: gray-box testing) involves having knowledge of internal data structures and algorithms for purposes of designing tests while executing those tests at the user, or black-box level. The tester will often have access to both "the source code and the executable binary." Grey-box testing may also include reverse engineering (using dynamic code analysis) to determine, for instance, boundary values or error messages.Manipulating input data and formatting output do not qualify as grey-box, as the input and output are clearly outside of the "black box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for the test.

By knowing the underlying concepts of how the software works, the tester makes better-informed testing choices while testing the software from outside. Typically, a grey-box tester will be permitted to set up an isolated testing environment with activities such as

seeding a database. The tester can observe the state of the product being tested after performing certain actions such as executing SQL statements against the database and then executing queries to ensure that the expected changes have been reflected. Grey-box testing implements intelligent test scenarios, based on limited information. This will particularly apply to data type handling, exception handling, and so on.

13.1. Type of Testing

Type of testing we Use in Our Project Here we just mentioned that how the testing is related to this software and in which way we have test the software? In our project we have used 4 types of testingthese are listed below –

I. **Unit testing:** Unit testing where individual program unit or object classes are tested here by using this testing we have focused on testing the functionality of methods.

II. **Module Testing :** Where this is the combination of unit is called module. Here we tested the unit program is where the module program have dependency

III. **Sub- system Testing :** The we combined some module for the preliminary system testing inour project

IV. **System Testing :** where it is the combination of two or more sub – system and then it is tested. Here we tested and entire system as per the requirements.

13.2. Use Case

A use case is a methodology used in system analysis to identify, clarify and organize

system requirements. The use case is made up of a set of possible sequences of interactions between systemsand users in a particular environment and related to a particular goal.

Every use case contains three essential elements:

I. The actor. The system user -- this can be a single person or a group of people interacting withthe process.
II. The goal. The final successful outcome that completes the process.
III. The system. The process and steps taken to reach the end goal, including the necessaryfunctional requirements and their anticipated behaviours.

The writing process includes:

I. Identifying all system users and creating a profile for each one. This includes every roleplayed by a user who interacts with the system.
II. Selecting one user and defining their goal -- or what the user hopes to accomplish byinteracting with the system. Each of these goals becomes a use case.
III. Describing the course taken for each use case through the system to reach that goal.
IV. Considering every alternate course of events and extending use cases -- or the differentcourses that can be taken to reach the goal.
V. Identifying commonalities in journeys to create common course use cases and writedescriptions of each.

VI. Repeating steps two through five for all other system users.

Benefits OF USE CASE :

A single use case can benefit developers by revealing how a system should behave while alsohelping identify any errors that could arise in the process.

Other benefits of use case development include:

I. The list of goals created in the use case writing process can be used to establish thecomplexity and cost of the system.

II. By focusing both on the user and the system, real system needs can be identified earlier in thedesign process.

III. Since use cases are written primarily in a narrative language they are easily understood bystakeholders, including customers, users and executives -- not just by developers and testers.

IV. The creation of extending use cases and the identification of exceptions to successful use casescenarios saves developers time by making it easier to define subtle system requirements.

V. By identifying system boundaries in the design scope of the use case, developers can avoidscope creep.

VI. Premature design can be avoided by focusing on what the system should do rather than howit should do it.

12.3 Test Case

A Test Case is a set of conditions or variables under which a tester will determine whether a systemunder test satisfies requirements or works correctly.

The process of developing test cases can also help find problems in the requirements or design of anapplication.

Test Case Template

A test case can have the following elements. Note, however, that a test management tool is normallyused by companies and the format is determined by the tool used.

Test Suite ID	The ID of the test suite to which this test case belongs.
Test Case ID	The ID of the test case.
Test Case Summary	The summary / objective of the test case.
Related Requirement	The ID of the requirement this test case relates/traces to.
Prerequisites	Any prerequisites or preconditions that must be fulfilledprior to executing the test.
Test Procedure	Step-by-step procedure to execute the test.

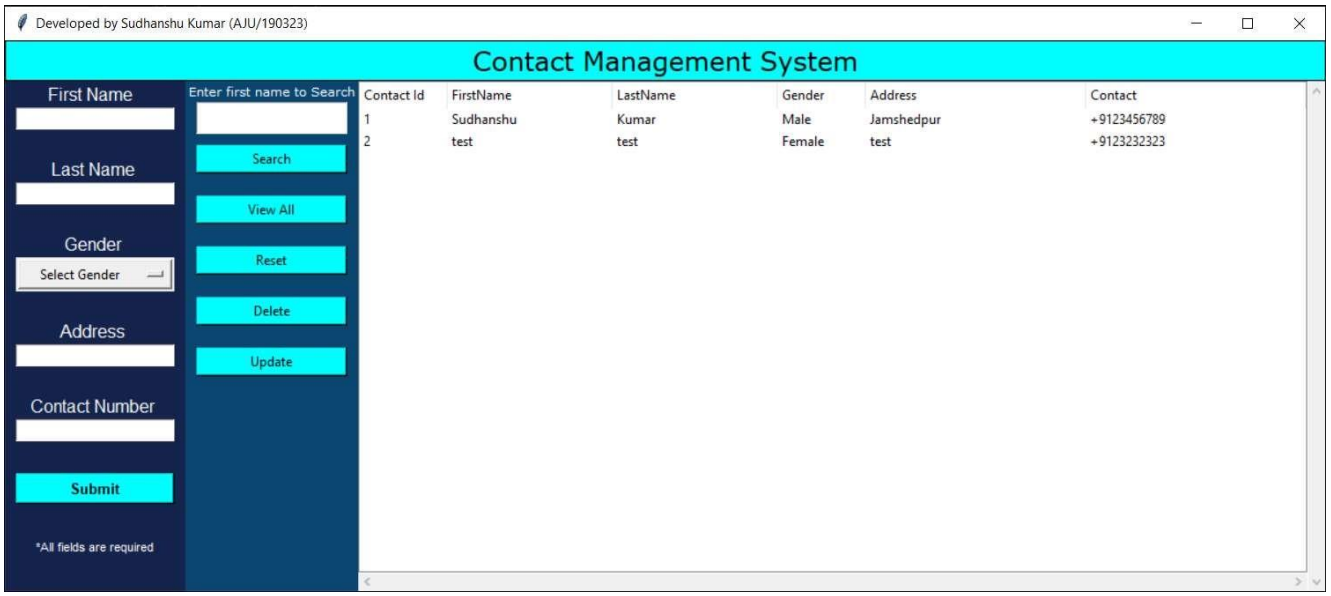
Test Data	The test data, or links to the test data, that are to be usedwhile conducting the test.
Expected Result	The expected result of the test.
Actual Result	The actual result of the test; to be filled after executingthe test.
Status	Pass or Fail. Other statuses can be „Not Executed“ if testing is not performed and „Blocked“ if testing is blocked.
Remarks	Any comments on the test case or test execution.
Created By	The name of the author of the test case.
Date of Creation	The date of creation of the test case.
Executed By	The name of the person who executed the test.
Date of Execution	The date of execution of the test.
Test Environment	The environment (Hardware/Software/Network) in which the test was executed.

Chapter 14

14. Output Screen of Contact Management System

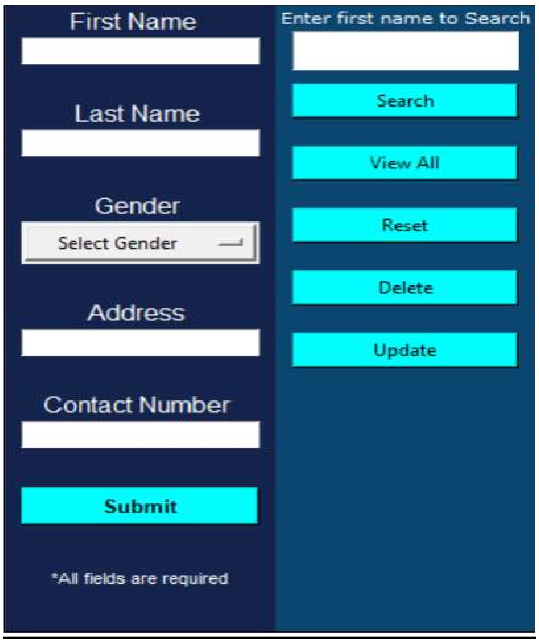
14.1 CMS Home Page:

To start a contact management application, we have to run a CMS.py file. Then the screen of the application will be opened.



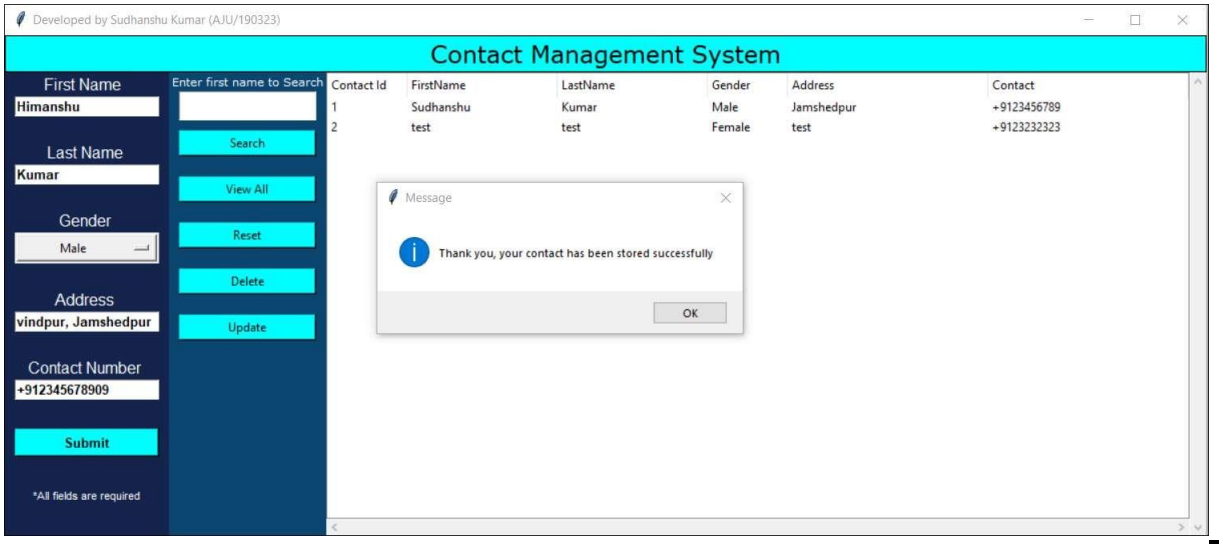
14.2 Operational Keys.

- Submit Button: To submit new contact.
- Search Button: To search any specific contact.
- View All Button: To display all records.
- Reset Button: To reset all fields value
- Delete Button: To delete the selected record.
- Update Button: To update the selected record

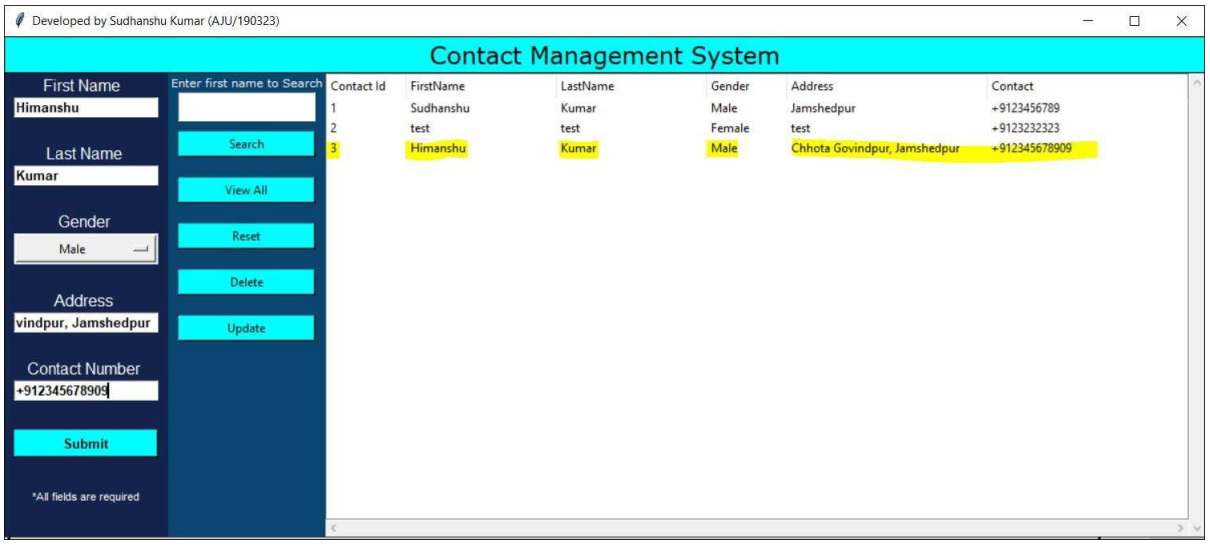


14.3 New contact entry :

User has to enter all required fields and then click on the ‘Submit’ button to save the record. System will notify that a new record has been stored.



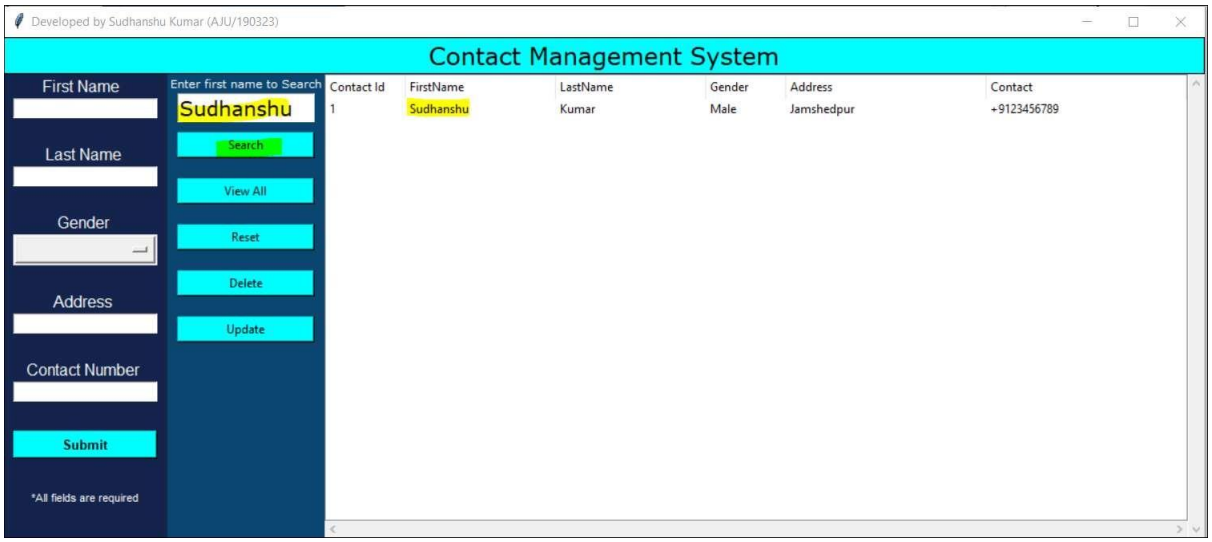
Once new record is stored in system then it will display on right side table as below:



14.4 Search Contact :

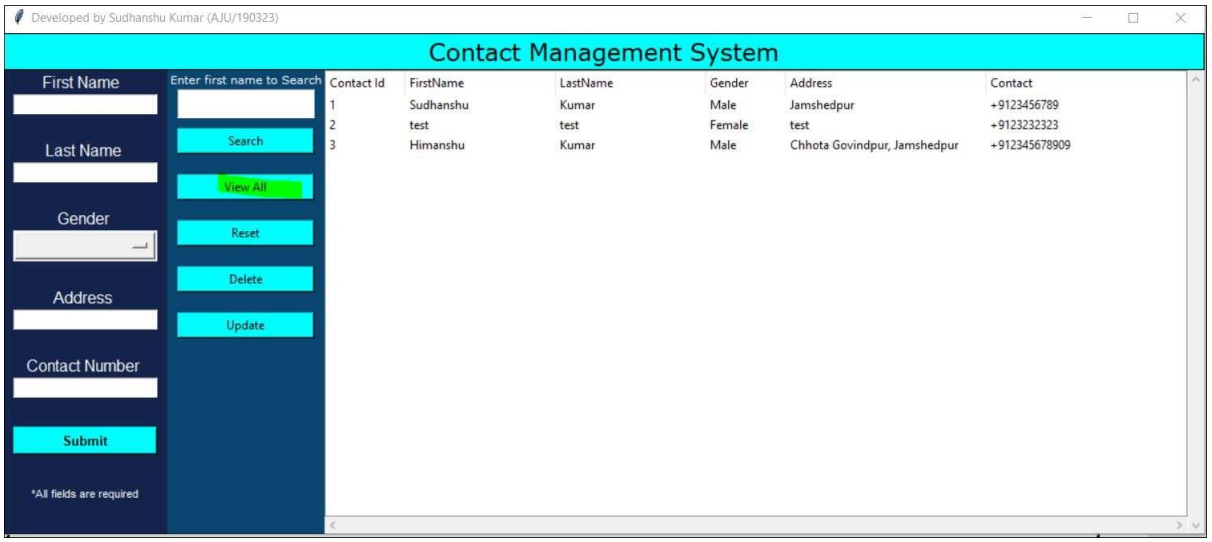
If you want to search any specific user record then you must enter user Firstname of the user is search text field, then click on ‘Search’ button.

Search results will be displayed on the right-side table for the specific user.



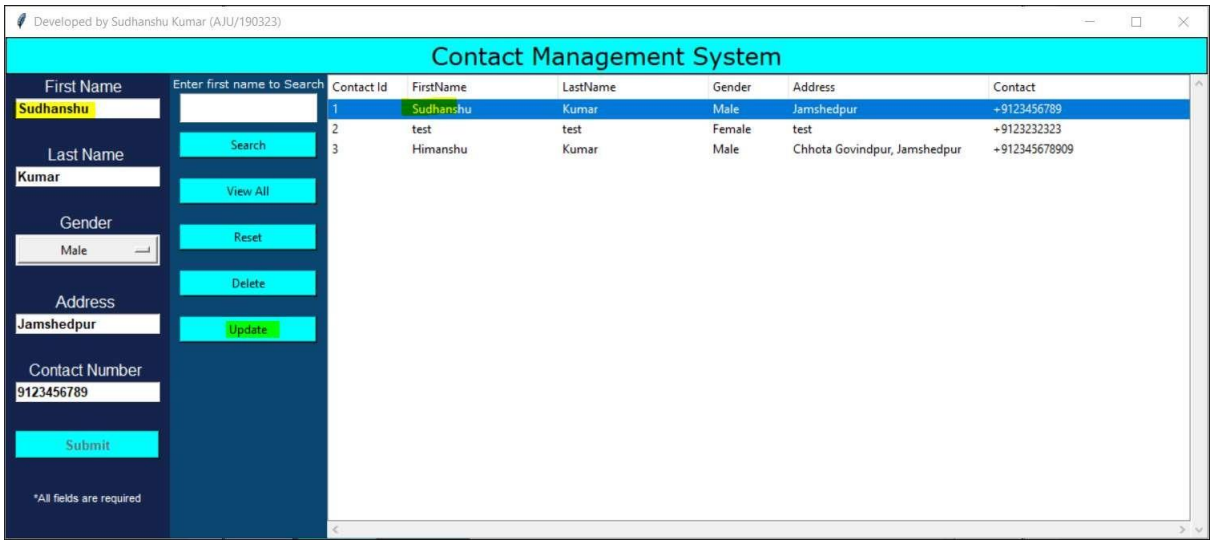
14.5 View All Contact :

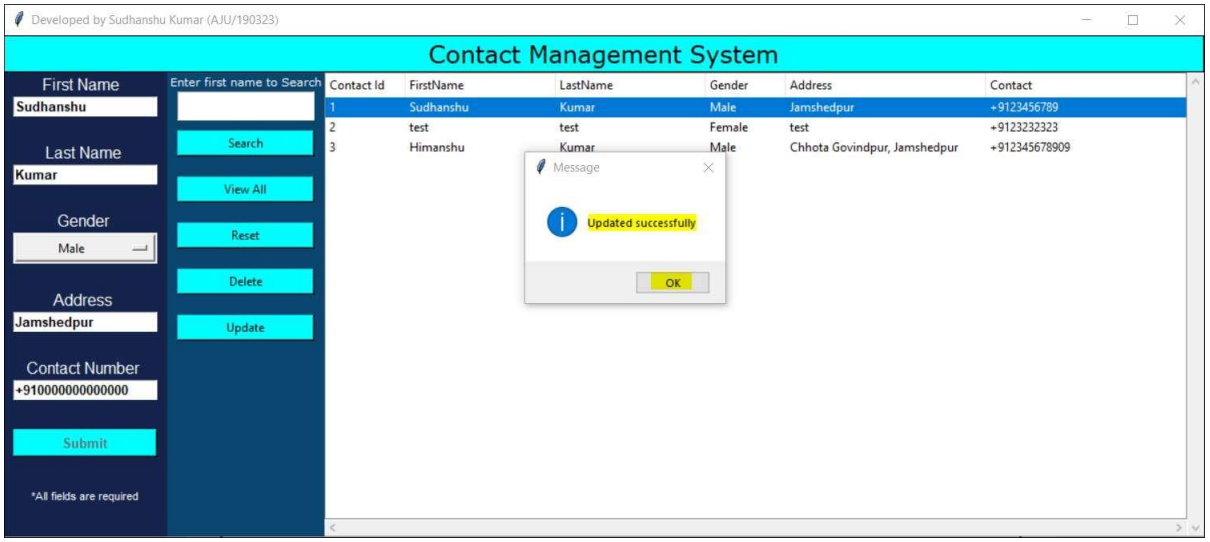
After searching any specific user recored, you have to clicked on ‘View All’ button to disply all recoreds.



14.6 Update record :

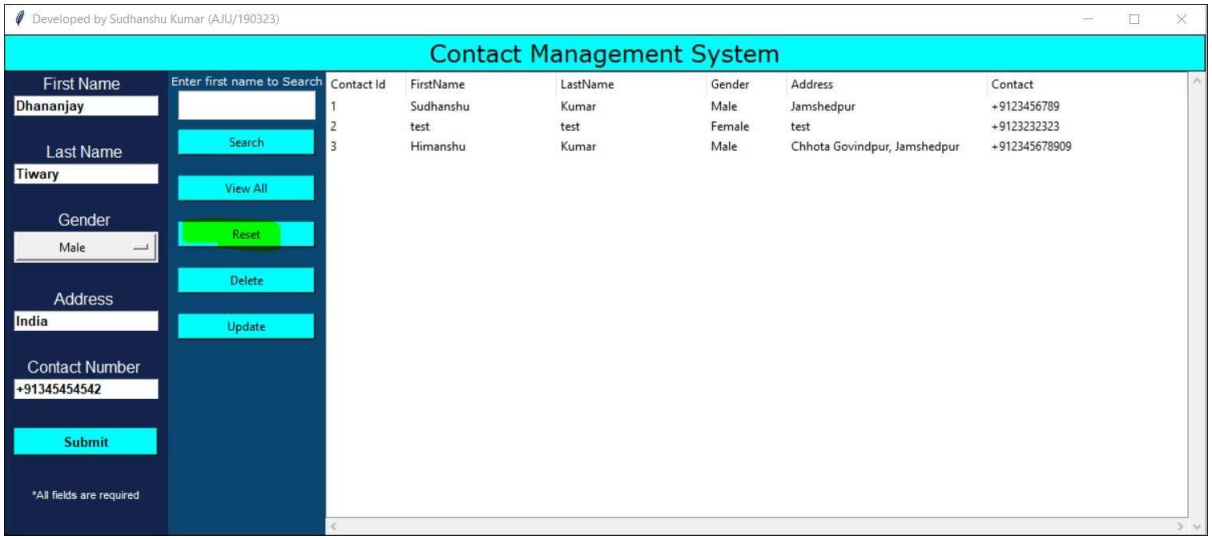
If the user wants to update any specific contact details then the user must double click on the specific record, then the selected record will be available on the left side field. And the ‘Submit’ button will be disable then the user can modify the respective field value and click on the ‘Update’ button. Then confirmation message will be given by system.





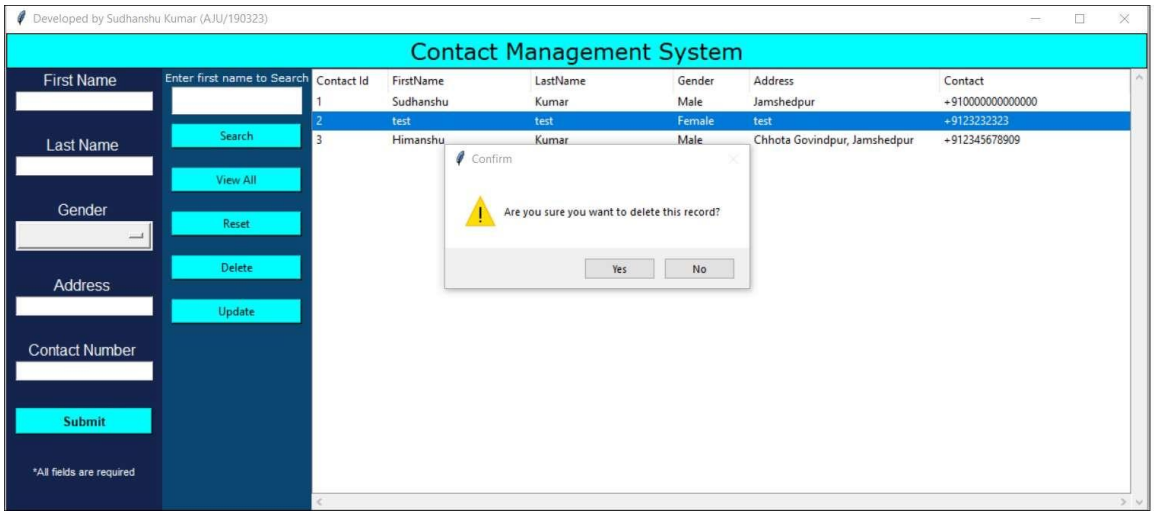
14.7 Reset All field :

If user want to clear all fields then he/she has to click on ‘Reset all’ button



14.8 Delete Contact :

If a user wants to delete any record from the contact management system, then he/she has to click on a specific record, then click on the ‘Delete’ button. System will ask for confirmation with a message, and on base of confirmation, a specific record will be deleted or not.



Conclusion

The project titled as Contact Management System (CMS) was deeply studied and analyzed to design the code and implement. It was done under the guidance of the experienced project guide. All the current requirements and possibilities have been taken care during the project time.

Contact Management System is used for daily operations in any organization to maintain or access employee related information for internal administration purposes.

Application Highlights:-

- CMS application is written in python . The project file contains a python script (index.py)
- This is simple GUI based project which is very easy to understand to use.
- Talking about the system, it contains all the required function which include adding , viewing , deleting, searching and updating contact list.
- While adding the contact the person, he/she provide first name, last name, gender, address, and contact details mainly focus on C.R.U.D.

Bibliography

For Python

https://www.w3schools.com/python/

https://www.python.org/

https://docs.python.org/3/tutorial/

https://www.tutorialspoint.com/python/index.htm

For SQLite

https://www.sqlitetutorial.net/sqlite-python/

https://docs.python.org/3/library/sqlite3.html

https://pynative.com/python-sqlite/