1. INTRODUCTION

Health care is not only everyone's right, but everyone's responsibility. Informed self-care should be the main goal of any health program or activity. Ordinary people provided with clear, simple information can prevent and treat most common health problems in their own homes—earlier, cheaper and often better than can doctors. Medical knowledge should not be the guarded secret of a select few, but should be freely shared by everyone. People with little formal education can be trusted as much as those with a lot. And they are just as smart. Basic health care should not be delivered, but encouraged.

Reason

This Project has been undertaken primarily to help those who live far from medical centers, travelling, have no one to take care of them in the time when they need it the most or are at present far from the reach of any doctor in places where there is no doctor. But even where there are doctors, people can and should take the lead in their own health care.
The Project will focus on providing a basic first aid to those in need, who are medically illiterate and feel stranded when there is no one to help.

Existing Solutions

From the research on the topic it has been found out that there is no existing solution like this one. The solutions that exist are either totally focused on the first aid for physical damages or are meant to help the pupil from the medical profession with medical terms which are out of the reach of a common man's vocabulary. Apart from this these solutions are totally internet based which require filling internet based forms and also need internet to communicate with their database and provide suitable cure. Thus making them of no use when the user is in an acute situation where there is no network access leave alone internet access.

Proposed Solution

As a result of the above listed problems the project thus proposed will aim at solving as many as the problem existing in the other software. This software will be a user friendly and simple to use application without the use of internet. It will not provide a replacement of the doctor but will give the user an initial aid to keep the user in a condition good enough till any professional medical help is available. It will also provide a basic first aid tips to the user. The main focus of the software would be on providing healing without medicines where ever there is possible.

Advantages of the Project

The main advantage of this project is its not being interdependent on the internet thus making it more useful during acute times where there is no network coverage an this feature will also contribute to it providing fast solution as compared to the once dependent on the internet. Thus making it more reliable than the other software.

The fact that this software will mostly focus on providing home based cures to diseases will help tremendously the lay man who know nothing of the medical world(medicals terms, drug names and dosages). And will also help the diseased get a more organic cure.

1.1 PROCESS APPROACH CHOSEN

INCREMENTAL MODEL

This is a combination of the linear sequential model and the iterative model. The problem is broken into increments, and each increment is tackled as a linear sequence. Further increments can either be done after the previous ones, or can overlap with the previous ones. Incremental delivery focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product.

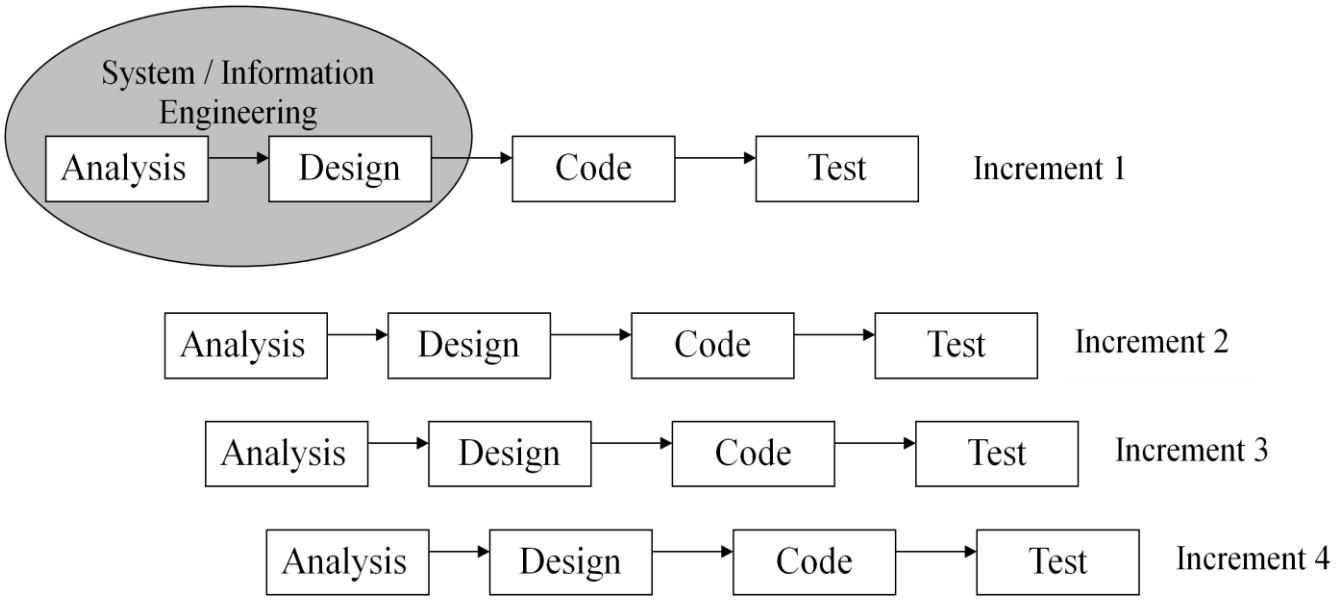

Fig. 1.1

In this project incremental model is being used, as in this case the first concern is about delivery of the core product but many supplementary features may remain undelivered. The core product is used by the customer & after evaluation; a plan for next increment is developed. The plan addresses modification in core product to meet the customer needs. Less staffing is required & also increments can be planned to manage technical risks.

1.2 TEAM STRUCTURE

CONTROLLED DECENTRALIZED (CD)

This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

We are implementing Controlled Decentralized team structure because a decentralized team generates more and better solutions than individuals. Therefore such teams have a greater probability of success when working on difficult problems. Since the CD team is centralized for problem solving, a CD team structure can be successfully applied to simple problems.

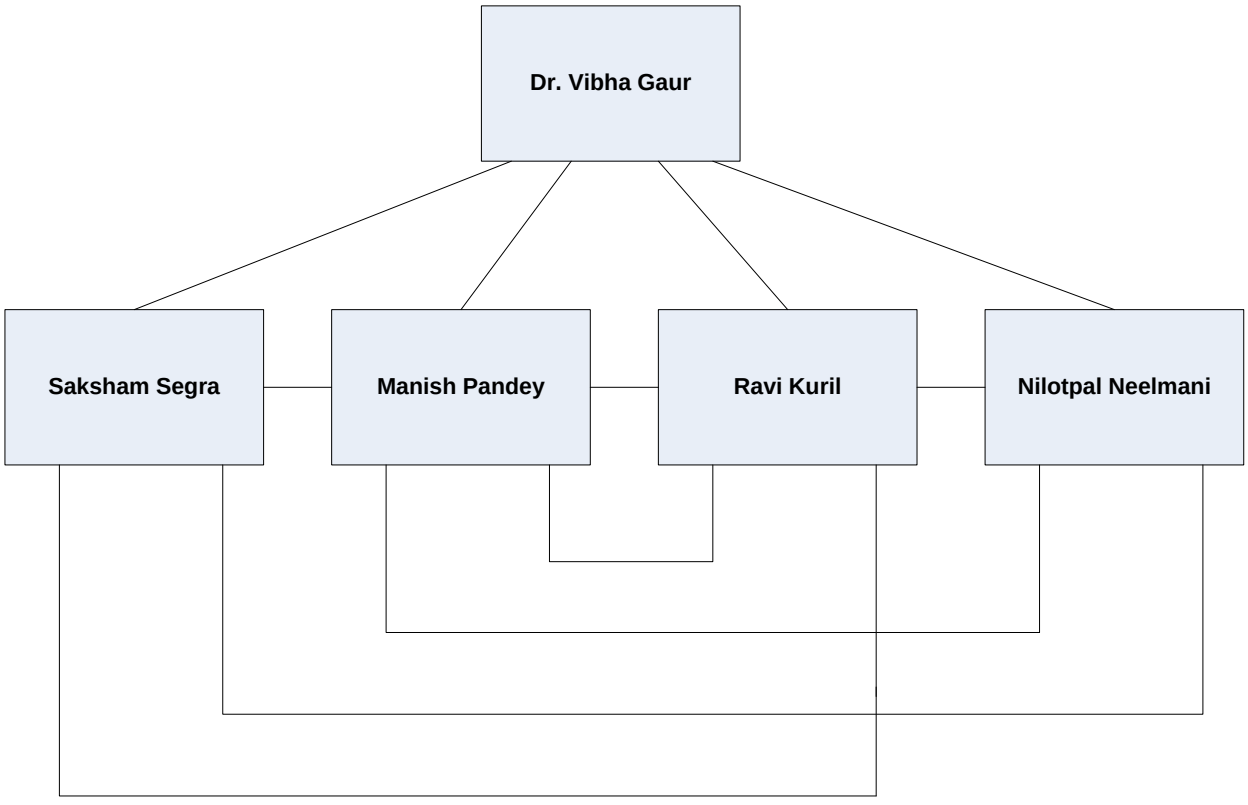

Fig. 1.2

2. REQUIREMENT ANALYSIS AND SPECIFICATION

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design. Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet. Requirements analysis allows the software engineer (sometimes called *analyst* in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software. Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs. Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

2.1 SOFTWARE SCOPE

Software scope describes the data and control to be processed, function, performance, constraints, interfaces, and reliability. Communication with the customer leads to a definition of data and control that are processed, the functions that must be implemented, the performance and constraints that bound the system, and related information.

Functions defined in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

The main aim of this project is to develop an extensive, real-time medical tool which will be very much beneficial for the common people. When the person is unwell, he can login, provide either symptoms or directly the disease (if he knows).The details will be matched by the database, disease is detected & the initial aid is provided.

2.1.1 Data and Control:

The user first opens the application and enters its details. Then a screen comes asking the user to select the options between finding the disease using symptoms or directly by entering the disease. Then the system finds the disease related to those symptoms and displays its cure and precautions.

2.1.2 Functions:

- Accept user details.
- Give option of mode to finding the disease.
- Give the disease and its cure and precautions.

2.1.4 Interfaces (Hardware and Software requirements):

Hardware interfaces*:*
The project has been made on the following systems
- Compaq C-621 laptop

Running of the project
- It can run on all java based mobiles..
- Minimum memory space is 5 Mb.

Software Interfaces*:*
- J2ME used as main programming language.
- Java (version "1.6.0_30") is used as support for Netbeans.
- Netbeans is used as the IDE.

2.1.5 Security and Reliability:

It is the probability that the service will perform its intended function during a specified period of time under stated conditions also known as *mean failure time*. Our service will perform at all times with the availability individual's mobiles battery conditions. This project is quite reliable and secure as there is no need of internet on the mobile phone to use it.

2.2 ENTITY RELATIONSHIP DIAGRAM (ERD)

The Entity Relation Diagram (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.

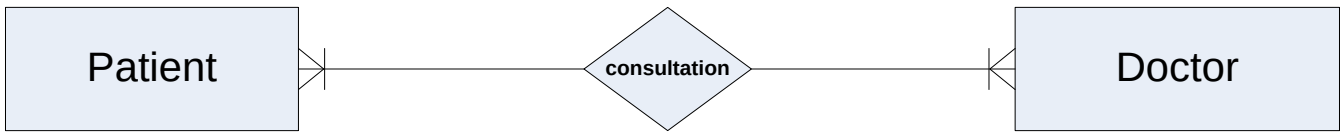

Fig. 2.1

2.3 DATA FLOW DIAGRAM

The Data Flow Diagram (DFD) serves two purposes:
(1) To provide an indication of how data are transformed as they move through the system and
(2) To depict the functions (and sub functions) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function.

0 LEVEL DFD

A level 0 DFD, also called a *fundamental system model* or a *context model*, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. In 0 level DFD the basic functionality of the software are showed. Here the circle depicts the core process of the software along with the other entities.

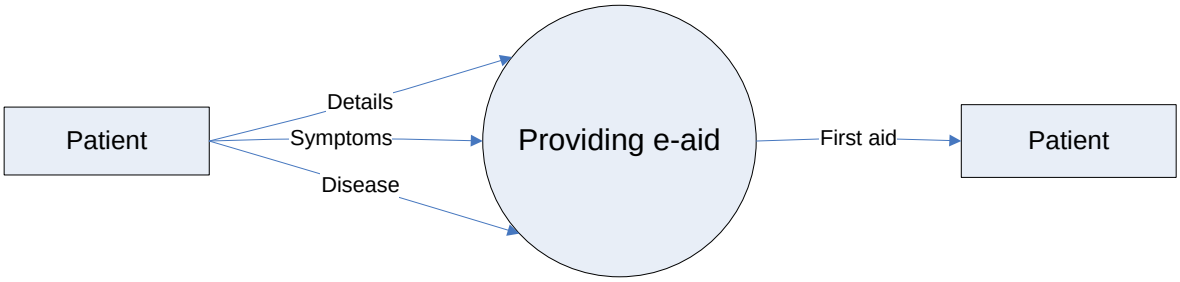

Fig. 2.2

In this DFD the rectangle is representing the entity that is using the software i.e. the Patient in this case. It gives,, mandatory details to the software.

LEVEL 1 DFD

The level 1 DFD is the extended form of the 0 level DFD. Each of the processes represented at level 1 is a sub function of the overall system depicted in the context model.

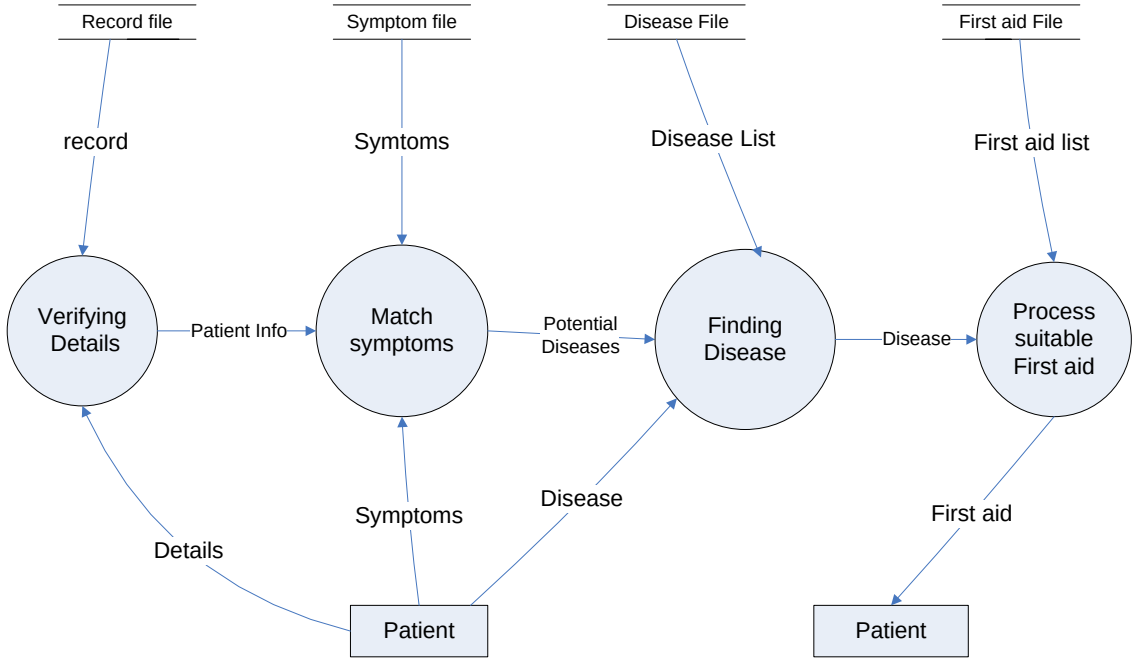

Fig. 2.3

2.4. DATA DICTIONARY

Data dictionary is a centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format.

Details = Gender + Age + Name
Gender = Male|Female
Name = First Name + Middle Name + Last Name
Age = Digit + Digit
History = Symptoms + Disease + Initial Aid + Date + Time
Date = Day + Month + Year
Day = (0|1|2|3) + Digit
Month = (0 + Digit) | (1 + (0|1|2))
Year = (1|2) + Digit + Digit + Digit
Date of Birth = Date
Record = *List of patients information*

2.5. PSPEC

A *process specification* (PSPEC) can be used to specify the processing detail simplified by a bubble within a DFD. The process specification describes the input to a function, the algorithm that is applied to transform the input, and the output that is produced.

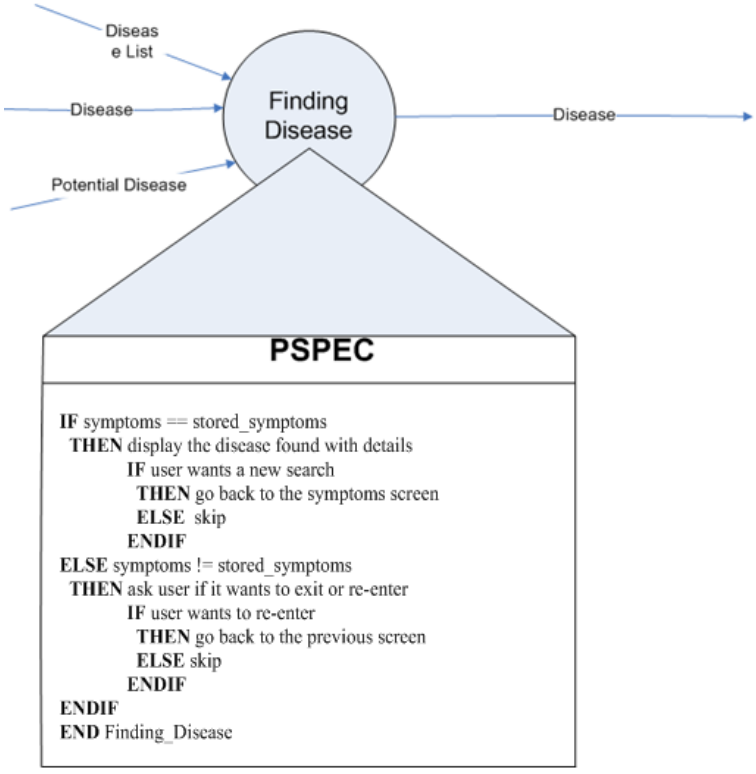

Fig. 2.4

2.6 Use case

In software and systems engineering, a use case is a list of steps, typically defining interactions between a role and a system, to achieve a goal.

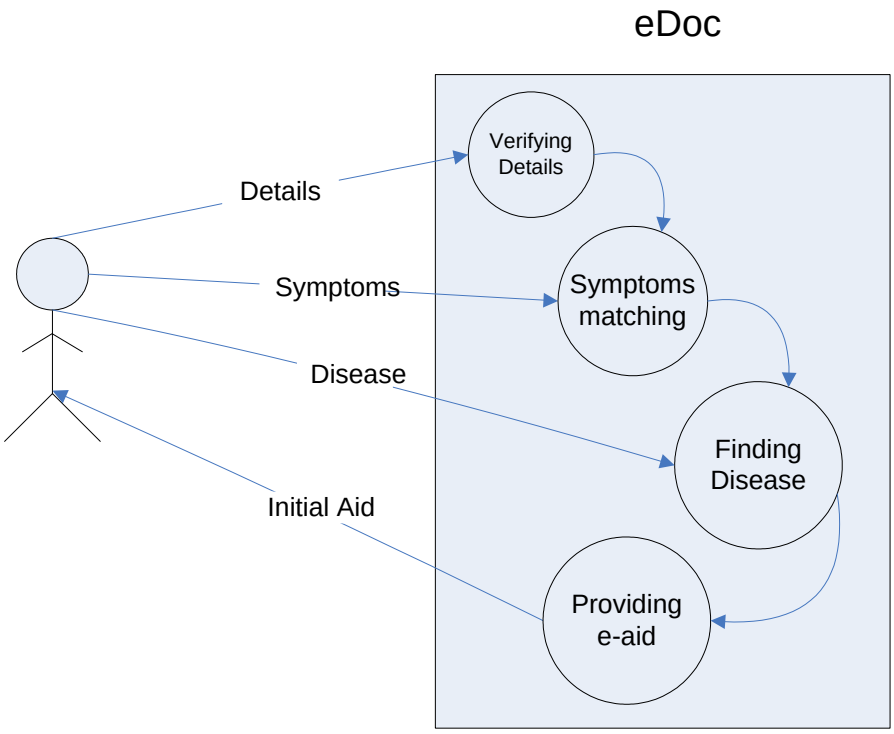

Fig. 2.5

3. DESIGN

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Software requirements, manifested by the data, functional, and behavioral models, feed the design task.

The design task produces:

1. Data design
2. Architectural design
3. Interface design
4. Component design

3.1 Data Design

The Data Design is an integrated part of design concepts and principles & appears at the basic foundation level of the design triangle.

The Data Design transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.

The Data Design model basically consists of inter-related pieces of information: the data object and the attributes that describes the data object.

The attributes of the data object 'patient' in e-Doc are explained in the following table:

ATTRIBUTE TABLE

Attributes	Length	Data Structure
First Name	15 characters	String
Middle Name	15 characters	String
Last Name	15 characters	String
Date of Birth	8 digits	Integer
Weight(approx.)	3 digits	Integer
Sex	1 character	Character
Blood Group	3 alphanumeric characters	String
Contact No.	11 digits	Integer

E-mail ID	30 alphanumeric characters	String
Patient ID	16 alphanumeric characters	String
Password	16 alphanumeric characters	String
Address	100 alphanumeric characters	String

Fig. 3.1

3.2 Architectural Design

The Architectural Design defines the relationship between major structural elements of the software, the 'design patterns' that can be used to achieve the requirements that have been designed for the system & the constraints that affect the way in which architectural design patterns can be applied.

Steps of Architectural Design:

1. Review the Fundamental System Model.

The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the System Specification and the Software Requirements Specification. Both documents describe information flow and structure at the software interface.

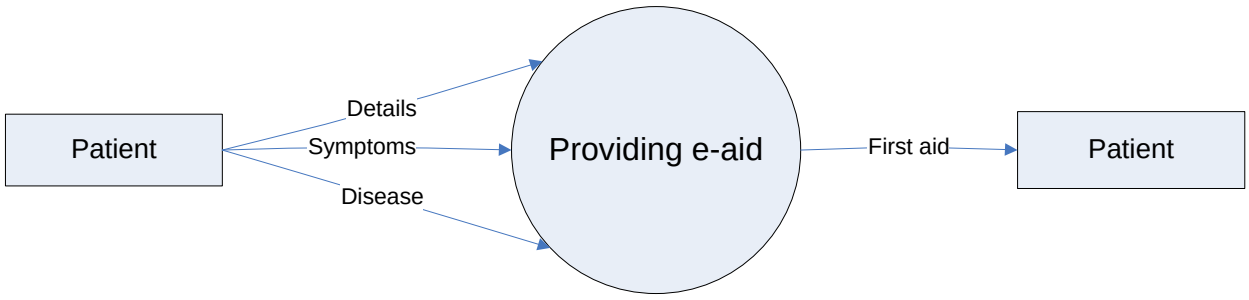

Fig. 3.2

2. Review & refine data flow diagrams for the software.

Information obtained from analysis models contained in the Software Requirements Specification is refined to produce greater detail. At level 3, each transform in the data flow diagram exhibits relatively high cohesion. That is, the process implied by a transform performs a single, distinct function that can be implemented as a module in the e-Doc software. Therefore, the DFD contains sufficient detail for a "first cut" at the design of architecture.

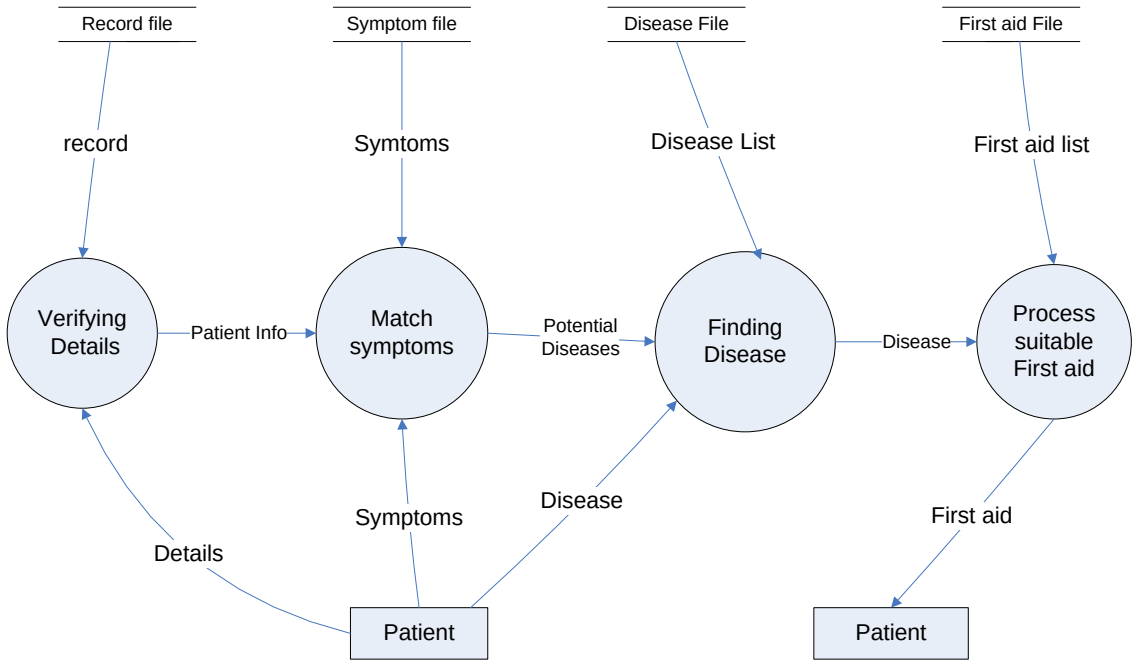

Fig. 3.3

3. Determine whether the DFD has transform or transaction flow characteristics.

In general, information flow within a system can always be represented as transform. However, when an obvious transaction characteristic is encountered, a different design mapping is recommended. In addition, local regions of transform or transaction flow are isolated. These sub flows can be used to refine program architecture derived from a global characteristic described.

Evaluating the DFD, we see data entering the software along one incoming path and exiting along three outgoing paths. No distinct transaction center is implied. Therefore, an overall transform characteristic will be assumed for information flow.

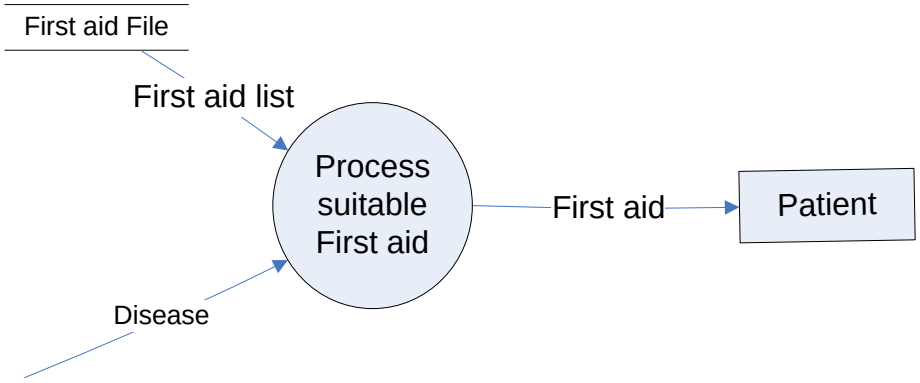

Fig. 3.4

4. Isolate the transform centre by specifying incoming & outgoing flow boundaries.

In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. Alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom. The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

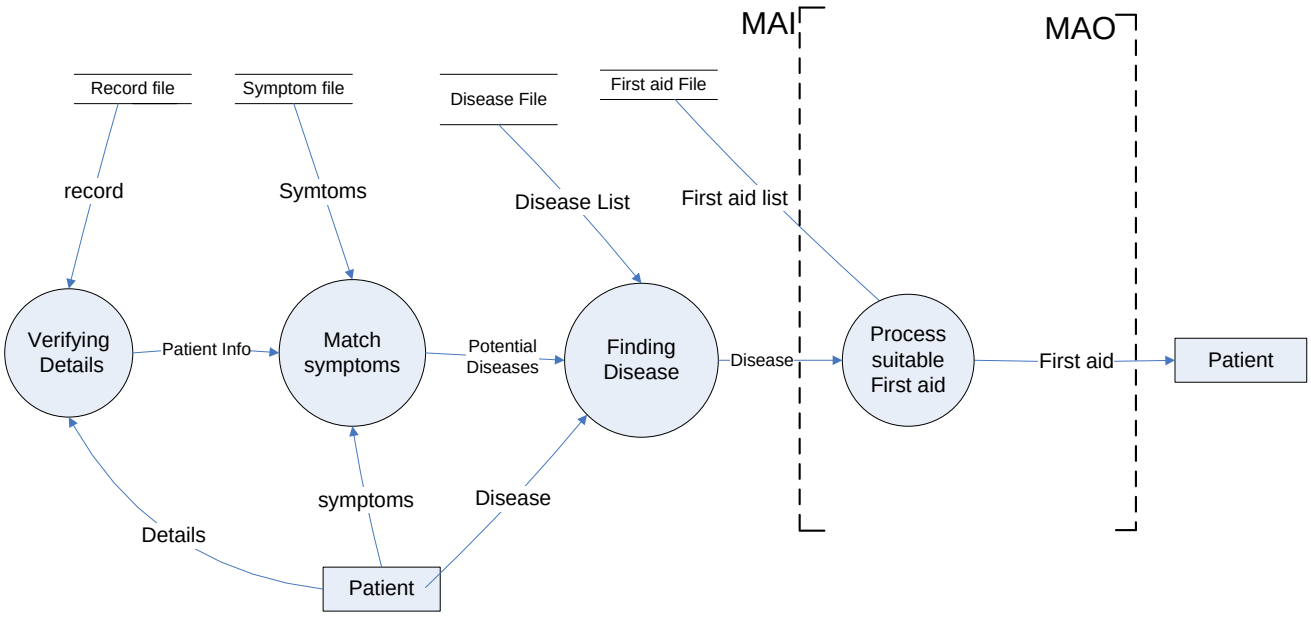

Fig. 3.5

5. First-level Factoring.

Program structure represents a top-down distribution of control. Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work. When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing.

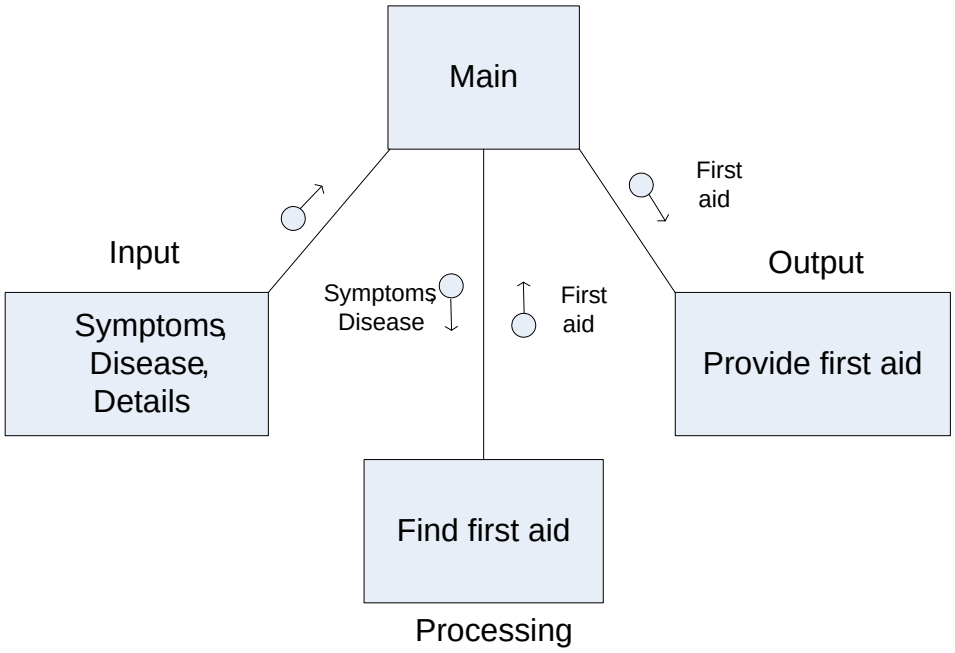

Fig. 3.6

6. Second-level factoring.

Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure.

Input Module

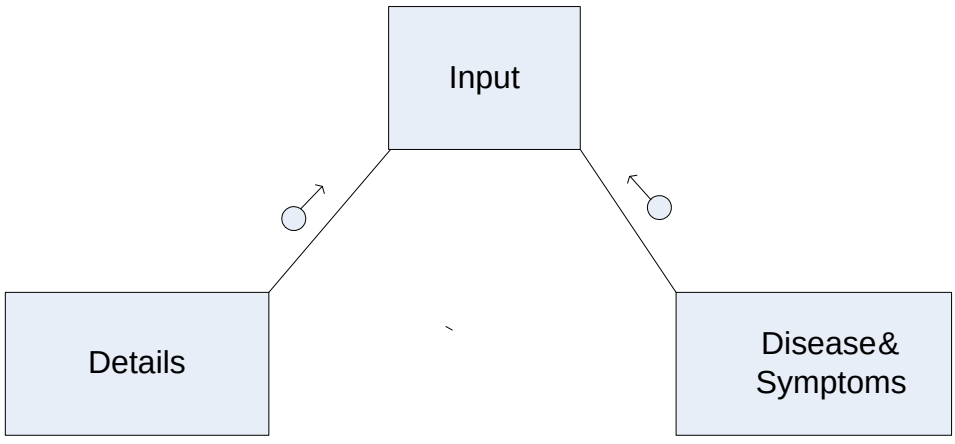

Fig. 3.7

Processing Module

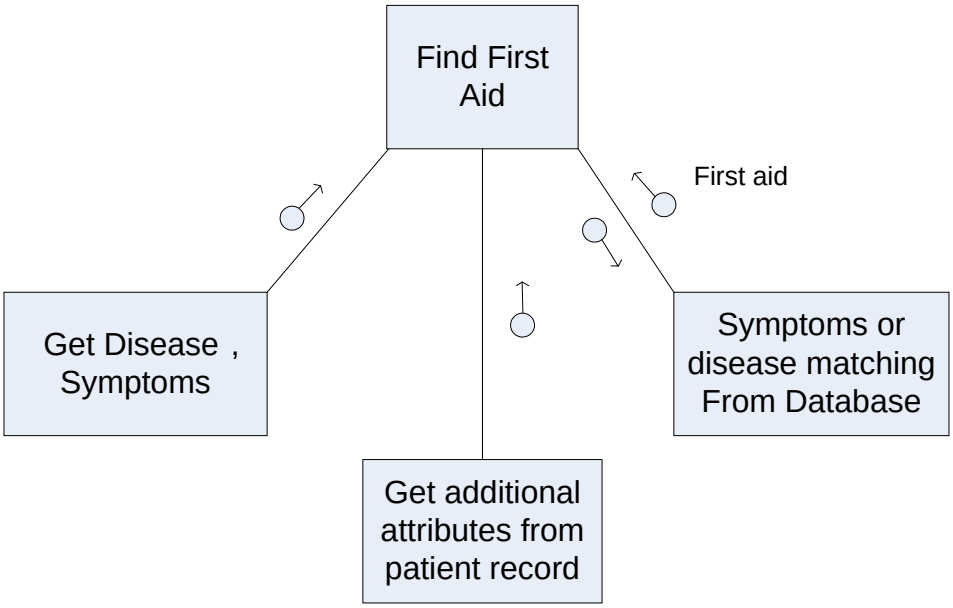

Fig. 3.8

Output Module

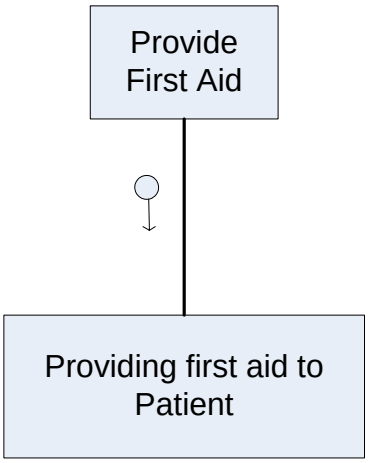

Fig. 3.9

3.3 Interface Design

The interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design.

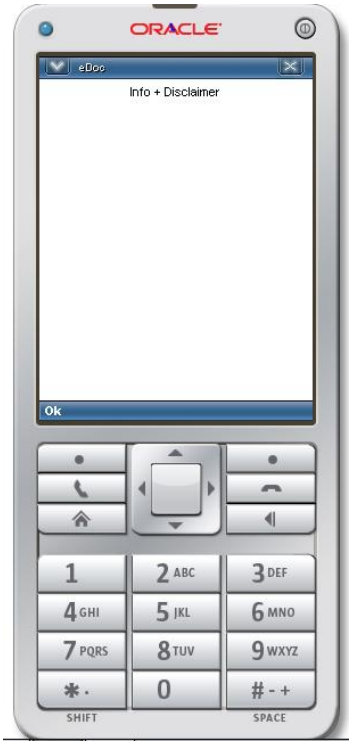

Fig. 3.1

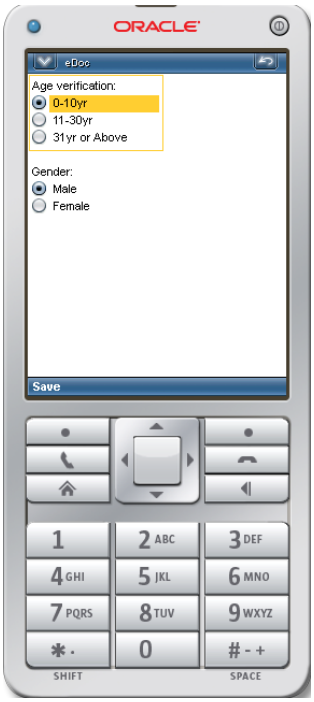

Fig. 3.11

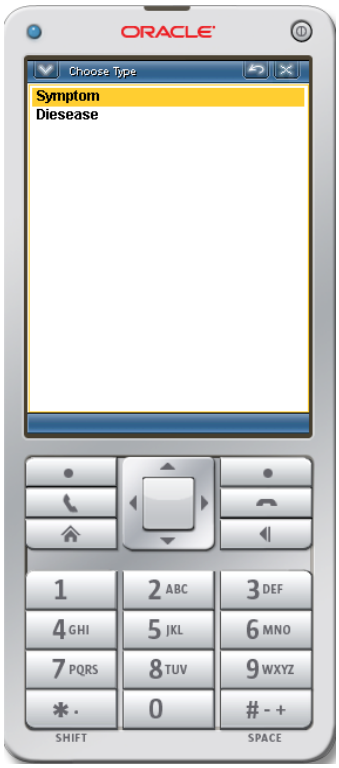

Fig. 3.12

Fig. 3.13


ORACLE
Diesease
List of diesease
1
2 ABC
3 DEF
4 GHI
5 JKL
6 MNO
7 PQRS
8 TUV
9 WXYZ
* .
0
- +
SHIFT
SPACE

Fig. 3.14

4. PROJECT MANAGEMENT

4.1 RISK TABLE

A risk table provides a project manager with a technique for risk projection. The first column lists all the risk associated with the project. This can be accomplished using risk item checklist. The second column lists the category of risk e.g. PS implies Project Size risk. The third column lists the probability of risk occurrence. The fourth column lists the impact value of the risks. The last column lists the risk mitigation, monitoring and management strategies. The column is then arranged according to the priority of impact and appropriate strategies are applied.

Risk	Category	Probability	Impact	RMMM
Size estimated may be low	PS	50%	3	Removing non-essential features
Staff inexperienced	ST	35%	1	Providing Training
Delivery deadline will be tightened	BU	65%	2	Providing core product initially
Technology will not meet expectations	TE	30%	2	Finding Alternative
Unavailable reusable components	TE	80%	3	Construct components
Lack of staff communication	ST	10%	2	Solve personal conflicts
Product size overrun	PS	70%	3	Removing non-essential features
User dissatisfaction	BU	45%	2	Taking feedback and Adding more features
Lack of training on tools	DE	70%	2	Providing Training
Interface complexity	PS	30%	3	Taking feedback and Removing non-essential features
High maintenance cost	BU	35%	2	Improve the integration of the software
Low software portability	TE	20%	2	Use portable Platform

Impact values:

1 ─Catastrophic
2 ─Critical
3 ─Marginal
4 ─Negligible

Categories:

PS →Project size risk
TE →Technical risk
ST →Staff size and experience
BU →Business risk
CU →Customer characteristics
DE →Development environment

4.2 FUNCTION POINTS

Function oriented software matrices use a measure of functionality delivered by the application as a normalization value. Function oriented matrices suggested a measure called a function point. Function points are delivered using an empirical relationship based on countable (direct) measures of software's information domain and assessment of software complexity.

User Inputs: The user inputs provided to the software includes the basic user details like name, date of birth, age, sex, weight, blood group, contact no., email-id, patient id, password & address and hence user inputs can be categorized as average. The number of user inputs is 11.

User Outputs: The outputs provided by the software are expected disease and the suggested first aid. The number of user output is 2, so, it can be categorized as simple.

User Inquiries: The master file made for saving patient record can be used to enquire the patient history. The output files are used to verify the validity of user. There are 2 user inquiries and the complexity of each is simple.

Files: Total number of files in the software is patient record file, disease file, symptom file, first aid file & history file. The complexity is complex as it requires the maintenance of high level database. The number of files is 5.

Interfaces: e-Doc makes use of 4 interfaces that interact with the users communicating with it. The registration form, login form, symptom form and disease description & suggested first aid. The complexity of the interfaces is simple to access and understand.

		WEIGHING FACTOR			
MEASUREMENT PARAMETERS	**COUNT**	**SIMPLE**	**AVERAGE**	**COMPLEX**	
Number of user inputs	11		4		44
Number of user outputs	2	4			8
Number of user inquiries	2	3			6

Number of files	5			15	75
Number of external interfaces	4	5			20
COUNT TOTAL					153

∑ fi (I =1 to 14) complexity adjustment values

S.No.	Factors	Value
1	Does the system require reliable backup and recovery?	5
2	Is data communication required?	4
3	Are there distributed processing functions?	3
4	Is performance critical?	1
5	Will the system run in an existing heavily utilized operational environment?	2
6	Does the system require on-line data entry?	0
7	Does the on line data entry require the input transaction to be over multiple screens or operations?	0
8	Are the master files updated on-line?	5
9	Are the inputs, outputs, files or inquiries complex?	3
10	Is the internal processing complex?	5
11	Is the code designed to be reusable?	2
12	Are conversion and installation included in the design?	2
13	Is the system designed for multiple installations in different organizations?	1
14	Is the application designed to facilitate change and ease of use by the user	5
∑ fi		38

Function point = Count Total*[0.65+0.01*∑fi]

= 153*[0.65+0.01*38]

= 153*[0.65+0.38]

= 153*1.03

Function point = 157.59

5. CODING

```
package eDoc;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import org.netbeans.microedition.lcdui.SplashScreen;

public class eDoc extends MIDlet implements CommandListener {

    private boolean midletPaused = false;

//<editor-fold defaultstate="collapsed" desc=" Generated Fields ">
    private Form form;
    private StringItem stringItem;
    private Form form1;
    private ChoiceGroup choiceGroup2;
    private ChoiceGroup choiceGroup3;
    private ChoiceGroup choiceGroup4;
    private Alert alert;
    private List list;
    private Form form2;
    private StringItem stringItem1;
    private Form form3;
    private StringItem stringItem2;
    private Command okCommand;
```

```
    private Command exitCommand;
    private Command Save;
    private Command backCommand;
    private Command backCommand1;
    private Command exitCommand1;
    private Command backCommand2;
    private Command backCommand3;
    private Command exitCommand2;
    private Font font1;
//</editor-fold>
    /**
     * The eDoc constructor.
     */
    public eDoc() {
    }

//<editor-fold defaultstate="collapsed" desc=" Generated Methods ">
//</editor-fold>
//<editor-fold defaultstate="collapsed" desc=" Generated Method: initialize ">
    /**
     * Initializes the application. It is called only once when the MIDlet is
     * started. The method is called before the
     * <code>startMIDlet</code> method.
     */
    private void initialize() {
```

```
        // write pre-initialize user code here


        // write post-initialize user code here
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Method: startMIDlet ">
    /**
     * Performs an action assigned to the Mobile Device - MIDlet Started point.
     */
    public void startMIDlet() {
        // write pre-action user code here
        switchDisplayable(null, getForm());
        // write post-action user code here
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Method: resumeMIDlet ">
    /**
     * Performs an action assigned to the Mobile Device - MIDlet Resumed point.
     */
    public void resumeMIDlet() {
        // write pre-action user code here


        // write post-action user code here
```

```
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Method: switchDisplayable ">
    /**
     * Switches a current displayable in a display. The
     * <code>display</code> instance is taken from
     * <code>getDisplay</code> method. This method is used by all actions in the
     * design for switching displayable.
     *
     * @param alert the Alert which is temporarily set to the display; if
     * <code>null</code>, then
     * <code>nextDisplayable</code> is set immediately
     * @param nextDisplayable the Displayable to be set
     */
    public void switchDisplayable(Alert alert, Displayable nextDisplayable) {
        // write pre-switch user code here
        Display display = getDisplay();
        if (alert == null) {
            display.setCurrent(nextDisplayable);
        } else {
            display.setCurrent(alert, nextDisplayable);
        }
        // write post-switch user code here
    }
```

```
//</editor-fold>


//<editor-fold defaultstate="collapsed" desc=" Generated Method: commandAction for Displayables ">
    /**
     * Called by a system to indicated that a command has been invoked on a
     * particular displayable.
     *
     * @param command the Command that was invoked
     * @param displayable the Displayable where the command was invoked
     */
    public void commandAction(Command command, Displayable displayable) {
        // write pre-action user code here
        if (displayable == form) {
            if (command == exitCommand) {
                // write pre-action user code here
                exitMIDlet();
                // write post-action user code here
            } else if (command == okCommand) {
                // write pre-action user code here
                switchDisplayable(null, getForm1());
                // write post-action user code here
            }
        } else if (displayable == form1) {
            if (command == Save) {
```

```
            // write pre-action user code here
            switchDisplayable(getAlert(), getList());
            // write post-action user code here
        } else if (command == backCommand2) {
            // write pre-action user code here
            switchDisplayable(null, getForm());
            // write post-action user code here
        }
    } else if (displayable == form2) {
        if (command == backCommand1) {
            // write pre-action user code here
            switchDisplayable(null, getList());
            // write post-action user code here
        }
    } else if (displayable == form3) {
        if (command == backCommand) {
            // write pre-action user code here
            switchDisplayable(null, getList());
            // write post-action user code here
        }
    } else if (displayable == list) {
        if (command == List.SELECT_COMMAND) {
            // write pre-action user code here
            listAction();
            // write post-action user code here
```

```
            } else if (command == backCommand3) {
                // write pre-action user code here
                switchDisplayable(null, getForm1());
                // write post-action user code here
            } else if (command == exitCommand2) {
                // write pre-action user code here
                exitMIDlet();
                // write post-action user code here
            }
        }
        // write post-action user code here
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: form ">
    /**
     * Returns an initialized instance of form component.
     *
     * @return the initialized component instance
     */
    public Form getForm() {
        if (form == null) {
            // write pre-init user code here
            form = new Form("eDoc", new Item[]{getStringItem()});
            form.addCommand(getOkCommand());
```

```
            form.addCommand(getExitCommand());
            form.setCommandListener(this);
            // write post-init user code here
        }
        return form;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: stringItem ">
    /**
     * Returns an initialized instance of stringItem component.
     *
     * @return the initialized component instance
     */
    public StringItem getStringItem() {
        if (stringItem == null) {
            // write pre-init user code here
            stringItem = new StringItem("Info + Disclaimer", "");
            stringItem.setLayout(ImageItem.LAYOUT_CENTER);
            // write post-init user code here
        }
        return stringItem;
    }
//</editor-fold>
```

```
//<editor-fold defaultstate="collapsed" desc=" Generated Getter: okCommand ">
    /**
     * Returns an initialized instance of okCommand component.
     *
     * @return the initialized component instance
     */
    public Command getOkCommand() {
        if (okCommand == null) {
            // write pre-init user code here
            okCommand = new Command("Ok", Command.OK, 0);
            // write post-init user code here
        }
        return okCommand;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: exitCommand ">
    /**
     * Returns an initialized instance of exitCommand component.
     *
     * @return the initialized component instance
     */
    public Command getExitCommand() {
        if (exitCommand == null) {
            // write pre-init user code here
```

```
            exitCommand = new Command("Exit", Command.EXIT, 0);
            // write post-init user code here
        }
        return exitCommand;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: form1 ">
    /**
     * Returns an initialized instance of form1 component.
     *
     * @return the initialized component instance
     */
    public Form getForm1() {
        if (form1 == null) {
            // write pre-init user code here
            form1 = new Form("eDoc", new Item[]{getChoiceGroup2(), getChoiceGroup4(),
getChoiceGroup3()});
            form1.addCommand(getSave());
            form1.addCommand(getBackCommand2());
            form1.setCommandListener(this);
            // write post-init user code here
        }
        return form1;
    }
```

```
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: choiceGroup2 ">
    /**
     * Returns an initialized instance of choiceGroup2 component.
     *
     * @return the initialized component instance
     */
    public ChoiceGroup getChoiceGroup2() {
        if (choiceGroup2 == null) {
            // write pre-init user code here
            choiceGroup2 = new ChoiceGroup("Age verification:", Choice.EXCLUSIVE);
            choiceGroup2.append("0-10yr", null);
            choiceGroup2.append("11-30yr", null);
            choiceGroup2.append("31yr or Above", null);
            choiceGroup2.setSelectedFlags(new boolean[]{false, false, false});
            // write post-init user code here
        }
        return choiceGroup2;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: choiceGroup3 ">
    /**
     * Returns an initialized instance of choiceGroup3 component.
```

```
     *
     * @return the initialized component instance
     */
    public ChoiceGroup getChoiceGroup3() {
        if (choiceGroup3 == null) {
            // write pre-init user code here
            choiceGroup3 = new ChoiceGroup("Gender:", Choice.EXCLUSIVE);
            choiceGroup3.append("Male", null);
            choiceGroup3.append("Female", null);
            choiceGroup3.setSelectedFlags(new boolean[]{false, false});
            // write post-init user code here
        }
        return choiceGroup3;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: choiceGroup4 ">
    /**
     * Returns an initialized instance of choiceGroup4 component.
     *
     * @return the initialized component instance
     */
    public ChoiceGroup getChoiceGroup4() {
        if (choiceGroup4 == null) {
            // write pre-init user code here
```

```
            choiceGroup4 = new ChoiceGroup("", Choice.MULTIPLE);
            // write post-init user code here
        }
        return choiceGroup4;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: Save ">
    /**
     * Returns an initialized instance of Save component.
     *
     * @return the initialized component instance
     */
    public Command getSave() {
        if (Save == null) {
            // write pre-init user code here
            Save = new Command("Save", Command.OK, 0);
            // write post-init user code here
        }
        return Save;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: alert ">
    /**
```

```
     * Returns an initialized instance of alert component.
     *
     * @return the initialized component instance
     */
    public Alert getAlert() {
        if (alert == null) {
            // write pre-init user code here
            alert = new Alert("eDoc", "Thanks!  Your Detail has been save ", null, null);
            alert.setTimeout(Alert.FOREVER);
            // write post-init user code here
        }
        return alert;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: list ">
    /**
     * Returns an initialized instance of list component.
     *
     * @return the initialized component instance
     */
    public List getList() {
        if (list == null) {
            // write pre-init user code here
            list = new List("Choose Type", Choice.IMPLICIT);
```

```
            list.append("Symptom", null);
            list.append("Diesease", null);
            list.addCommand(getBackCommand3());
            list.addCommand(getExitCommand2());
            list.setCommandListener(this);
            list.setSelectedFlags(new boolean[]{true, true});
            list.setFont(0, getFont1());
            list.setFont(1, getFont1());
            // write post-init user code here
        }
        return list;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Method: listAction ">
    /**
     * Performs an action assigned to the selected list element in the list
     * component.
     */
    public void listAction() {
        // enter pre-action user code here
        String __selectedString = getList().getString(getList().getSelectedIndex());
        if (__selectedString != null) {
            if (__selectedString.equals("Symptom")) {
                // write pre-action user code here
```

```
            switchDisplayable(null, getForm2());
            // write post-action user code here
        } else if (__selectedString.equals("Diesease")) {
            // write pre-action user code here
            switchDisplayable(null, getForm3());
            // write post-action user code here
        }
    }
    // enter post-action user code here
}
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: form2 ">
    /**
     * Returns an initialized instance of form2 component.
     *
     * @return the initialized component instance
     */
    public Form getForm2() {
        if (form2 == null) {
            // write pre-init user code here
            form2 = new Form("Symptoms", new Item[]{getStringItem1()});
            form2.addCommand(getBackCommand1());
            form2.setCommandListener(this);
            // write post-init user code here
```

```
        }
        return form2;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: form3 ">
    /**
     * Returns an initialized instance of form3 component.
     *
     * @return the initialized component instance
     */
    public Form getForm3() {
        if (form3 == null) {
            // write pre-init user code here
            form3 = new Form("Diesease", new Item[]{getStringItem2()});
            form3.addCommand(getBackCommand());
            form3.setCommandListener(this);
            // write post-init user code here
        }
        return form3;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: backCommand ">
    /**
```

```
     * Returns an initialized instance of backCommand component.
     *
     * @return the initialized component instance
     */
    public Command getBackCommand() {
        if (backCommand == null) {
            // write pre-init user code here
            backCommand = new Command("Back", Command.BACK, 0);
            // write post-init user code here
        }
        return backCommand;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: backCommand1 ">
    /**
     * Returns an initialized instance of backCommand1 component.
     *
     * @return the initialized component instance
     */
    public Command getBackCommand1() {
        if (backCommand1 == null) {
            // write pre-init user code here
            backCommand1 = new Command("Back", Command.BACK, 0);
            // write post-init user code here
```

```
        }
        return backCommand1;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: exitCommand1 ">
    /**
     * Returns an initialized instance of exitCommand1 component.
     *
     * @return the initialized component instance
     */
    public Command getExitCommand1() {
        if (exitCommand1 == null) {
            // write pre-init user code here
            exitCommand1 = new Command("Exit", Command.EXIT, 0);
            // write post-init user code here
        }
        return exitCommand1;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: backCommand2 ">
    /**
     * Returns an initialized instance of backCommand2 component.
     *
```

```
     * @return the initialized component instance
     */
    public Command getBackCommand2() {
        if (backCommand2 == null) {
            // write pre-init user code here
            backCommand2 = new Command("Back", Command.BACK, 0);
            // write post-init user code here
        }
        return backCommand2;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: backCommand3 ">
    /**
     * Returns an initialized instance of backCommand3 component.
     *
     * @return the initialized component instance
     */
    public Command getBackCommand3() {
        if (backCommand3 == null) {
            // write pre-init user code here
            backCommand3 = new Command("Back", Command.BACK, 0);
            // write post-init user code here
        }
        return backCommand3;
```

```
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: exitCommand2 ">
    /**
     * Returns an initialized instance of exitCommand2 component.
     *
     * @return the initialized component instance
     */
    public Command getExitCommand2() {
        if (exitCommand2 == null) {
            // write pre-init user code here
            exitCommand2 = new Command("Exit", Command.EXIT, 0);
            // write post-init user code here
        }
        return exitCommand2;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: font1 ">
    /**
     * Returns an initialized instance of font1 component.
     *
     * @return the initialized component instance
     */
```

```
    public Font getFont1() {
        if (font1 == null) {
            // write pre-init user code here
            font1 = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD, Font.SIZE_LARGE);
            // write post-init user code here
        }
        return font1;
    }
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: stringItem1 ">
    /**
     * Returns an initialized instance of stringItem1 component.
     *
     * @return the initialized component instance
     */
    public StringItem getStringItem1() {
        if (stringItem1 == null) {
            // write pre-init user code here
            stringItem1 = new StringItem("List of symptoms", "");
            stringItem1.setLayout(ImageItem.LAYOUT_CENTER);
            // write post-init user code here
        }
        return stringItem1;
    }
```

```
//</editor-fold>

//<editor-fold defaultstate="collapsed" desc=" Generated Getter: stringItem2 ">
    /**
     * Returns an initialized instance of stringItem2 component.
     *
     * @return the initialized component instance
     */
    public StringItem getStringItem2() {
        if (stringItem2 == null) {
            // write pre-init user code here
            stringItem2 = new StringItem("List of diesease", "");
            stringItem2.setLayout(ImageItem.LAYOUT_CENTER);
            // write post-init user code here
        }
        return stringItem2;
    }
//</editor-fold>


    /**
     * Returns a display instance.
     *
     * @return the display instance.
     */
    public Display getDisplay() {
```

```
    return Display.getDisplay(this);
}

/**
 * Exits MIDlet.
 */
public void exitMIDlet() {
    switchDisplayable(null, null);
    destroyApp(true);
    notifyDestroyed();
}

/**
 * Called when MIDlet is started. Checks whether the MIDlet have been
 * already started and initialize/starts or resumes the MIDlet.
 */
public void startApp() {
    if (midletPaused) {
        resumeMIDlet();
    } else {
        initialize();
        startMIDlet();
    }
    midletPaused = false;
}
```

```
    /**
     * Called when MIDlet is paused.
     */
    public void pauseApp() {
        midletPaused = true;
    }

    /**
     * Called to signal the MIDlet to terminate.
     *
     * @param unconditional if true, then the MIDlet has to be unconditionally
     * terminated and all resources has to be released.
     */
    public void destroyApp(boolean unconditional) {
    }
}
```

6. SOFTWARE TESTING

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and code generation. Software Testing is the one step in the software process that could be viewed as destructive rather than constructive. Testing is the process of executing a program with the intent of finding an error. Testing can be conducted in two ways:

1.) White-Box Testing
2.) Black- Box Testing

6.1 WHITE-BOX TESTING

White-box testing sometimes called "glass-box testing", is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that 1) guarantee that all independent paths within a module be exercised at least once, 2) exercise all logical decisions on their true and false sides, 3) execute all loops at their boundaries and within their operational bounds, and 4) exercise all internal data structures to ensure validity.

6.1.1 BASIS PATH TESTING

Basis path testing is a white box testing technique. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least once during testing.

6.1.1.1 Flow Graph Notation

The flow graph depicts logical control flow. Each structured construct has a corresponding flow graph symbol. Each circle is called a flow graph node which represents one or more procedural statements. The arrows on the flow graph are called edges or links which represent flow of control. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

6.1.1.2 Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. The value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program and provides with an upper bound for the number of test cases that must be conducted to ensure that all statements have been executed at least once. An independent path is any path through the program that introduces at least one

new set of processing statements or a new condition. Cyclomatic complexity is computed in one of the three ways:

1.) The number of regions of the flow graph corresponds to the Cyclomatic complexity.

2.) Cyclomatic complexity (G), for a flow graph, G, is defined as: V(G) =E-N+2 where E is the number of flow graph edges; N is the number of flow graph nodes.

3.) Cyclomatic complexity, V(G), for a flow graph, G, is also defined as (G) = P+1 where P is the number of predicate nodes contained in the flow graph.

6.1.1.3 Deriving Test Case

The basis path testing method can be applied to a procedural design or to source code.

The following steps can be applied to derive the basis set:

Step 1: **Using the design or code as a foundation, draw a corresponding flow graph**. A flow graph is created using the symbols prescribed. Referring to the PDL for Finding_disease, a flow graph is created by numbering those PDL statements that will be mapped to corresponding flow graph nodes.

PROCEDURE: Finding_disease

*This procedure checks that the data values entered by the user are stored into the database, the password entered by the user matches and that the entered username is available and that all the entries have been filled completely.

INTERFACE RETURNS: message conforming disease found, message showing availability of disease entered/computed, a new screen showing the disease, cure and precautions;

INTERFACE ACCEPTS: symptoms, disease;

TYPE: symptoms, symptoms_again, disease and disease_again.

TYPE1: symptoms and symptoms_again.

TYPE IS TEXT;

TYPE1 IS SYMPTOMS;

IF symptoms == stored_symptoms **- (1)**
THEN display the disease found with details **- (2)**
IF user wants a new search **- (3)**
THEN go back to the symptoms screen
ELSE skip **- (4)**

ENDIF **- (5)**

ELSE symptoms != stored_symptoms; **- (6)**

THEN ask user if it wants to exit or re-enter **- (7)**

IF user wants to re-enter **- (8)**

THEN go back to the previous screen

ELSE skip **- (9)**

ENDIF **- (10)**

ENDIF **- (11)**

END Finding_Disease

The flow graph is mapped into appropriate flow graph in the following manner. The flow graph has 11 Nodes, 3 Predicate Nodes and 13 Edges. The flow graph creates 4 internal regions spanned by the edges of the flow graph and one external region which is outside the graph. The information obtained from the flow graph can be used to compute Cyclomatic complexity to help in the design of test cases to check the correctness of procedural detail. It gives us the status of every statement at a given instance.

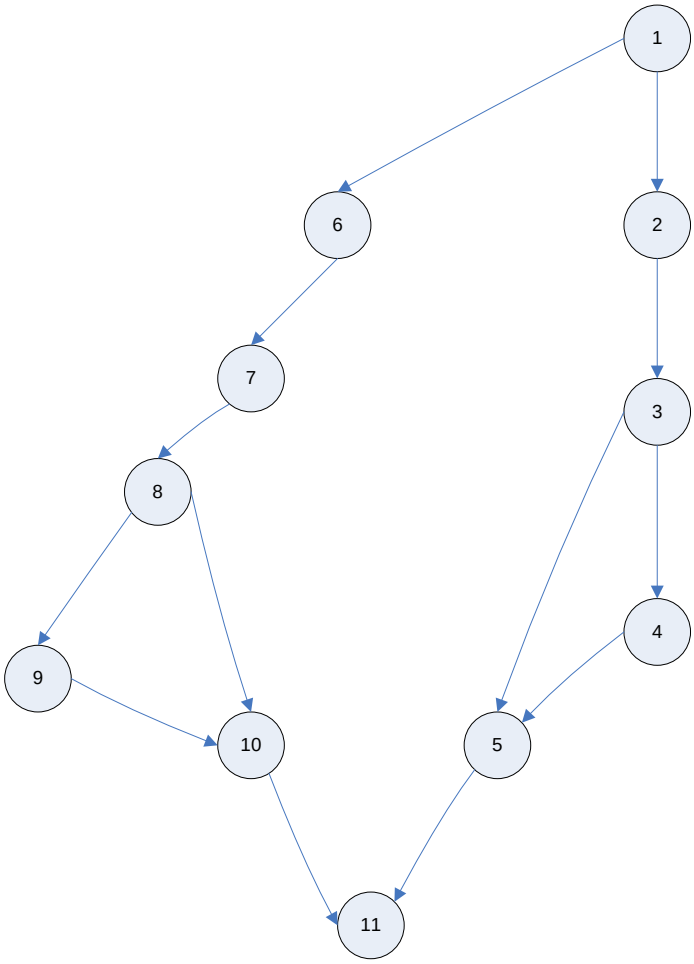

Fig 6.2 Flow graph of the procedure Finding_disease

Step 2: Determine the Cyclomatic complexity of the resultant flow graph. The Cyclomatic complexity can be determined by any of the tree stated algorithms.

Algorithm 1: The number of regions in the flow graph = 4. Cyclomatic Complexity = 4.

Algorithm 2: Number of edges = 11, Number of nodes = 9

Hence, V(G) = E-N+2

= 13-11+2 = 4

Algorithm 3: Number of predicate nodes = 3. Hence the Cyclomatic complexity is

V(G) = 3+1

= 3+1 = 4

The value of V(G) provides us with an upper bound for the number of independent paths that form the basis set and by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Step 3: Determine the basis set of linearly independent paths. The value of V(G) provides the number of linearly independent paths through the program control structure. In the case of procedure Finding_disease, we expect to specify 5 paths:

Path 1: 1-2-3-5-11

Path 2: 1-2-3-4-5-11

Path 3: 1-6-8-10-11

Path 4: 1-6-8-9-10-11

In this case node 1, 3 and 8 are predicate nodes.

Step 4: Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set described are:

Path 1 test case:
symptoms = NOT_NULL
User wants to exit.
Expected results: Symptoms will be matched and Disease its related details will be shown then the user will exit the application.

Path 2 test case:
symptoms = NOT_NULL
User wants a new search.
Expected results: Symptoms will be matched and Disease its related details will be shown then the user will start a new disease search.

Path 3 test case:
symptoms = NULL/NOT-MATCHED
User wants to exit.
Expected results: Symptoms will not be matched and Disease will not be shown then the user will exit.

Path 4 test case:
symptoms = NULL/NOT-MATCHED
User wants a new search.
Expected results: Symptoms will not be matched and Disease will not be shown the user will start a new disease search.

Each test case is executed and compared to expected results. Once all the tests have been completed, the tester can be sure that all statements in the program have been executed at least once. Some independent paths cannot be tested in standalone fashion. That is, the combination of data required to traverse such a path cannot be achieved in the normal flow of program.

6.2 BLACK BOX TESTING

Black-box testing, also called behavioral testing, focuses on the functional requirement of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

Graph based testing methods:

The first step in black-box testing is to understand the objects. To accomplish this, its required to create a graph- a collection of nodes that represents objects; links that represent the relationships between objects; node weights that describe the properties of a node; and link weights that describe some characteristics of a link.

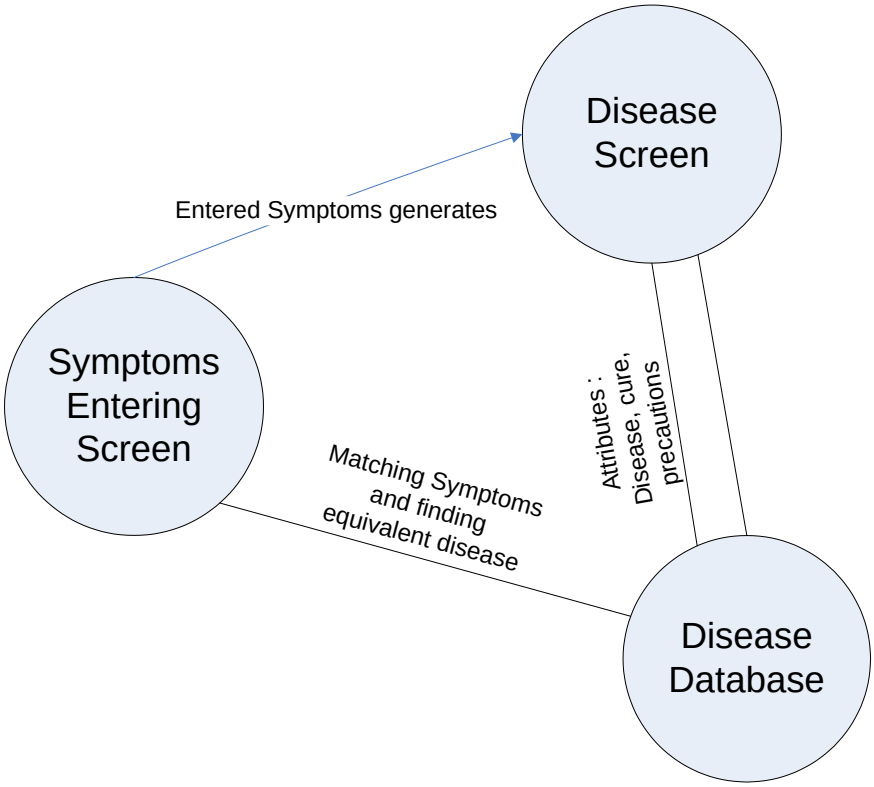

Fig 6.3 Graph notation

The symbolic representation of a graph is shown in Fig 16.3 .Nodes are represented as circles connected by links that takes a number of different forms. A directed link indicates that a relationship moves in only one direction. A bidirectional link, also called symmetric link. Parallel lines used when number of different relationships are established. Referring to the figure menu select generates registration window. Then this registration document will generate student's database. .

Behavioral testing method used**: Data Flow Modeling.** The nodes are data objects and the links are the transformations that occur to translate one data object to another.

7. Reference

The Following books are being used to fulfill the requirements of the proposed project and are helpful in the understanding, development and the maintenance of the project.

1. J2ME Complete Reference by Herbert Schildt
2. Where there is no doctor a village heath care handbook by David Werner
3. Software Engineering by Roger S. Pressman
4. Integrated approach to software engineering by Pankaj Jalote

Websites Referred:

www.java.sun.com
www.google.com
www.oracle.com