PROJECT REPORT

ON

MODELING AND TESTING OF GUI'S USING FINITE STATE MACHINES

(Project semester July-December 2012)

Submitted for the partial fulfillment of the degree of

Bachelor of Technology in Computer Science and Engineering

Department of CSE and IT,

Jaypee University of Information Technology

Waknaghat.

Under the guidance of	**Submitted by:**
Dr.Nitin	091295- Rahul Raheja

CERTIFICATE

This is to certify that the work titled "**Modeling And Testing Of GUI'S Using Finite State Machines**" submitted by Rahul Raheja for the partial fulfillment of the award of degree of Bachelor of Technology in Computer Science & Engineering, Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma

Signature of Supervisor ......Nitin......

Name of Supervisor ......DR. NITIN......

Designation ......Associate Professor......

Date ......28/05/2013......

Acknowledgement

“Achievement is finding out what you would be doing, what you have to do. The higher the summit, higher will be the climb.” It has been rightly said that we are build on the shoulders of others but the satisfaction that accompanies the successful completion of any task would be incomplete without the mention of the people who made it possible. I am very thankful to Dr. Nitin for his valuable suggestions and encouragement to working towards this project. I find it difficult to verbalize my deepest sense of indebt of my parents and friends for their boundless love and support, which has been a source of inspiration.

ABSTRACT

Most of the Human-Computer-Interfaces will be materialized by Graphical User Interfaces (GUI). With the growing complexity of the computer-based system, also their GUIs become more complex, accordingly making the test process more and more costly. The project introduces a holistic view of fault modelling that can be carried out as a revealing many rationalization potential while testing. Appropriate formal notions and tools enable to design and test the software systematically. Based on a basic black box test criteria, test case selection can be carried out efficiently. The elements of the approach will be narrated by a realistic example which will be used also to validate the approach. In our case the example been considered is that of an Intelligent Automated Teller Machine owning to its widespread use in today's world as well as its growing complexity to make it more user friendly. We will showcase through this example how formal tools and notions can be used efficiently to design the GUI. Furthermore, we test the same using the notion of Black Box Testing.

Table of Contents

Chapter 1

INTRODUCTION

1.1 GENERAL INTRODUCTION

There are two distinct types of construction work while developing software:

-Design , implementation, and test of the programs.

-Design, implementation, and test of the user interface(UI).

We assume UI might be constructed separately, as it requires different skilss, and different techniques than construction of common software. The design part of the development job requires familiarity with the technical equipment, i.e. programming platform and language. Testing requires both: a good understanding of user requirements and familiarity with the technical equipment. Our project is about UI testing that includes testing of the programs that materialize the UI and considering the design aspects. Testing GUIs is a difficult and challenging task for many reasons: First, the input space posses a great, potentially indefinite number of combinations of inputs and events that occur as system outputs wherein events may interact with these inputs. Second, even simple GUIs possess as enormous number of states which are also due to interact with the inputs. Last but not least, many complex dependencies may hold between different states of the GUI system, and/or between its states and inputs.

Test cases generally require the determination of meaningful test inputs and expected system outputs for these inputs. Accordingly, to generate test cases for a GUI, one has to identify the test objects and test objectives. Robust systems also possess a good exception handling mechanism, i.e. they are responsive mot in terms of behaving properly in case of correct, legal inputs, but also by behaving good-natured in case of illegal inputs, generating constructive warnings, or tentative correction trials that help to navigate the user to move in the right direction. In order to validate such robust behaviour, one needs systematically generated erroneous inputs which would usually entail injection of undesired events into the Software Under Test (SUT). Such events would usually transduce the software under test into an illegal state, e.g. system crash, if the program does not process an appropriate exception handling mechanism.

1.2 AIMS and OBJECTIVE

The aim of this project is to develop a GUI on java platform and further test the same GUI, using finite state testing technique. My objectives for this project are:

- Designing a GUI in JAVA.
- Creating test cases
- Testing the GUI on the basis of test cases

1.3 PROGRAMMING LANGU7AGES USED:

- JSP (Java Server Pages)
- HTML (Hyper Text Markup Language)

1.4 Hardware Specification:

Operating System- Microsoft Windows XP professional 2002, SP2

Primary Memory- 512 MB RAM

Processor- Intel Pentium 4 CPU 2.4 GHz

Secondary Memory- 80 HDD

1.5 Formal Languages and Tools

1. Automata

Automata consists of states (represented in the figure by circles), and transitions (represented by arrows). As the automaton sees a symbol of input, it makes a transition (or jump) to another state, according to its transition function (which takes the current state and the recent symbol as its inputs).Automata theory is also closely related to formal language theory. An automaton is a finite representation of a formal language that may be an infinite set. Automata are often classified by the class of formal languages they are able to recognize.

Automata play a major role in theory of computation, compiler design, parsing and formal verification.

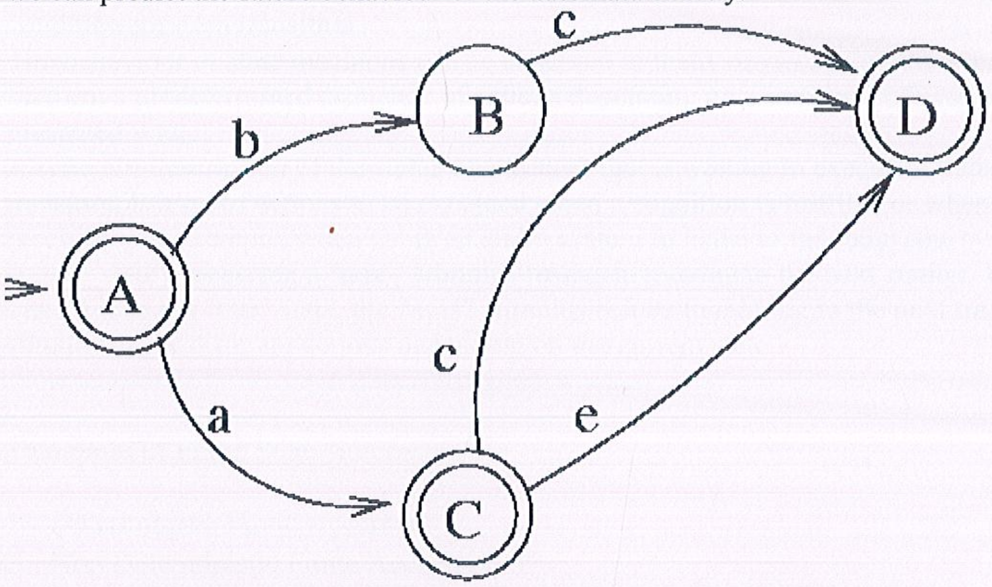
Two types of Automata:

- Deterministic Automata

- Non Deterministic Automata

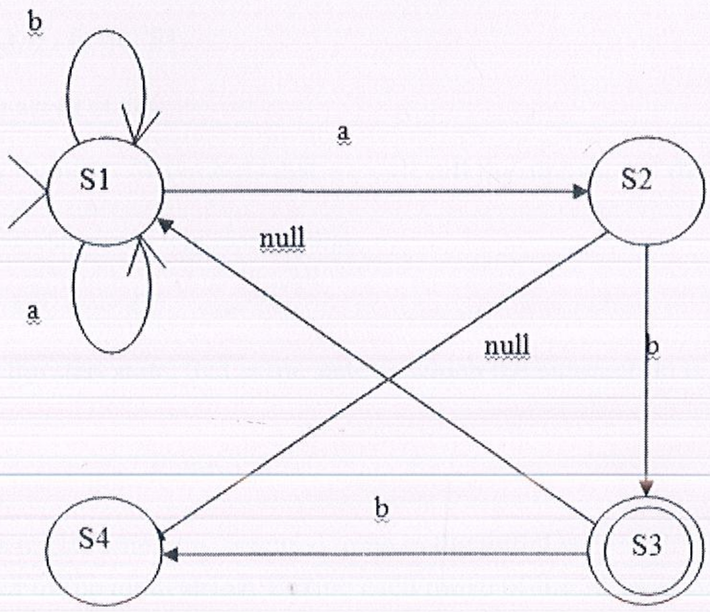
I. **Deterministic Automata**

A deterministic automata is one I which each move is uniquely determined by the current configuration. If we know the internal state, the input and the contents of temporary storage, we can predict the future behaviour of the automata exactly.



II. **Non Deterministic Automata**

In a non deterministic automata at a single point there may be several possible moves so, we can only predict a set of possible acions.



III. Finite State Automata

A finite-state machine (FSM) or finite-state automaton
is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition, this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.
The behaviour of state machines can be observed in many devices in modern society which perform a predetermined sequence of actions depending on a sequence of events they are presented with.
A state is a description of the status of a system that is waiting to execute a transition. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received. For example, when using an audio system to listen to the radio (the system is in the "radio" state), receiving a "next" stimulus results in moving to the next station. When the system is in the "CD" state, the "next" stimulus results in moving to the next track. Identical stimuli trigger different actions depending on the current state.

Two types of Finite State Automata

1. Deterministic Finite Acceptors
2. Non Deterministic Finite Acceptors

Deterministic Finite Acceptor

A deterministic Finite Acceptor of DFA is defined by the quintuple

$M=[Q, \sum, \delta, qo, F]$

Q is a set of states

$\sum$ is a finite set of symbols that we will call the alphabet of the language the automation accepts.

Δ is the transition function, that is

$\Delta : Q \times \sum \text{---->} Q$

Qo is the start state, that is the state in which the automation is when no input has been processed yet.

F is a set of states of Q, called accept states.

At the initial time, it is assumed to be in the initial state with its input mechanism on the leftmost symbol of the input string. During each move of the automation, the input mechanism advances one position to the right so, each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automation is one of the final states otherwise the string is rejected.

The transition are governed by the transition symbol δ .

Non Deterministic Finite Acceptors

Nondeterminism means a choice of moves for an automation. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that it ranges over a set of possible states.

A non deterministic finite acceptor if defined by the quintuple

M=[Q, ∑, δ, qo, F]

Q is a set of states

∑ is a finite set of symbols that we will call the alphabet of the language the automation accepts.

Δ is the transition function, that is

Δ : Q x (∑ U {λ}) ---->2 Q

There are three major differences between the definition of a deterministic finite set 2Q. Also, we allow **λ** as the second argument of **δ**. This means that an NFA can make a transition without consuming an input symbol. Although we still assume that the transition is towards right, it is possible that it is stationary on some moves.

CHAPTER 2

MODELING AND DESIGNING OF GUI

2.1 UNDERLYING IDEA

Modeling of a system requires the t of abstractions, extracting the relevant issues and information from the irrelevant ones, taking the present stage of the system development into account. While modelling a GUI, the focus is usually addressed rather to the correct behaviour of the system as desired situations, triggered be legal inputs. Describing the system behaviour in undesired exceptional situations which will be triggered by illegal inputs and other undesired events are likely to be neglected, due to time and cost pressure of the project. The precise description of such undesired situations is, however, of decisive importance for a user-oriented fault handling, because the user has not only a clean understanding how his or her system functions properly, but also which situations are not in compliance with his or her exceptions. In other words, we need a specification to describe the system behaviour both in legal and illegal situations, in accordance with the expectations of the user. Once we have such a complete description, we can then also precisely specify or hypothesis to detect undesired situations, and determine the due steps to localize and correct the faults that causes these situations.

Finite State Automata are broadly accepted for the design and specification of sequential systems for good reason. First, they have excellent recognition capabilities to effectively distinguish between correct and faulty events/situations. Moreover, efficient algorithms exist for converting FSA into equivalent regular expressions (RegEx). RegEx, on the other hand, are traditional means to generate legal and illegal situations and events systematically.

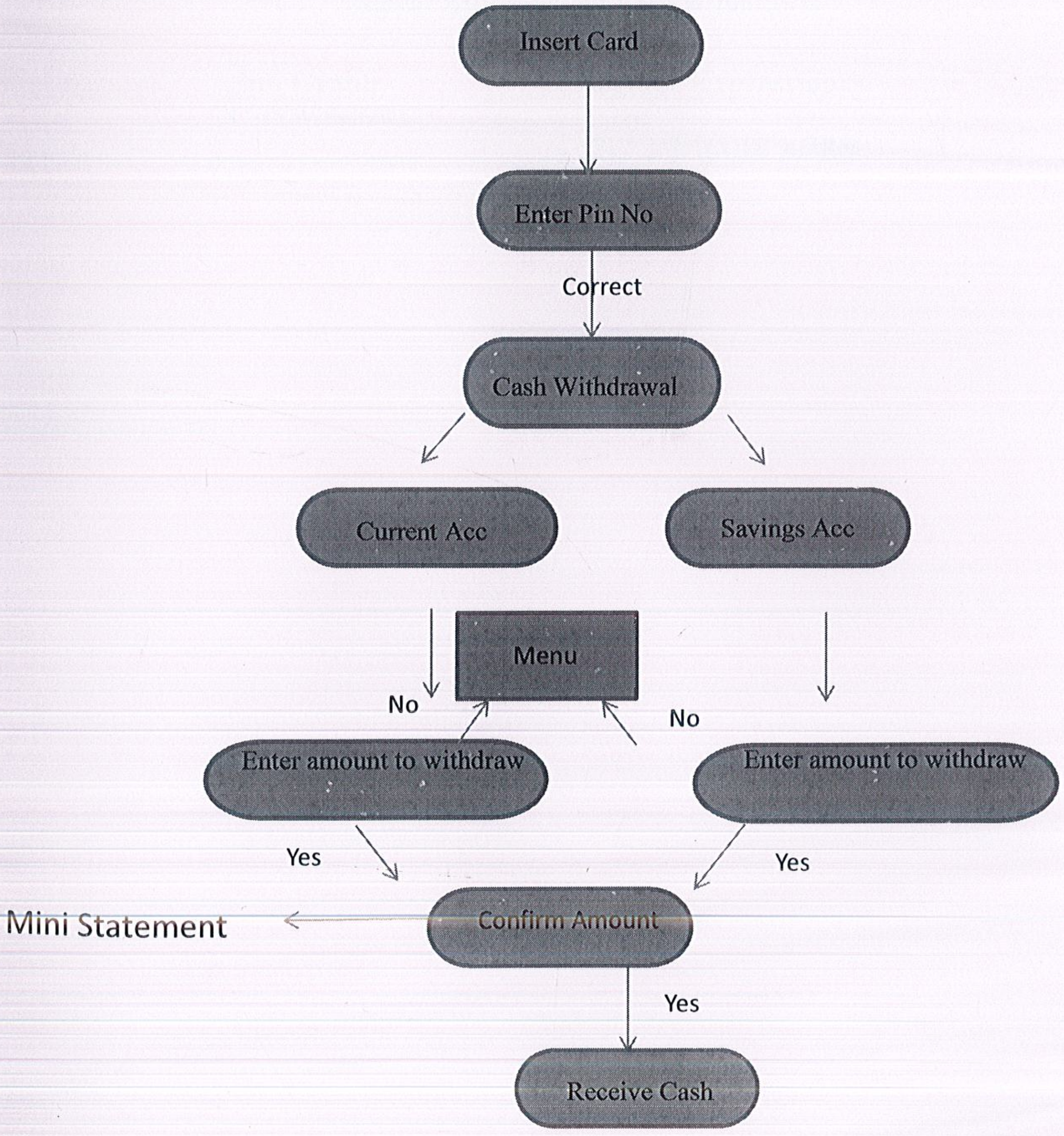
A FSM CAN BE REPRESENTED BY

- A set of inputs
- A set of outputs
- A set of states
- An output function that maps pairs and states to outputs
- A next-state function that maps pairs of inputs and states to next states

Once the FSA has been constructed, more information can be gained by the means of its state transition graph. First, we can identify now all legal sequences of user-system interactions which may be complete or incomplete, depending on the fact whether they do or do not lead to a well-defined system response that the user expects the system to carry out. Second, we can identify the entire set of the compatible, i.e. legal interaction pairs (IP) of inputs as the edges of the FSA. This is key issue of the present approach, as it will enable us to define the edge coverage notion as a test termination criterion. We start the designing process with the example of an Automatic Teller Machine automaton including all states. Its sub automatons as shown one by one as under.

2.2 Sub-Automata Diagrams

2.2.1 Cash Withdrawl



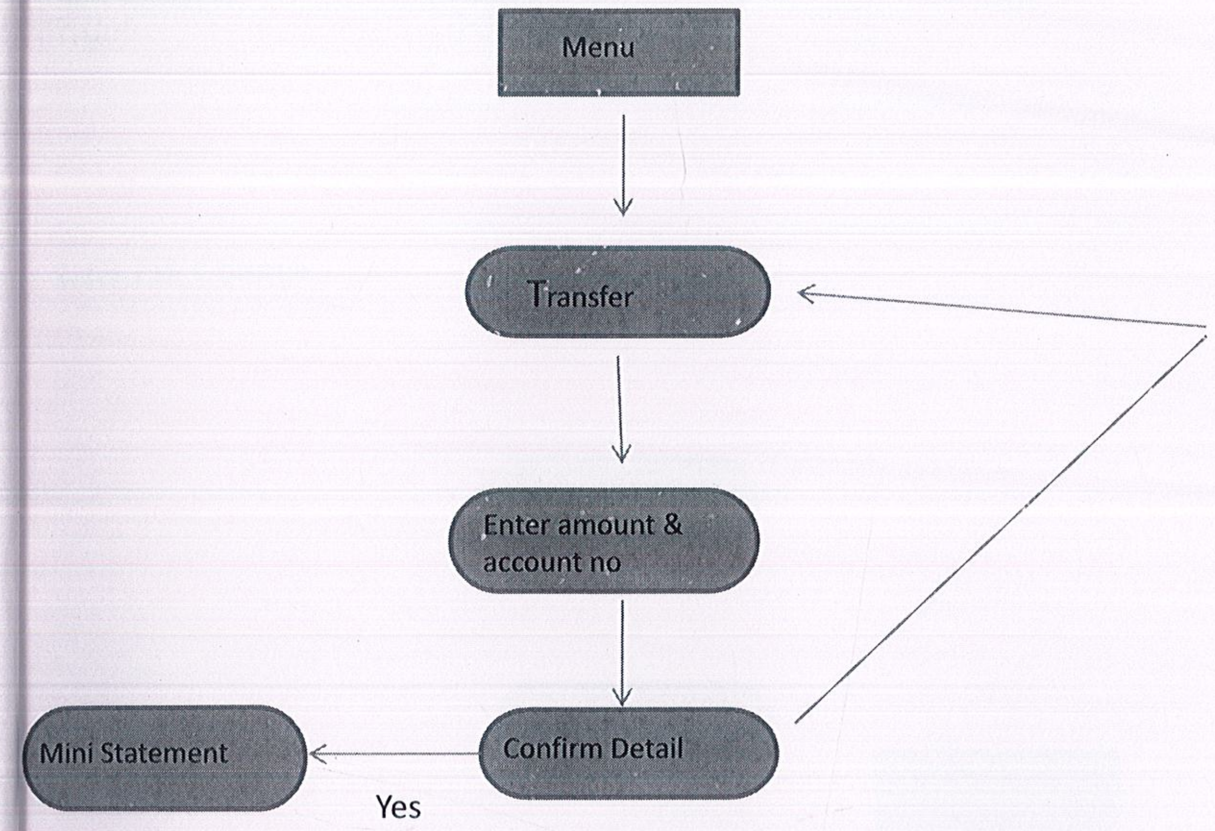
IP'S and FIP's of Cash Withdrawal

Sub Graph	Interaction Pairs	Pairs
Cash Withdrawal.	CC1, CS, C1E, EM, EC2, C2R, RM1, SE1, E1M, E1C2	C1C, SC, EC1, ME, C2E, RC2, M1R, E1S, ME1, C2E1, CC2, CR, CM, ME, ME1, MM, RR, E1E1, EE, C2C2, CC, SS, C1C1

Regular Expression for Cash Withdrawal

Sub Graph	Regular Expression
Cash Withdrawal.	$(CC1E^{+} + CSE1^{+})M+(CC1E^{+})C2RM1)$

2.2.2 Transfer of Funds



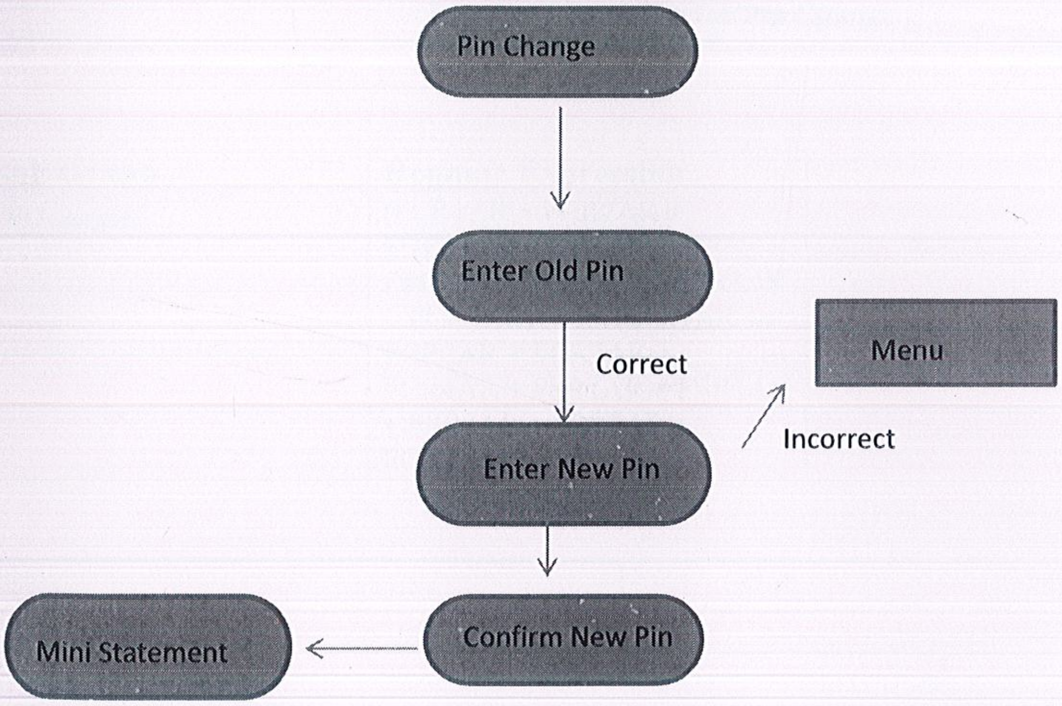
IP'S and FIP's of Transfer of Funds

Sub Graph	Interaction Pairs	Faculty Interaction Pairs
Transfer	PO, ON, NC, CM, NE	PP, OO, NN, CC, MM, OP, NO, CN, MC, EN, MP, MO, MM, ME, CP, CO, NP, PM, PN, PC, PE, OC, OM, EP, EO, EC, EM

Regular Expression for Cash Withdrawal

Sub Graph	Regular Expression
Transfer	(PO $^+$ NCM)+PO$^+$ NE)

2.2.3 Pin Change



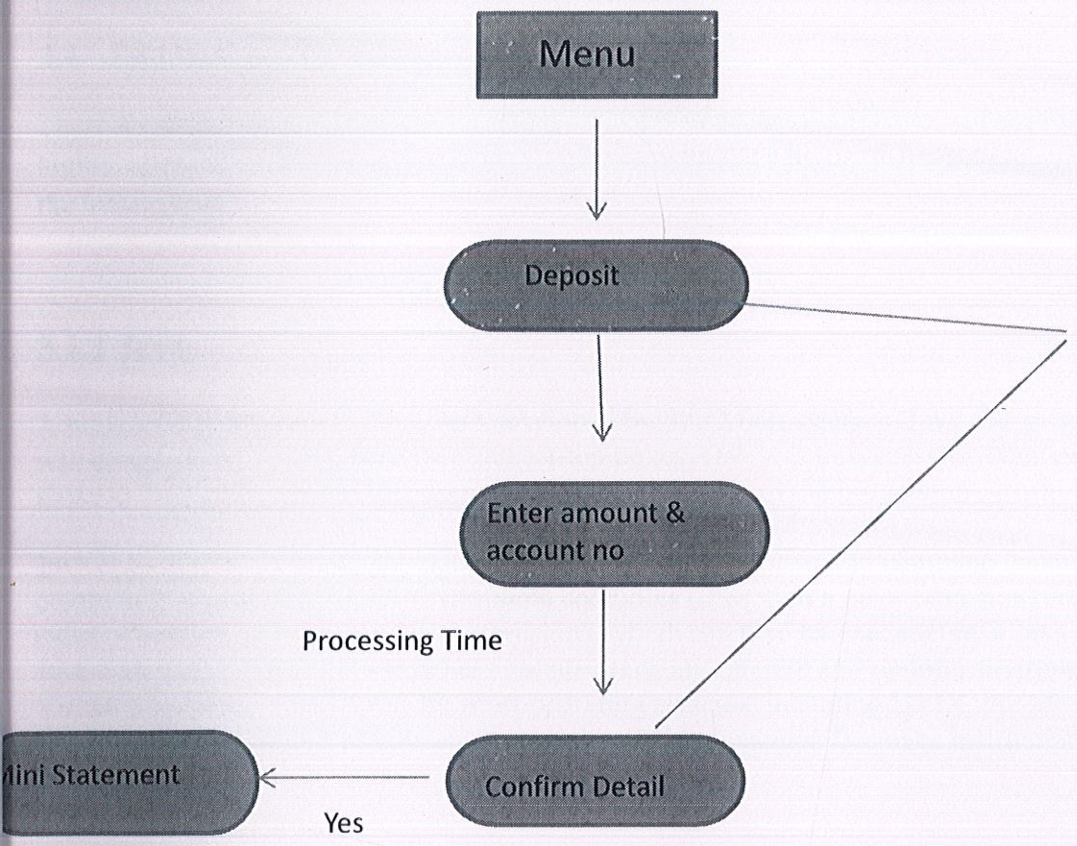
IP’S and FIP’s of Pin Change

Sub Graph	Interaction Pairs	Faculty Interaction Pairs
Pin Change	FC, FS, CR1, CR2, AR, CR3, CR4, SR1, RM, SR2, R1, SR3, SR4, R1A, R2A, R3A, R4A	FF, CC, SS, R1R1, R2R2, R3R3, R4R4, MM, AA, RR, II, CF, SF, R1C, R2C, RA, R3C, R4C, R1S, MR, R2S, IR, R3S, R4S, AR1, AR2, AR3, AR4, MF, MC, MS, MR1, MR2, MR3, MR4

Regular Expression for Pin Change

Sub Graph	Regular Expression
Pin Change	(FCR1AR + FCR2AR + FCR3AR + FCR4AR + FSR1AR + FSR2AR + FSR3AR + FCR4AR) M + (FC R1AR + FCR2AR + FCR3AR + FCR4AR + FSR1AR + FSR1AR + FSR2AR + FSR3AR + FCR4AR) I

2.2.4 Cash Deposit



IP'S and FIP's of Deposit

Sub Graph	Interaction Pairs	Faculty Interaction Pairs
Deposit	MD, DA, AC, CS	MM, DD, AA, CC, SS, DM, AD, SC, SM, SD, SA, CM, CD, AS, AM, DS, DC, MS, MC, MA)

Regular Expression for Deposit

Sub Graph	Regular Expression
Deposit	$(MDA^{+} C)S + M(DA^{+} C)^{+}$

CHAPTER 3
SOFTWARE OVERVIEW

3.1 PROGRAMMING LANGUAGES USED

There are programming languages, which are required to test the GUI's. These programming languages are highly essential for this project in order to interact with the Microsoft Access Databases via JDBC to read the information

3.1.1 Java

A high-level programming language developed by Sun Microsystems. Java was originally called OAK, and was designed for handheld devices and set-top boxes. Oak was unsuccessful so in 1995 Sun changed the name to Java and modified the language to take advantage of the burgeoning World Wide Web.

Java is an object-oriented language similar to C++, but simplified to eliminate language features that cause common programming errors. Java source code files (files with a .java extension) are compiled into a format called bytecode (files with a .class extension), which can then be executed by a Java interpreter. Compiled Java code can run on most computers because Java interpreters and runtime environments, known as Java Virtual Machines (VMs), exist for most operating systems, including UNIX, the Macintosh OS, and Windows. Bytecode can also be converted directly into machine language instructions by a just-in-time compiler (JIT).

Example

```
// OddEven.java
import javax.swing.JOptionPane;

public class OddEven {
   /**
    * "input" is the number that the user gives to the computer
    */
   private int input; // a whole number("int" means integer)

   /**
    * This is the constructor method. It gets called when an object of the OddEven type
    * is being created.
    */
   public OddEven() {
      /*
       * In most Java programs constructors can initialize objects with default values, or create
       * other objects that this object might use to perform its functions. In some Java programs, the
       * constructor may simply be an empty function if nothing needs to be initialized prior to the
       * functioning of the object. In this program's case, an empty constructor would suffice.
       * A constructor must exist; however, if the user doesn't put one in then the compiler
       * will create an empty one.
```

```
     */
  }

  /**
   * This is the main method. It gets called when this class is run through a Java interpreter.
   * @param args command line arguments (unused)
   */
  public static void main(final String[] args) {
    /*
     * This line of code creates a new instance of this class called "number" (also known as an
     * Object) and initializes it by calling the constructor. The next line of code calls
     * the "showDialog()" method, which brings up a prompt to ask you for a number
     */
    OddEven number = new OddEven();
    number.showDialog();
  }

  public void showDialog() {
    /*
     * "try" makes sure nothing goes wrong. If something does,
     * the interpreter skips to "catch" to see what it should do.
     */
    try {
      /*
       * The code below brings up a JOptionPane, which is a dialog box
       * The String returned by the "showInputDialog()" method is converted into
       * an integer, making the program treat it as a number instead of a word.
       * After that, this method calls a second method, calculate() that will
       * display either "Even" or "Odd."
       */
      this.input = Integer.parseInt(JOptionPane.showInputDialog("Please enter a number."));
      this.calculate();
    } catch (final NumberFormatException e) {
      /*
       * Getting in the catch block means that there was a problem with the format of
       * the number. Probably some letters were typed in instead of a number.
       */
      System.err.println("ERROR: Invalid input. Please type in a numerical value.");
    }
  }

  /**
   * When this gets called, it sends a message to the interpreter.
   * The interpreter usually shows it on the command prompt (For Windows users)
   * or the terminal (For *nix users).(Assuming it's open)
   */
  private void calculate() {
    if ((this.input % 2) == 0) {
      JOptionPane.showMessageDialog(null, "Even");
    } else {
      JOptionPane.showMessageDialog(null, "Odd");
    }
  }
}
```

3.2. SCRIPTING LANGUAGES

There have been two scripting languages used in this project.

-HTML

-XML

3.2.1. HTML

Hyper Text Markup Language (HTML) is the main markup language for displaying web pages and other information that can be displayed in a web browser.HTML is written in the form of HTML elements consisting of tags enclosed in angle brackets (like `<html>`), within the web page content. HTML tags most commonly come in pairs like `<h1>` and `</h1>`, although some tags, known as empty elements, are unpaired, for example `<img>`. The first tag in a pair is the start tag, the second tag is the end tag (they are also called opening tags and closing tags). In between these tags web designers can add text, tags, comments and other types of text-based content.The purpose of a web browser is to read HTML documents and compose them into visible or audible web pages. The browser does not display the HTML tags, but uses the tags to interpret the content of the page.HTML elements form the building blocks of all websites. HTML allows images and objects to be embedded and can be used to create interactive forms. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items. It can embed scripts in languages such as JavaScript which affect the behaviour of HTML webpages.

3.2.2. XML

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all gratis open standards. The design goals of XML emphasize simplicity, generality, and usability over the Internet. It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services .Many application programming interfaces (APIs) have been developed to aid software developers with processing XML data, and several schema systems exist to aid in the definition of XML-based languages.

XML documents consist entirely of characters from the Unicode repertoire. Except for a small number of specifically excluded control characters, any character defined by Unicode may appear within the content of an XML document.XML includes facilities for identifying the encoding of the Unicode characters that make up the document, and for expressing characters that, for one reason or another, cannot be used directly.

The XML specification defines an XML document as a text that is well-formed, i.e., it satisfies a list of syntax rules provided in the specification. The list is fairly lengthy; some key points are:

- It contains only properly encoded legal Unicode characters.
- None of the special syntax characters such as "<" and "&" appear except when performing their markup-delineation roles.

- The begin, end, and empty-element tags that delimit the elements are correctly nested, with none missing and none overlapping.
- The element tags are case-sensitive; the beginning and end tags must match exactly. Tag names cannot contain any of the characters !"#$%&'()*+,/;<=>?@[\]^`{|}~, nor a space character, and cannot start with -, ., or a numeric digit.
- There is a single "root" element that contains all the other elements.

3.3.3 The XML PARSER FUNCTION IN JAVA

```
Public static String Value(String file) throws Exception {

IXMLParser parser= XMLParserFactory.createDefaultXMLParser();

IXMLReader reader= StdXMLReader.fileReader(file);

Parser.setReader(reader);

XMLElement xml= (XMLElement) parser.parse();

Return xml.getContent();

}
```

3.4 JDBC

A **JDBC driver** is a software component enabling a Java application to interact with a database. JDBC drivers are analogous to ODBC drivers, ADO.NET data providers, and OLE DB providers. To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

Type 1 Driver - JDBC-ODBC bridge

The JDBC type 1 driver, also known as the **JDBC-ODBC bridge**, is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls.The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon. Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver. The use of this driver is discouraged if the alternative of a pure-Java driver is available. The other implication is that any application using a type 1 driver is non-portable given the binding between the driver and platform. This technology isn't suitable for a high-transaction environment. Type 1 drivers also don't support the complete Java command set and are limited by the functionality of the ODBC driver.Sun provides a JDBC-ODBC Bridge driver: sun.jdbc.odbc.JdbcOdbcDriver. This driver is native code and not Java, and is closed source.

Functions

- Translates a query by JDBC into a corresponding ODBC query, which is then handled by the ODBC driver.

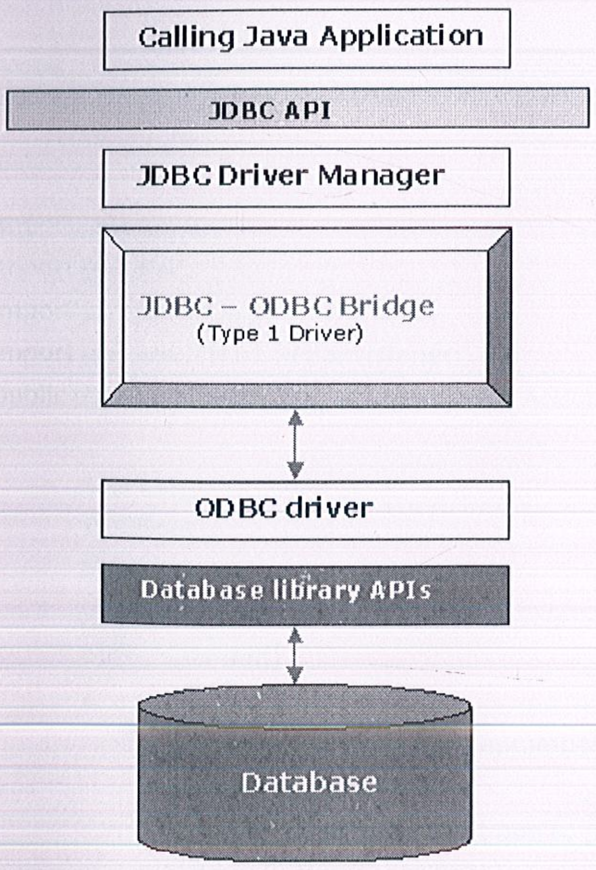
Advantages

Almost any database for which ODBC driver is installed, can be accessed.

Disadvantages

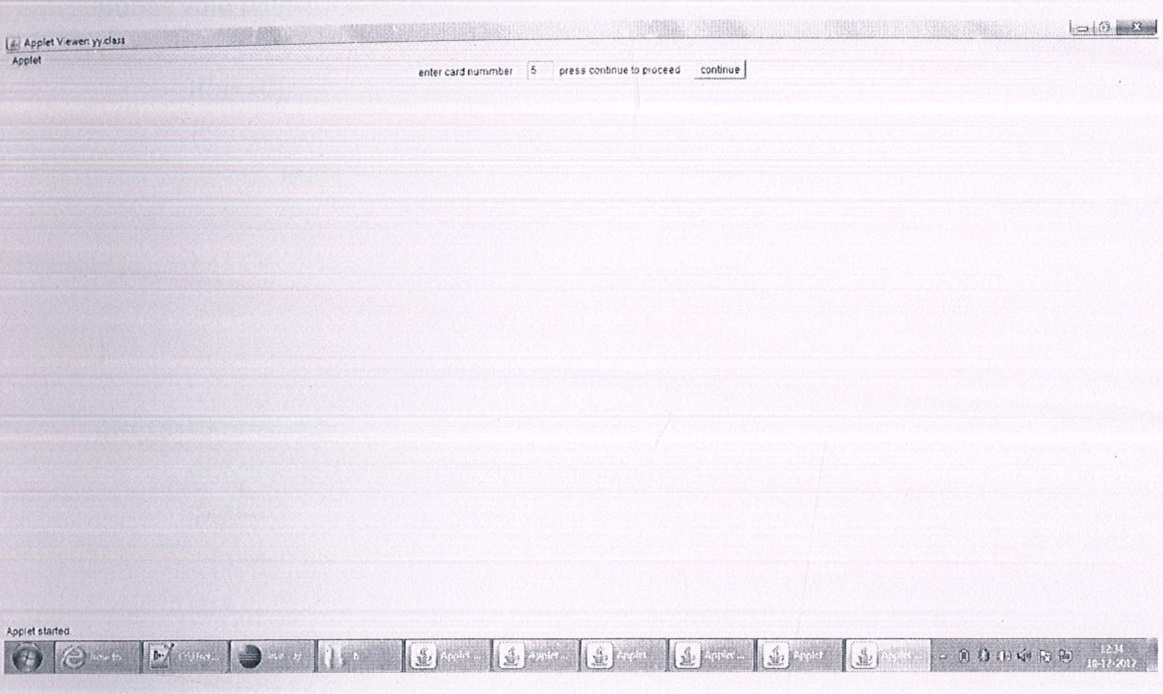
- Performance overhead since the calls have to go through the jdbc Overhead bridge to the ODBC driver, then to the native db connectivity interface (thus may be slower than other types of drivers).
- The ODBC driver needs to be installed on the client machine.
- Not suitable for applets, because the ODBC driver needs to be installed on the client.

How does it work



CHAPTER 4
DESIGNING OF GUI

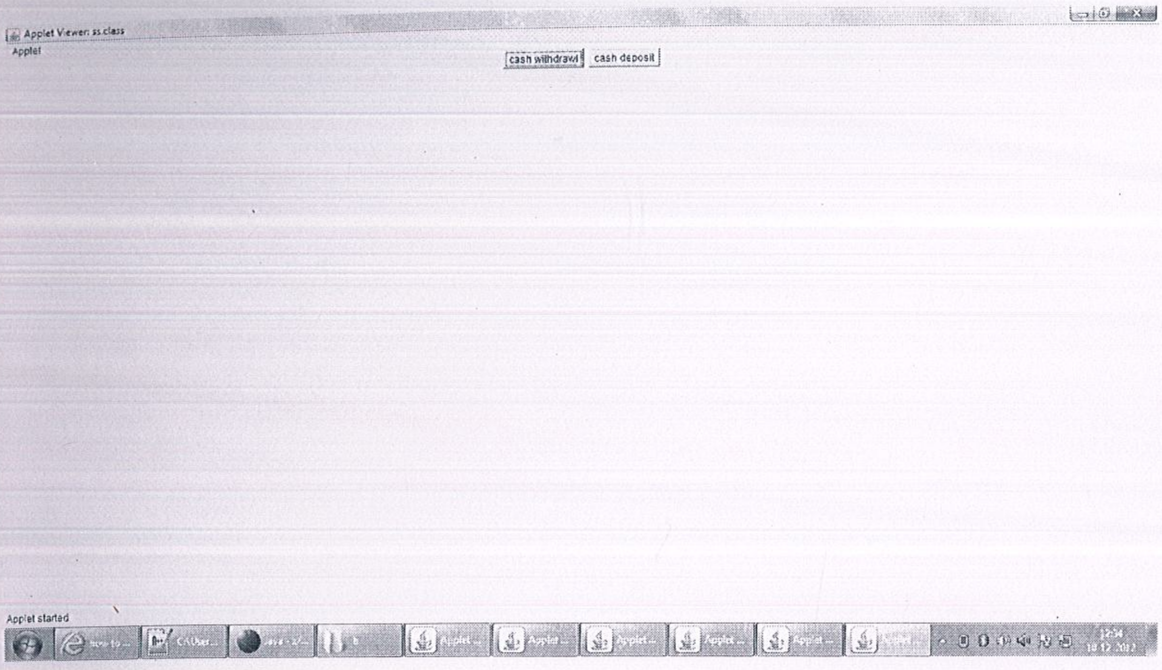
Interface 1:



```
import java.applet.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class yy extends Applet implements ActionListener
{
        Label l,m;
        TextField t;
        Button b,c,d;
        int a=1;
        public void init()
        {
                l=new Label("enter card nummber");
                t=new TextField("5");
                m=new Label("press continue to proceed");
                b=new Button("continue");
                c=new Button("cash withdrawl");
                d=new Button("cash deposit");
                add(l);
```

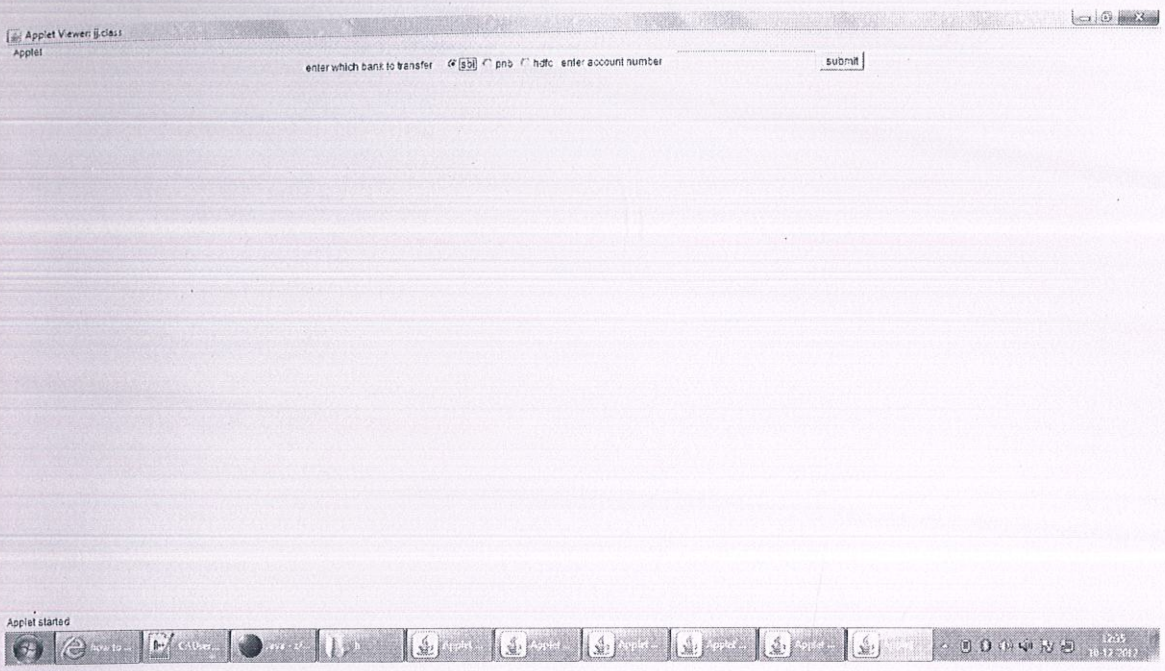
```
            add(t);

            add(m);
            add(b);
            b.addActionListener(this);


    }
    public void paint(Graphics g)
    {
            if(a==0)
            {
                    load();

            }


    }

    private void load() {
            // TODO Auto-generated method stub
            add(c);
            add(d);
            a=1;
    }
    public void actionPerformed(ActionEvent arg0)
    {
            a=0;
            repaint();
    }

}
```

Interface 2:



```
import java.applet.Applet;

import java.awt.Button;

public class ss extends Applet

{

        public void init()

        {

                add(new Button("cash withdrawl"));

                add(new Button("cash deposit"));

        }


}
```

Interface 3:



```
import java.applet.*;

import java.awt.*;

public class jj extends Applet

{

        public void init()

        {

                add(new Label("enter which bank to transfer"));

                CheckboxGroup h=new CheckboxGroup();

                add(new Checkbox("sbi",h,true));

                add(new Checkbox("pnb",h,false));

                add(new Checkbox("hdfc",h,false));

                add(new Label("enter account number"));

                TextField t=new TextField(20);

                add(t);
                add(new Button("submit"));

        }

}
```

CHAPTER 5
TESTING

5.1 Advantages of Black Box Testing

- Tester needs no knowledge of implementation

- Tester and programmer are independent of each other

- Tests are done from user's point of view

- Help's to expose any ambiguities or inconsistencies in the specification

- Test cases can be designed as soon as the specifications are complete

5.2 Cause – Effect Graphing

Cause & Effect Graphing is done through the following steps:

Step – 1: For a module, identify the input conditions (causes) and actions(effect).

Step – 2: Develop a cause-effect graph.

Step – 3: Transform cause-effect graph into a decision table.

Step – 4: Convert decision table rules to test cases. Each column of decision table represents a test case.

5.3 Test Cases

5.3.1 PIN Authenticaion

1. Causes:

C1: Enter PIN

C2: PIN is not a 4 digit number

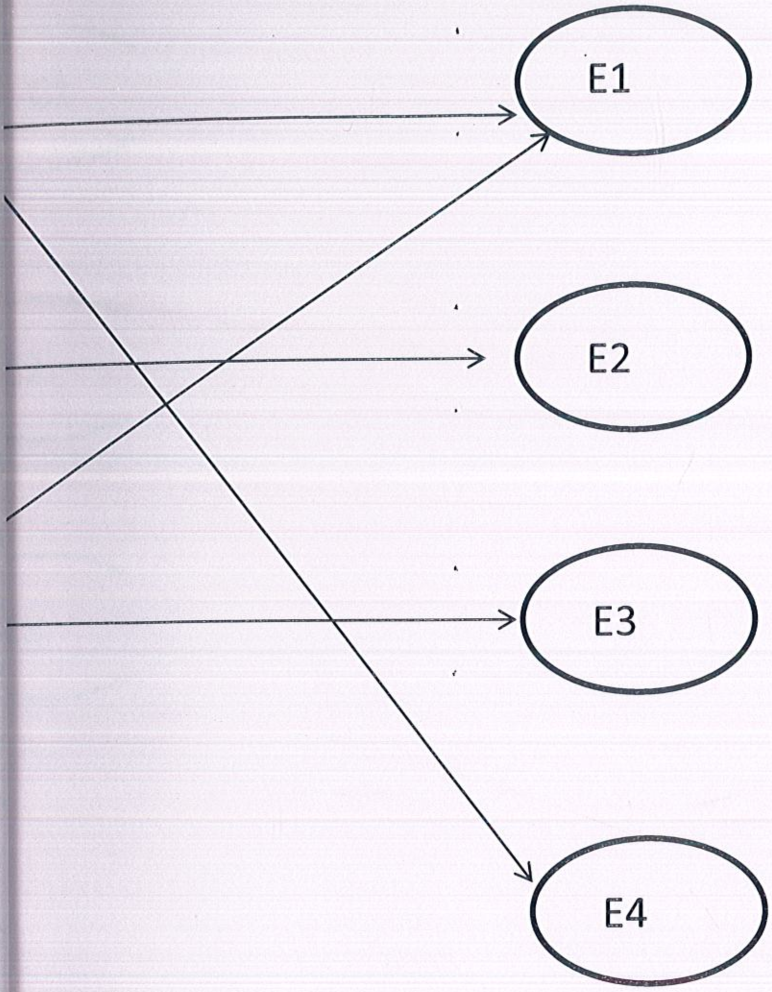
C3: PIN is not valid one

2. Effects:

E1: The message – PIN Invalid

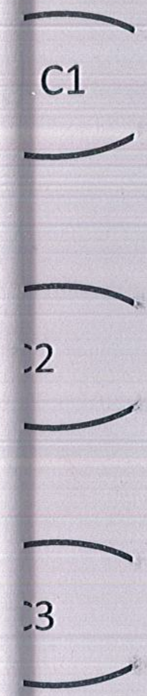
E2: The message – PIN should be of 4-digits

E3: The message – PIN Invalid, Reset System ed



Cause Effect Graph for PIN entry.

ses for the above Cause Effect Graph. The symbols followed in the design of the nessage –



wing are
are as ur

resent

bsent

:are

present

absent

	Test 1	Test 2	Test 3	Test 4
C1	I	I	I	I
C2	S	I	S	X
C3	S	S	I	X
E1	A	A	P	A
E2	A	P	A	A
E3	A	A	P	P

Table 3.1 – Test Cases for Card No. entry

5.3.2 Bank and Account No. Authentication

1. Causes

C1: Select Bank

C2: Enter account No.

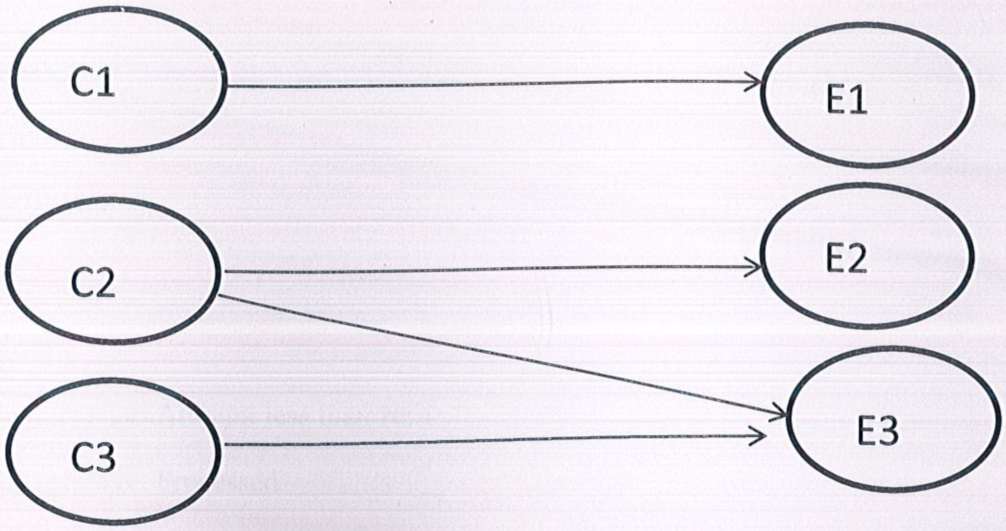
C3: Account No. is not valid one

2. Effects

E1: The message- Invalid Bank, Account No. combination

E2: The message- Account No. does not exist

E3: The message- Processed



Cause Effect Graph for Bank and Account No. Authentication

	Test 1	Test 2	Test 3	Test 4
C1	I	I	S	I
C2	I	S	I	X
C3	S	S	S	X
E1	A	A	P	P
E2	A	P	A	A
E3	P	A	A	A

Table 3.1 – Test Cases for Bank and Account No. Authentication

5.3.3 Deposit

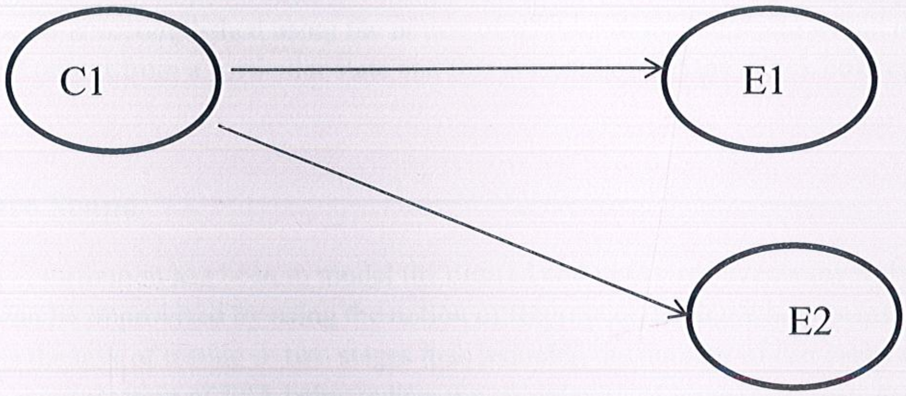
1. Causes

C1: Enter Amount

2. Effects

E1: The message- Amount less than zero

E2: The message- Processed



Cause Effect Graph for Deposit

	Test 1	Test 2
C1	**I**	**I**
E1	**A**	**P**

Table 3.3 – Test Cases for Deposit

Chapter 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

Formal notions and tools have been efficiently used to design the Graphical User Interface of an Intelligent Automated Teller Machine. The complementary view of the system has been taken into account. The software not only showcases the good system behaviour but also helps the user learn all the faults in the system behaviour so that no matter how complex the Human- Computer Interface gets the user is aware of the system behaviour.

Black Box Testing has been efficiently used to test the software developed as the software is concerned to educate the user about the good and the faulty system behaviour. Essentially, Black Box Testing deals with just the input and output which is what the user is interested in. Furthermore, the design process is accomplished using the notion of a Finite State Automata which at a basic level deals with input to and output from a particular state and this is exactly what the black box testing tests.

6.2 Future Scope

Our major contribution has been to model the desired and undesired events for software. However, the approach can be improvised by using the notion of Regression testing which helps provide a better solution and divides the task of testing in two stages thus reducing the number of test cases available at one go. Moreover, minimization of FSA helps reduce the complexity of large interfaces, thereby, decreasing the cost of testing considerably enabling the cumulating costs of testing to be in compliance with test budget.

Bibliography

References:

1. F.Belli, "Finite- State Testing and Analysis of Graphical User Interfaces", Proc. Of 12th ISSRE, IEEE Computer Society Press.
2. F.Belli, B.Guldali, "A Holistic Approach to Test- Driven Model Checking", Proc of 18th International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems

Websites:

1. www.wikipedia.org
2. www.google.com
3. www.w3schools.com

Appendix

```
import java.applet.*;

import java.awt.*;

import java.awt.event.*;


class App1 extends Applet implements ActionListener {


  /**

             *

             */

           private static final long serialVersionUID = 1L;


           Frame f = new Frame();


  Button cash = new Button("cash withdrawl");

  Button cd = new Button("cash deposit");

  App2 app2 = new App2();



  public void init() {

     System.out.println("check point 2");
```

```
    f.add(cash);
    f.add(cd);
    f.setLayout(new FlowLayout());
    f.setVisible(true);
    f.setSize(500, 500);
    f.addWindowListener(new WindowAdapter() {

        public void windowClosing(WindowEvent we) {
            System.exit(0);
        }
    });
    cash.addActionListener(this);
    cd.addActionListener(this);
}

public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == cash) {
        app2.init();
    }
    if (ae.getSource() == cd) {
        app2.init();
    }
    }
}

class App2 extends Applet implements ActionListener
{

  /**
          *
          */
          private static final long serialVersionUID = 1L;
          Frame f = new Frame();
          qq r=new qq();
 Button b1;
  public void init() {
    Label l1 = new Label("enter which bank to transfer");
    Label l2 = new Label("enter account number");
    b1 = new Button("submit");
    f.add(l1);
    CheckboxGroup h = new CheckboxGroup();
    f.add(new Checkbox("sbi", h, true));
    f.add(new Checkbox("pnb", h, false));
    f.add(new Checkbox("hdfc", h, false));
    b1.addActionListener(this);
    f.add(l2);
    TextField t = new TextField(20);
    f.add(t);
    f.add(b1);
    f.setVisible(true);
    f.setSize(300, 300);
    f.setLayout(new FlowLayout());
    f.addWindowListener(new WindowAdapter() {

        public void windowClosing(WindowEvent we) {
            System.exit(0);
        }
    });

}


        public void actionPerformed(ActionEvent e)
        {
            if(e.getSource()==b1)
            {

                r.init();
                }
                // TODO Auto-generated method stub


            }
}
class qq extends Applet implements ActionListener

{



        Frame f=new Frame();
        end e=new end();
        Label l=new Label("enter the amount");
        TextField t=new TextField(20);
        Button b2=new Button("submit");
        public  void init()
        {




            f.add(l);
            f.add(t);
            f.add(b2);
            f.setLayout(new FlowLayout());
```

```
                f.setSize(400,600);

                f.setVisible(true);

                b2.addActionListener(this);


        }

        public void actionPerformed(ActionEvent arg0) {

                // TODO Auto-generated method stub

                if(arg0.getSource()==b2)

                {

                        e.init();

                }

        }

}

class end extends Applet

{

        Frame f=new Frame();

        Label l=new Label("Processed");

        public void init()

        {

                f.setLayout(new FlowLayout());

                f.add(l);

                f.setSize(400, 400);

                f.setVisible(true);
                }
}
public class intera extends Applet implements ActionListener {


    private static final long serialVersionUID = 1L;
    Label l, m,n;
    TextField t;
    Button b, c, d;
    App1 a = new App1();
    qq r=new qq();
String mm="0";
   int q=0;
    public void init() {
        l = new Label("enter PIN");
        t = new TextField(20);
        m = new Label("press continue to proceed");
        b = new Button("continue");
        c = new Button("cash withdrawl");
        d = new Button("cash deposit");
        add(l);
        add(t);
        add(m);
        add(b);
        b.addActionListener(this);
```

```
}

   public void actionPerformed(ActionEvent arg0) {
        if (arg0.getSource() == c || arg0.getSource() == d)
     {
              r.init();


     }
        if (arg0.getSource() == b)
   {
         mm=t.getText().toString();
        q=Integer.parseInt(mm);
        if(q/1000<10)
        {
    a.init();
        }
        else
        {
              n=new Label("please enter the 4 digit number");
              add(n);
        }
```

```
            } else {
                b.setLabel("error");


            }
        }


    }
```