From PSM to Code

And summery of Development Process

For Insulin Pump

Written by:

Morteza Yousef-Sanati

Hamid Mohammad-Gholizadeh

Assignment 4

Course: 703 Software Design

Professor: Dr. Tom Maibaum

April 2011

Table of Contents

Development Process

Our system development started defining some elementary usescase for simulating an insullin pump. You can see the overall use case diagram below.

Figure 1: Use case diagram for the insulin pump simulator

After defining these use case we started to design the class diagrams based on these use cases. First design PIM and then tried to get the PSM from that and finally reach the code. For example you can see a part a PIM class diagram in the figure bellow

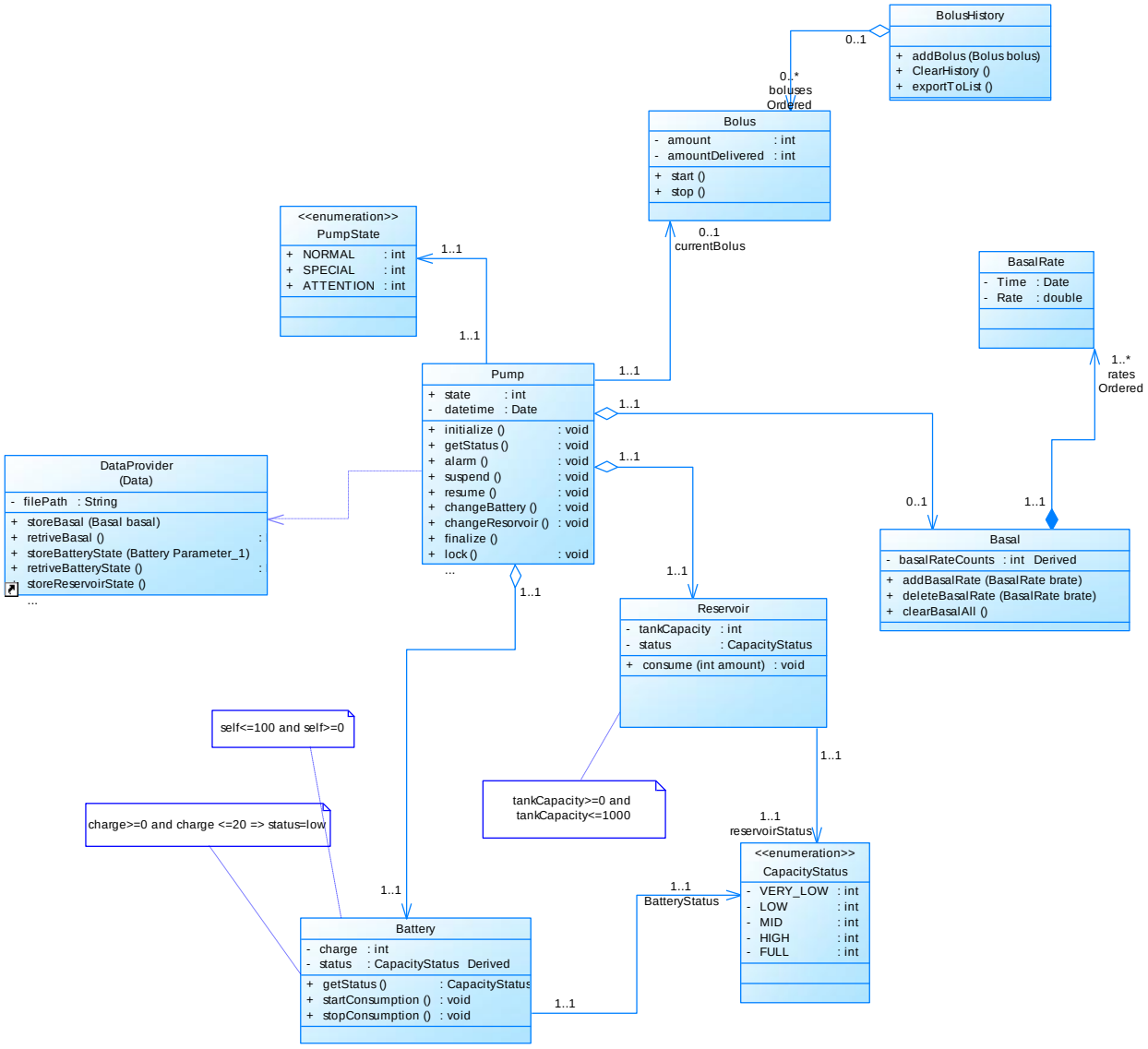


Figure 2: PIM Class diagram of the controller package

Our design is based on 3-tier architecture which is reflected by three packages: GUI, Controller and Data. This decision is made because of separating the functionality of the system between different packages. For

doing this we mostly considered less coupling between packages and more cohesion inside each package principle.

GUI includes classes related to GUI Design like, Frames, Buttons, Panels and etc. which is constructing our user interface. Controller includes core classes like classes which control the pump functionality and Data package contains a class which is responsible for storing and retrieving system data.

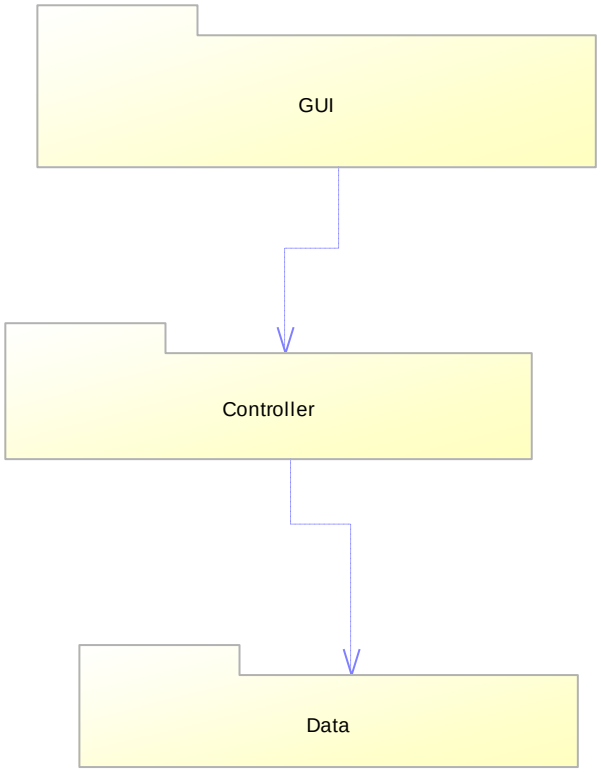


Figure3 :System Design Architecture

After doing PIM design we refine the models in PIM models to get the PSM models. You can find detailed explanation for this refinement in our

previous document. As an example you see below the PSM class diagram for the previous PIM class diagram already demonstrated.

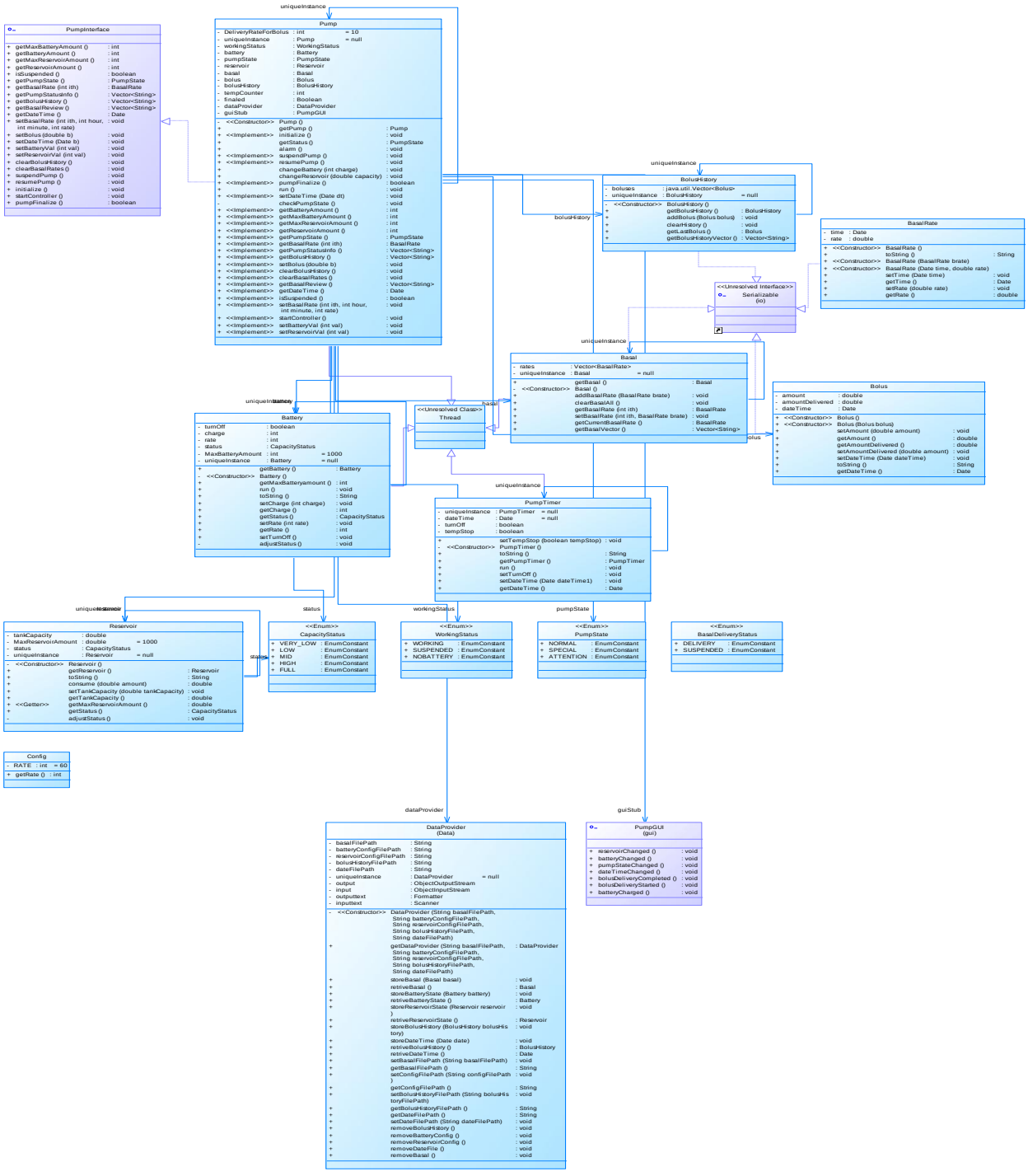


Figure 4: PSM class diagram for the controller package

For modual(package) communication we put an Interface between these modules to ensure the loose coupling between them. Two interfaces named PumpGUI and PumpInterface provide Modula interfaces for the gui and ccontroller packags respectively. Two classes that implement these two interfaces are MainFram and Pump respectively as you see in figure below:

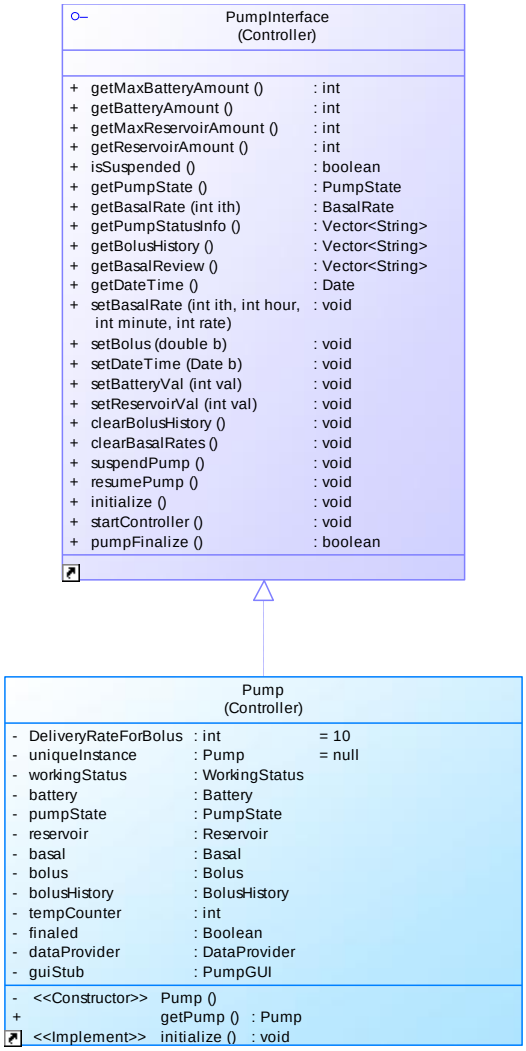


Figure 5 - PumpInterface in controller package. Pump implements this interface

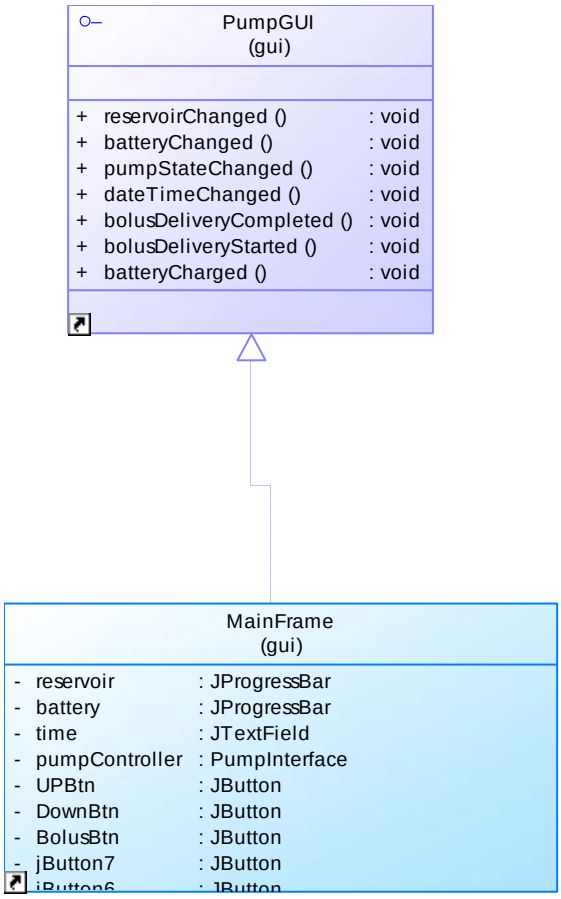


Figure 6- PumpGui Interface is an interface for gui modual. MainFrame implements this interface. Most attributes and operations of Mainframe is omitted in this diagram.

PSM to Code Movement

Almost all of our class diagram conforms to the code, i.e. they are reflected in the code without any modification. We used a code generator of PowerDesigner to do that and reverse Engineering of the tool helped us to keep the consistency between the code and the diagrams. Then we don't have that much modification while moving to the code.

You can see an screenshot of the simulator in Figure below:

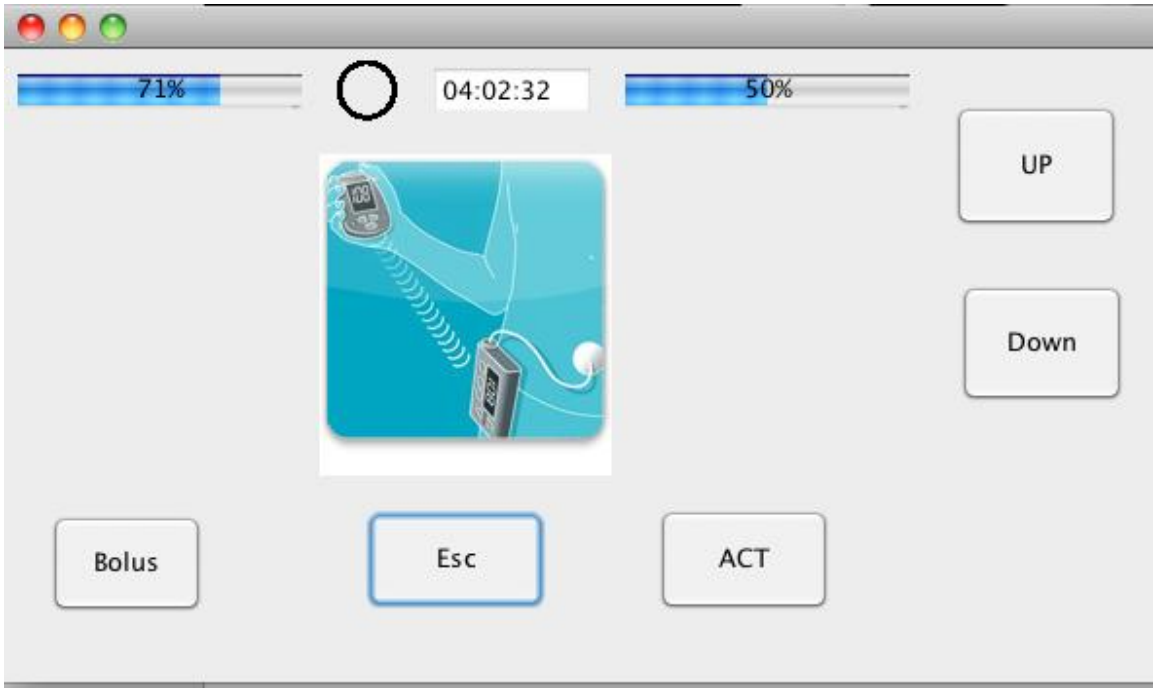


Figure 7: an Screenshot of the running insulin pump simulator

In the following we discuss some of the techniques we used for implementing statechart diagrams and some algorithm implementations.

MainScreen To Code

What we mean by the main screen is the screen section of the simulator. For closing our simulation to a real insulin pump, we limited the number of lines in our main screen. We implemented our screen using a JPanel component of the JAVA. This panel contains three Panels which represents the three line of the screen. In each line we used a JLabel to reprent the text which can be displayed in each line. So we defined a template for the main screen named TemplateScreen. This template screen implements all the commonality necessary for the screens to show something inside.Below is the code for implementation of this class.

```
public abstract class TemplateScreen extends javax.swing.JPanel {
	private JPanel jPanel1;
	private JLabel jLabel1;
	private JPanel line3;
	private JPanel line2;
	private JPanel line1;
	public JPanel getLine3() {
		return line3;
	}

	public JPanel getLine2() {
		return line2;
	}

	public JPanel getLine1() {
		return line1;
	}

	public void setTitle(String title) {
		this.title = title;
		this.jLabel1.setText(title);
	}

	private String title;

	public TemplateScreen() {
		super();
		initGUI();

	}

	private void initGUI() {
		try {
			GridLayout thisLayout = new GridLayout(4, 1);
			thisLayout.setHgap(5);
			thisLayout.setVgap(5);
			thisLayout.setColumns(1);
			thisLayout.setRows(4);
			this.setLayout(thisLayout);
			this.setPreferredSize(new java.awt.Dimension(301, 240));
			{
				jPanel1 = new JPanel();
				this.add(jPanel1);
				jPanel1.setPreferredSize(new java.awt.Dimension(301, 40));
				{
					jLabel1 = new JLabel();
					FlowLayout jLabel1Layout = new FlowLayout();
					jLabel1.setLayout(jLabel1Layout);
					jPanel1.add(jLabel1);
					jLabel1.setText("");
					jLabel1.setFont(new Font("Times", Font.BOLD, 20));
					jLabel1.setPreferredSize(new java.awt.Dimension(217, 27));
				}
			}
```

```
                        {
                                line1 = new JPanel();
                                this.add(line1);
                        }
                        {
                                line2 = new JPanel();
                                this.add(line2);
                        }
                        {
                                line3 = new JPanel();
                                this.add(line3);
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
        protected abstract void makeScreen();

        public void showMsg(String msg) {
                this.getLine1().removeAll();
                this.getLine2().removeAll();
                this.getLine3().removeAll();

                JLabel m=new JLabel(msg);
                m.setFont(new Font("Times",         Font.ITALIC, 15));

                this.getLine2().add(m);
                this.repaint();
                this.validate();

        }
}
```

Figure 8: Implementation of Template screen

As you see in the code above there are three Lines, named Line1, Line2 and Line 3 representing the three lines of the screen. These lines are used to show the information to the user.

In the picture bellow you can see an snapshot of setting basal rates in the system. This screen is a class that implements the Templatescreen above.

Figure 9: setBasal Screen uses the TemplateScreen for showing its component to the user.

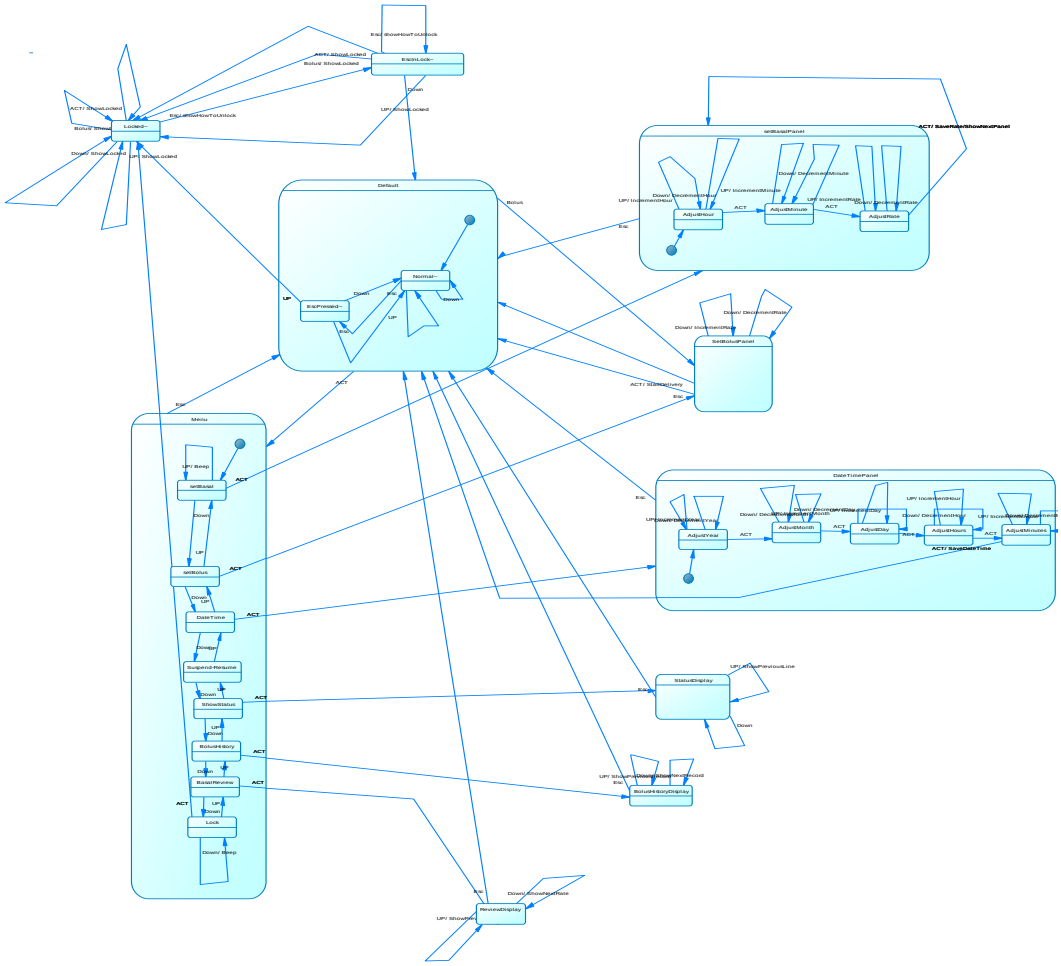
MainFrame StateChart To code

We defined an enumeration named GUIStatus to represent the different states in the statechart diagram below.

According to the states of the MainFram, functionalities of the Buttons changes. As we stated in our previous documentations, We exploited Command Pattern to deal with this problem. The state change of the MainFrame class is encapsulated in a method named "changeStatusTo(guiStatus)" as you see in figure 2.

Every command in the system which like to change the status of the system, call this method and pass the status as a parameter to that. As it is clear in this method, for each status of the system a collection of command class is assigned. For example for the setDateTime status, the corresponding command Classes will be: DateTimeACTCommand, VoidCommand, DateTimeESCCommand, DateTimeUPCommand and

DateTimeDownCommand. "VoidCommand" represents no actions! While this command is assigned to a button nothing will happen when user clicks that button. As another example, in "Default" status, the ACTCommand will show the menu to the user and this command is assigned to the ACTButton in the system.



```
public void changeStatusTo(GUIStatus guiStatus) {
                switch (guiStatus) {
```

```
case Menu:
        this.setACTCommand(new gui.menu.ACTCommand());
        this.setBolusCommand(new gui.menu.BolusCommand());
        this.setESCCommand(new gui.menu.ESCCommand());
        this.setUPCommand(new gui.menu.UPCommand());
        this.setDownCommand(new gui.menu.DownCommand());
        break;
case Default:
        this.setACTCommand(new DefaultACTCommand());
        this.setBolusCommand(new DefaultBolusCommand());
        this.setESCCommand(new VoidCommand());
        this.setUPCommand(new VoidCommand());
        this.setDownCommand(new VoidCommand());
        break;
case SetBalsal:
        this.setACTCommand(new BasalACTCommand());
        this.setBolusCommand(new VoidCommand());
        this.setESCCommand(new BasalESCCommand());
        this.setUPCommand(new BasalUPCommand());
        this.setDownCommand(new BasalDownCommand());
        break;
case SetBolus:
        this.setACTCommand(new BolusACTCommand());
        this.setBolusCommand(new VoidCommand());
        this.setESCCommand(new BolusESCCommand());
        this.setUPCommand(new BolusUPCommand());
        this.setDownCommand(new BolusDownCommand());
        break;
case SetDateTime:
        this.setACTCommand(new DateTimeACTCommand());
        this.setBolusCommand(new VoidCommand());
        this.setESCCommand(new DateTimeESCCommand());
        this.setUPCommand(new DateTimeUPCommand());
        this.setDownCommand(new DateTimeDownCommand());
        break;
case SuspendResum:
        this.setACTCommand(new DefaultACTCommand());
        this.setBolusCommand(new DefaultBolusCommand());
        this.setESCCommand(new DefaultESCCommand());
        this.setUPCommand(new VoidCommand());
        this.setDownCommand(new VoidCommand());
        break;
case StatusDisplay:
        this.setACTCommand(new VoidCommand());
        this.setBolusCommand(new VoidCommand());
        this.setESCCommand(new ListDisplayESCCommand());
        this.setUPCommand(new ListDisplayUPCommand());
        this.setDownCommand(new ListDisplayDownCommand());
        break;
case BolusHistoryDisplay:
        this.setACTCommand(new VoidCommand());
        this.setBolusCommand(new VoidCommand());
        this.setESCCommand(new ListDisplayESCCommand());
        this.setUPCommand(new ListDisplayUPCommand());
```

```
                this.setDownCommand(new ListDisplayDownCommand());
                break;
        case BasalReviewDisplay:
                this.setACTCommand(new VoidCommand());
                this.setBolusCommand(new VoidCommand());
                this.setESCCommand(new ListDisplayESCCommand());
                this.setUPCommand(new ListDisplayUPCommand());
                this.setDownCommand(new ListDisplayDownCommand());
                break;
        case Lock:
                this.setACTCommand(new LockACTCommand());
                this.setBolusCommand(new LockBolusCommand());
                this.setESCCommand(new LockEscCommand());
                this.setUPCommand(new LockUPCommand());
                this.setDownCommand(new LockDownCommand());
                break;

        }
        this.validate();
}
```

Figure 10: method "changeStateTo()" controlles the state change in the "MainFrame" Class

Menu Staechart to Code:

We should also implement the menu according to the statechart in figure 3.

For implementing that the state of the menu is represented by a "status" property of the MenuScreen, which is of type "MenuStatus". MenuStatus is an enumeration type of: "ViewUP", "ViewMID" and "ViewDown". **"D" and "U" in the statechart diagram represent Down and UP commands respectively.** You can see the implementation of the Down Command in Figure 4.

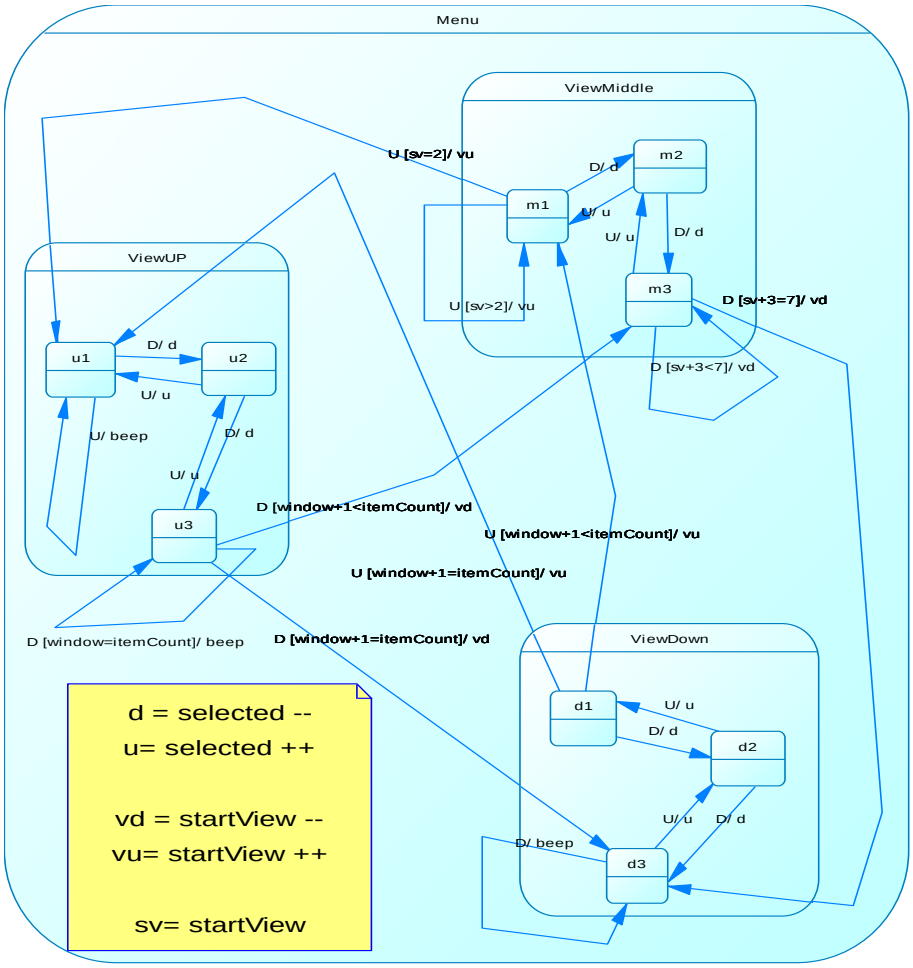


Figure 11: An state chart for "MenuScreen" Class

```
public void execute(Object obj) {
		// TODO Auto-generated method stub
		MenuScreen s=(MenuScreen)MainFrame.getFrame().getScreenContent();
		switch (s.getStatus()){
			case ViewUP:
				switch (s.getSelected()){
				case 1:
					s.setSelected(s.getSelected()+1);
					break;
				case 2:
					s.setSelected(s.getSelected()+1);
					break;
				case 3:
					s.setStartView(s.getStartView()+1);
					s.setStatus(MenuStatus.ViewMID);
					break;
				}
				break;
```

```
                case ViewMID:
                        switch (s.getSelected()){
                        case 1:
                                s.setSelected(s.getSelected()+1);
                                break;
                        case 2:
                                s.setSelected(s.getSelected()+1);
                                break;
                        case 3:
                                if (s.getStartView()==s.ITEM_COUNT-4){
                                        s.setStartView(s.getStartView()+1);
                                        s.setStatus(MenuStatus.ViewDown);
                                }
                                else{ //(s.getEndView()<7)
                                        s.setStartView(s.getStartView()+1);
                                }
                                break;
                        }
                        break;
                case ViewDown:
                        switch (s.getSelected()){
                        case 1:
                                s.setSelected(s.getSelected()+1);
                                break;
                        case 2:
                                s.setSelected(s.getSelected()+1);
                                break;
                        case 3:
                                //beep
                                java.awt.Toolkit.getDefaultToolkit().beep();
                                break;
                        }
                        break;
        }
        s.updateScreen();
}
```

Figure 12: implementation of menu statechart Diagram.

Inner cases in this implementation map to the inner states in the statechart diagram. In the case that the menu reaches the last element of the menu it beeps to indicates no further movement is possible.

Like the other commands in the system, Command of the Down button is an implementation of the "Command" interface. That is why we don't have a method like Down or something like that, and we have a method named execute instead. You can see the implementation of the menu in figure bellow.

Figure 13:Implementation of the menu in the insulin pump

Pump Class to Code

We need to use thread to implement concurrency in the program executions. Since we have some tasks that execute concurrently like consuming battery and injecting reservoir and also counting down times. The classes that are uses threads are pump, battery and pumptimer.

Pump class is the most important class in the system, which controls the whole behavior of the system. This class has to be a thread to control the delivery of insulin and also to control the some other parts such as reservoir, basal and bolus. This class has many functions and implements

the pump interface to provide some facility for GUI packages. In the other word, the pump class plays as a virtual machine for the GUI level (Figure 1).

```
package Controller;


import java.util.Date;
import java.util.Vector;


public interface PumpInterface {


    public int getMaxBatteryAmount();
    public void setMaxBatteryAmount(int amount);
    public void setBatteryUsageRate(int rate);
    public void setDeliveryRateForBolus(int rate);
    public void AlarmRateSeconds(int rate);
    public int getBatteryAmount();
    public int getMaxReservoirAmount();
    public void setMaxReservoirAmount(int amount);
    public int getReservoirAmount();
    public boolean isSuspended();
    public PumpState getPumpState();
    public BasalRate getBasalRate(int ith);
    public Vector<String> getPumpStatusInfo();
    public Vector<String> getBolusHistory();
    public Vector<String> getBasalReview();
    public Date getDateTime();
    public  void setBasalRate(int ith, int hour, int minute, int
rate);
```

```
    public boolean setBolus(double b);
    public void setDateTime(Date b);
    public void setBatteryVal(int val);
    public void setReservoirVal(int val);
    public void clearBolusHistory() throws Exception;
    public void clearBasalRates() throws Exception;
    public void suspendPump();
    public void resumePump();
    public void initialize()throws Exception ;
    public void startController();
    public boolean pumpFinalize();
    public int getBolusAmount();
    public int getMaxBolusAmount();

}
```

Figure 14- Pump Interface

Battery Class toCode:

Battery is a major part in a real pump and also battery is used continuously by pump to do its activities. Therefore in this simulation battery has been developed by a java thread. Importantly, battery has different usage rates to simulate real situation correctly. We change the battery status to the NOBATTERY which means the battery has finished and the pump doesn't work any more but If the tester recharge the battery by the provided slider then pump ill work properly again. Figure shows the implementation of the battery class.

```
package Controller;


public class Battery extends Thread {

```

```
private boolean turnOff;
private int charge;
private static int Rate = 1;
private CapacityStatus status;
private static int MaxBatteryAmount = 1000;
private static Battery uniqueInstance = null;


public static Battery getBattery(){
      if (uniqueInstance == null) {
            uniqueInstance = new Battery();
      }
      return uniqueInstance ;
}

public static void setBatteryUsageRate(int rate){
      Rate = 1;
      if (rate >= 1)
            Rate = rate;
      System.out.println("RATE ---->"+rate);
}
private Battery() {
      charge = MaxBatteryAmount;
      status = CapacityStatus.FULL;
      turnOff = false;
}

public int getMaxBatteryamount(){
      return MaxBatteryAmount;
```

```
}

public static void setMaxBatteryamount(int amount){
      if (amount > 0)
      {
            MaxBatteryAmount = amount;
      }else{
            MaxBatteryAmount = 1000;
      }
}

public void run(){
      try {
            charge = charge - (1*Rate);
            Thread.sleep(1000);
            adjustStatus();
            if (charge <0){
                  charge = 0;
            }
            if (!turnOff){
                  run();
            }
            else{
                  System.out.println("Battery Terminated");
            }
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
}
```

```
    @Override
    public String toString() {
          return "Battery [turnOff=" + turnOff + ", charge=" +
charge + ", rate="
                      + Rate + ", status=" + status + "]";
    }



    public void setCharge(int charge) {
          if (charge < 0)
                charge = 0;
          if (charge >= MaxBatteryAmount)
          {
                this.charge = MaxBatteryAmount;
          }
          else{
                   this.charge = charge;
          }
       adjustStatus();
    }


    public int getCharge() {
       return charge;
    }


    public CapacityStatus getStatus() {
          return status;
    }
```

```
    public int getRate() {
          return Rate;
    }

    public void setTurnOff(){
          turnOff = true;
    }
    private void adjustStatus(){
          if (charge >= 0 && charge < 0.2*MaxBatteryAmount){
                status = CapacityStatus.VERY_LOW;
          }
          if (charge >= 0.2*MaxBatteryAmount && charge <
0.4*MaxBatteryAmount){
                status = CapacityStatus.LOW;
          }
          if (charge >= 0.4*MaxBatteryAmount && charge <
0.6*MaxBatteryAmount){
                status = CapacityStatus.MID;
          }
          if (charge >= 0.6*MaxBatteryAmount && charge <
0.8*MaxBatteryAmount){
                status = CapacityStatus.HIGH;
          }
          if (charge >= 0.8*MaxBatteryAmount){
                status = CapacityStatus.FULL;
          }
    }

}
```

Figure 15- Battery Class

PumpTimer to Code

This class simulates the time and plays the clock role in the pump. Hence this class has to be a thread and run simultaneously with other parts of the pump.

Using Java Properties Class

To save and manages the configuration parameter of the pump, we used the properties class of Java. Desired properties, which we manage them, these properties are:

MaxBatteryAmount

This value shows the maximum amount, which the battery can have.

MaxResevoirAmount

This value shows the maximum tank capacity, which the reservoir can have.

BatteryUsageRate

This property indicates what is the battery usage ratio when the pump tries to deliver insulin to the patient.

DeliveryRateForBolus

In this property we can set how many seconds do we like to deliver whole bolus amount to the patient.

AlarmRateSeconds

This property has the period (in seconds) that the pump generates an alarm for the patient.

Implementing Collections

We have used Vector class to implement the compositions and the aggregations in the UML model. For example this can be seen in the

Using Java Serializable class

In order to save and to restore some classes from and to file we inherit our classes from this class. Therefore we could store and retrieve some classes in the dataprovider class easily.

Singleton Pattern

To ensure that some classes have just one object in runtime we implement this pattern for the below classes.

1) DataProvider
2) Basal
3) PumpTimer
4) Pump
5) Battery
6) BolusHistory
7) Reservoir

Exception error handling

We have used try and catch mechanism to handle the errors, which occur in runtime.

Test Plan

Black Box test

“Black-box testing is a method of software testing that tests the functionality of an application as opposed to its internal structures or workings. Specific knowledge of the application's code/internal structure and programming knowledge in general is not required. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. It uses external descriptions of the software, including specifications, requirements, and design to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure. (Ref: wikipedia)”

We have used this kind of testing in the level of unit, integration and acceptance.

Unit Test

“Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. In object-oriented programming a unit is usually a method.“

In our project almost we tested each packages separately. For this reason we defined some test driver program and we feed the driver with some data to check correctness of the package activity.

Integration Test

“Integration is the phase in software testing in which individual software modules are combined and tested as a group.”

After unit test, we integrated the packages then we tested the application again by test driver program. You can see the program in the Figure 16. This program was used for unit test and integration test.

Test Driver Program

```
import java.io.IOException;
import java.io.ObjectInputStream.GetField;
import java.sql.Time;
import java.util.*;
import Controller.*;
import Data.*;

public class main {
    private static void PrintMenu()
    {

          System.out.println("1: Set Time");
```

```
          System.out.println("2: Add Bolus");
          System.out.println("3: Add basel");
          System.out.println("4: Change Reservoir");
          System.out.println("5: Change Battery");
          System.out.println("6: Show Basal Rates");
          System.out.println("7: Show Pump Status");
          System.out.println("8: Show Bolus History");
          System.out.println("9: Suspend");
          System.out.println("10: Resume");
          System.out.println("11: Remove Bolus History");
          System.out.println("12: Clear Basal Rates");
          System.out.println("13: Terminate");

    }

    public static void main(String[] args) {

          Pump pump = Pump.getPump();
          BasalRate br = new BasalRate();
          try{

          /*   DataProvider dataProvider =
DataProvider.getDataProvider("basal.txt",
"battery.txt","reservoir.txt","bolushist.txt", "date.txt");

               Reservoir reservoir = Reservoir.getReservoir();
               reservoir.setTankCapacity(34.98);
               dataProvider.storeReservoirState(reservoir);
               reservoir.setTankCapacity(120);
               reservoir=dataProvider.retriveReservoirState();
```

```
                System.out.println(reservoir.getTankCapacity());
*/
                    int test;
                    pump.initialize();
                    pump.start();
                    Scanner in = new Scanner(System.in);

                    while(true){
                        PrintMenu();
                        test=in.nextInt();
                        if (test == 1){
                            long tt;
                            tt = in.nextLong();
                            Date dt = new Date(tt);
                            pump.setDateTime(dt);
                        }
                        if (test==2){
                            double db = in.nextDouble();
                            pump.setBolus(db);
                        }
                        if (test == 3){
                            System.out.print("Enter ith ---> ");
                            int ith = in.nextInt();
                            System.out.print("Enter Rate ---> ");

                            br.setRate(in.nextDouble());
                            System.out.print("Enter Date ---> ");
                            Date t = new Date(in.nextLong());
                            br.setTime(t);
                            pump.setBasalRate(ith, br);
```

```
                        }
                        if(test == 4){
                              double b = in.nextDouble();
                              pump.changeResorvoir(b);
                        }
                        if (test == 5){
                              int i = in.nextInt();
                              pump.changeBattery(i);
                        }
                        if(test == 6){
                              int j = 1;
                              if (pump.getBasalRate(1)==null){
                                    System.out.println("No Basal
Rate");
                              }
                              else{
                              while(pump.getBasalRate(j)!=null){

    System.out.println(pump.getBasalRate(j).toString());
                                    j++;
                              }
                                    }
                        }
                        if (test == 7)
                        {
                              Vector<String> vc =new
Vector<String>();
                              if (pump.getPumpStatusInfo()!=null){
                                    vc = pump.getPumpStatusInfo();
                              }
                              if (vc != null){
```

```
                                        System.out.println(vc.size());
                                        for(int i =0; i < vc.size();
i++){
    System.out.println(vc.elementAt(i).toString());
                                        }
                                        }
                                  else{
                                        System.out.println("No
History");
                                  }

                            }
                            if (test==8){
                                  Vector<String> vc =new
Vector<String>();
//                                  if (pump.getBolusHistory()!=null){
                                        vc = pump.getBolusHistory();
    //                              }
                                  if (vc != null){

    //System.out.println(vc.size());
                                        for(int i =0; i < vc.size();
i++){
    System.out.println(vc.elementAt(i).toString());
                                        }
                                        }
                                  else{
                                        System.out.println("No
History");
                                  }
                            }
```

```
                        if (test == 9)
                        {
                              pump.suspendPump();
                        }
                        if(test==10){
                              pump.resumePump();
                        }
                        if (test == 11){
                              pump.clearBolusHistory();
                        ;}
                        if (test == 12){
                              pump.clearBasalRates();
                        }
                        if (test == 13){
                              if (pump.pumpFinalize())
                              break;}
                    }
                  in.close();



/*             DataProvider dataProvider =
DataProvider.getDataProvider("basal.txt",
"battery.txt","reservoir.txt","bolushist.txt");


              BasalRate br1 = new BasalRate();
              BasalRate br2 = new BasalRate();
              BasalRate br3 = new BasalRate();
              br1.setTime(new Time(120000000));
              br1.setRate(12.1);
              br2.setTime(new Time(220000000));
```

```
br2.setRate(14.3);
br3.setTime(new Time(320000000));
br3.setRate(16.2);



Battery bat = new Battery();
bat.setCharge(100);
bat.start();

Basal bs = Basal.getBasal();
bs.addBasalRate(br3);
bs.addBasalRate(br2);
bs.addBasalRate(br1);
bs.exportToList();

dataProvider.storeBasal(bs);
bs.clearBasalAll();
bs.exportToList();
bs = dataProvider.retriveBasal();

bs.exportToList();
for (long  i=0; i < 100; i++)
{
      for(long j =0; j < 100; j++)
      {
            System.out.println("test");
            if (i*j ==8000)
                  bat.setCharge(10);
```

```
                }

        }


        Bolus b1 = new Bolus();
        b1.setAmount(20);
        b1.setAmountDelivered(21);
        Bolus b2 = new Bolus();
        b2.setAmount(40);
        b2.setAmountDelivered(41);
        BolusHistory bh= new BolusHistory();
        bh.addBolus(b1);
        bh.addBoluses(b2);
        bh.addBolus(b1);
        bh.addBolus(b1);

        dataProvider.storeBolusHistory(bh);
        BolusHistory bh2= new BolusHistory();
        bh2 = dataProvider.retriveBolusHistory();
        bh2.exportToList();



    /*  Battery battery = new Battery();
        battery.setCharge(110);
        dataProvider.storeBatteryState(battery);
        battery = dataProvider.retriveBatteryState();
        System.out.println(battery.getCharge());
```

```
                */



            }
            catch(Exception exception){
                  System.err.println(exception.toString());
                  pump.pumpFinalize();
            }



      }
}
```

Figure 16- Test Driver Program

A Typical Feature of system for testing

We tried to use some slider in the interface to facilitate testing the pump while intentionally decreasing to increasing the battery and the reservoir content. Using these slide bar, user can increase or decrease the battery level or he/she can increase or decrease the reservoir content and get the reaction of the systems to these events.

When user, for example push up the battery slidebar, the Pump class in the controller package is informed by the user interface. Then the appropriate reaction to this change is calculated and generated by the pump. This reaction will be reflected by the userinterface to the user. As an example if

the battery levels comes down from an special amount, say 20 percent, Pump goes to an especial status and this is reflected on the pump interface by showing an special indicator on the screen. Figure bellow:

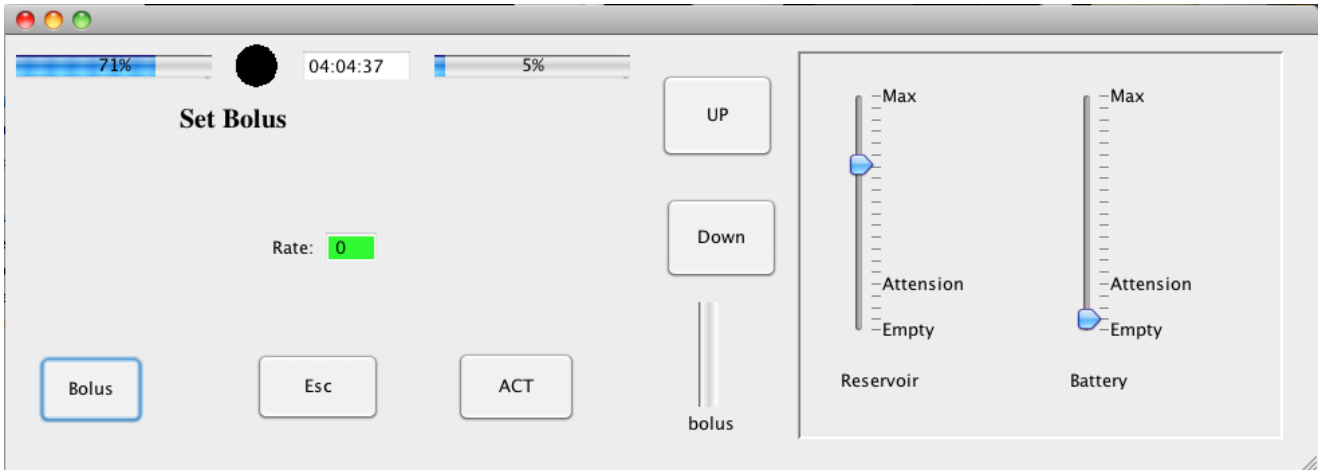


Figure 17: Slidebars to change the values of reservoir and battery

Sample Test cases

We ignored the whole scenario of the test cases since the steps are trivial therefore just we've mentioned the conditions and the results.

Unit test, test cases level

Test Case 1: Store and retrieve of Battery information

SCENARIO: In the test driver program we constructed an object of Battery then we set the battery level to 110 then we stored them and after that we retrieved the object of the battery from the file and we printed the information (see below).

```
Battery battery = new Battery();
battery.setCharge(110);
dataProvider.storeBatteryState(battery);
battery = dataProvider.retriveBatteryState();
System.out.println(battery.getCharge());
```

Test Case 2: Store and retrieve of bolus history information

SCENARIO: In the test driver program we constructed an object of Bolus history and also we constructed two bolus object and delivered insulin to the patient by them then add them to the bolus history. Afterward we stored the bolus history by dataprovider object and after that we retrieved the object of the bolus history from the file and we printed the information.

```
Bolus b1 = new Bolus();
b1.setAmount(20);
b1.setAmountDelivered(21);
Bolus b2 = new Bolus();
b2.setAmount(40);
b2.setAmountDelivered(41);
BolusHistory bh= new BolusHistory();
bh.addBolus(b1);
bh.addBoluses(b2);
bh.addBolus(b1);
bh.addBolus(b1);
```

```
dataProvider.storeBolusHistory(bh);
BolusHistory bh2= new BolusHistory();
bh2 = dataProvider.retriveBolusHistory();
bh2.exportToList();
```

Acceptance test, test cases level

Test Case 1: Change Battery Level

SCENARIO: Tester change battery level to ZERO level to test the behavior of software and check the stability of system.

RESULT: Software produced some EXCEPTION.

Test Case 2: Test battery usage ratio

SCENARIO: Tester use bolus key and set the bolus rate to 10 units and compare the battery usage per minute to the past.

RESULT: There isn't any change in the usage ratio.

Test Case 3: Display Bolus History

SCENARIO: Tester push the ACT button then selects the bolus history menu.

RESULT: The bolus history was displayed successfully without any problem.