SoSafe Home Security Systems

COEN359 – Design Patterns Project Documentation

-By
He Shouchun (Kenny)
Sakshi Bhatia

Instructor:
Prof. Rani Mikkilineni
Santa Clara University

Table of Contents

1. Project Description

1.1 Definition

SoSafe Security systems provide security to buildings in terms of Door Sensors and Temperature Sensors.

Door Sensors trigger an alarm event whenever a door open or close event occurs.

Temperature Sensors trigger an alarm event when room temperature goes above a threshold value (For our implementation purpose, Threshold is 100 F) or return to the normal level.

Functionality of our implementation can be explained as below:

The system administrator should be possible to launch the GUI console to create a new user profile, or login with the UID and password of an existing profile.

During the user profile creating, the system administrator should be possible to enter details such as UID, password, type of user and a map of the environment to be monitored. With these basic details a new user profile will be created.

After the new profile has been created or a user login to the system, the system will show the map to allow the system administrator to create, view status, disable and re-enable the sensors on the map by simple mouse clicks. And the administrator should be possible to generate the bill and view the past bills..

The system uses an event generator to randomly generate simulated events of door open/close and temperature change. Each time there is a change in door state, which is door open (Break-in) or door closed, the corresponding sensor generates an Alarm event captured in Alarm Event manager. Similarly, when temperature rises above the critical threshold value or drops from critical to normal temperature, an event is generated by the temperature sensor. Alarm Event manager who is observer on the sensor handles this event and notifies user of the alarm. In our implementation user can login through GUI console or remote login to clients. On clients we send text based notification of all the alarm events where as on GUI we notify by changing the icon for door sensor and temperature sensor.

The GUI console also start a server in the background to handle the login request from the remote clients. Users should be possible to remotely connect to this server through TCP port 12345. Once the user login failed, the remote client will be disconnected. If users login successfully with the correct UID and password from a remote client, the system should keep the connection and send the alarm events to the remote client and the later will display it on its own UI. User can also view the current bill information through the remote client when the system administrator creates a new bill.

We have also taken into consideration the type of user. This defines if our user is normal/Premium or VIP. Depending on the type, security can be increased and also, billing rates and strategy is dependent on user type.

Bill has two components to it – maintenance and usage part and alarm event part. Maintenance part in bill calculation is the security system installation and monthly maintenance charges calculated after the last bill generation. Variable part is dependent on the events that have occurred since last bill generation. Services provided due to an event occurrence are taken into consideration while bill calculation.

1.2 Scope

1. Multiple user profiles can be created and maintained.
2. Different user types supported – Normal, Premium, and VIP. Type of user helps decide the security level and billing strategy.
3. Two types of sensor supported – Door Sensor and Temperature sensor. Design as well as code is open for extension in future.
4. The system has a GUI console for system administrator to create users, login and create sensors, enable/disable sensors, view sensor statuses, as well as generate and display bills.
5. The system offer client for user to login and be notified about the alarm events.

1.3 Assumptions

1. For each user the map can be defined by any given map file.
2. User has flexibility to decide where to install which type of sensor (Door Sensor/ Temperature Sensor) on the GUI.

1.4 Limitations

Below mentioned are the limitations of current implementation. However design has been made flexible to include these implementations in future.

1. The system can only support 2D map, and it cannot adapt to the complicated environment such as 3D.
2. The system can only handle the simple sensors which implement the current Sensor abstract class. It cannot handle the more complicated or advanced sensors, such as flood sensor or earthquake meter.

2. Design Methodology and Tools

2.1 UML Tools

StarUML – Used for class diagrams.

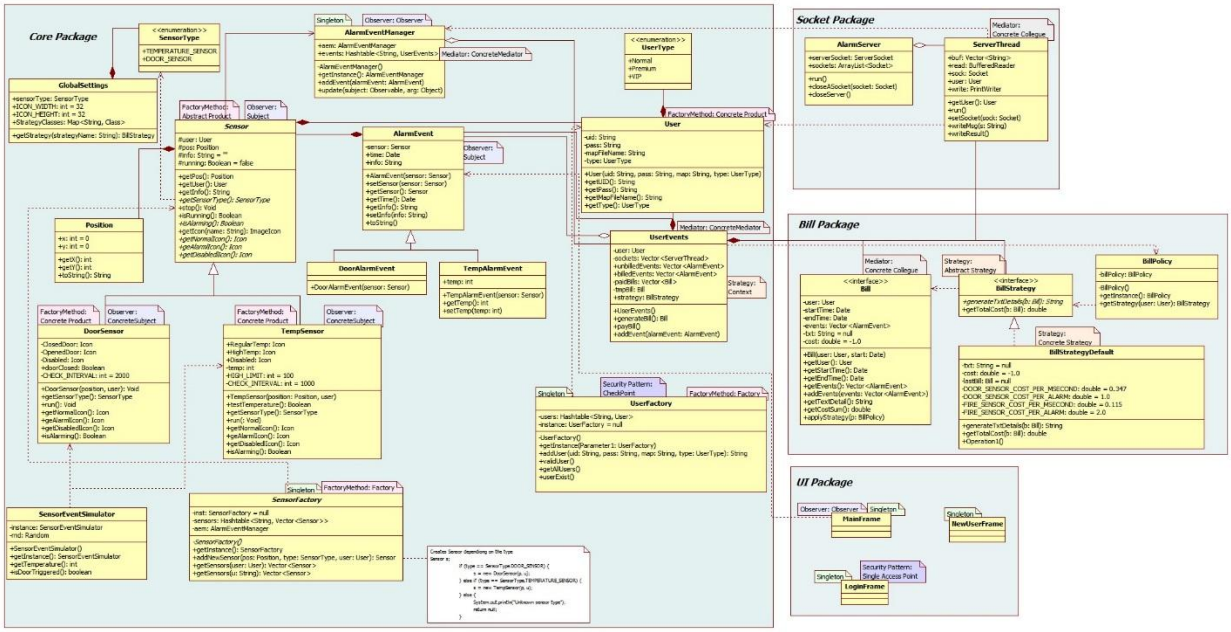
2.2 Implementation Language

Java, Java Swing.

2.3 IDE

Eclipse

3. Class Diagram



UML file for the design – Also submitted in deliverables.

SoSafe Security System.uml

4. Discussion on Patterns

Following patterns have been used in the design of this project:

4.1 Observer Pattern

Used in multiple scenarios where a class needed to do its specific tasks depending on the state change of another class objects. Below is detailed discussion of these scenarios:

4.1.1 Sensor and Alarm Event Manager

We have an abstract class Sensor which is extended by DoorSensor and TempSensor. A DoorSensor activates an Alarm whenever there is a break-in (Door Open in case of Door Sensor) or Door closed. A TempSensor activates alarm in case of high temperature (above critical threshold) and when temperature drops from critical to normal. This requirement itself brings in the need to have a class (related to Alarm) that keeps observing Sensor (Observable or Subject) for any break-in/Fire event. All the alarm events are stored in a hash map of user events in AlarmEventManager class.

We have implemented this requirement using GOF Observer pattern. We have class Sensor as subject, class DoorSensor and class TempSensor as concrete subject and class AlarmEventManager as Observer.

The code inside the DoorSensor (subject) class :

```
	public void run() {
		running = true;
		// Check the door sensor periodically, and trigger the event when
necessary.
		while (running) {
			// Sleep an interval
			try {
				Thread.sleep(CHECK_INTERVAL);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}

			if (SensorEventSimulator.getInstance().isDoorTriggered()) {
				doorClosed = ! doorClosed; // Open->Close, or Close->Open

				DoorAlarmEvent dae = new DoorAlarmEvent(this);
				if (doorClosed) {
					dae.setInfo("The door is closed.");
				} else {
					dae.setInfo("The door is opened.");
				}
			    setChanged();
			    notifyObservers(dae);
			}
		}
```

```
}
```

The code in the AlarmEventManager (Observer) class:

```
public void update(Observable o, Object arg) {
	if (arg instanceof AlarmEvent) {
		AlarmEvent ae = (AlarmEvent) arg;
		addEvent(ae);
	}
}
```

4.1.2 Main Frame UI and Sensor

Similar to AlarmEventManager, MainFrame (UI class) keeps on observing Sensor for any alarm event (Door Open/ Door closed or High Temperature/ Normal Temperature) Depending on the event, a corresponding image is displayed on the GUI.

Here also we have used GOF Observer pattern with class Sensor as subject, class DoorSensor and class TempSensor as concrete subject and class MainFrame as Observer.

Below code illustrates the implementation of the pattern.

The code in the MainFrame (UI Observer) class:

```
public void update(Observable arg0, Object arg1) {
	Sensor s = (Sensor) arg0;
	JButton b = buttons.get(s);
	if (s.isRunning()) {
		if (s.isAlarming())
			b.setIcon(s.getAlarmIcon());
		else
			b.setIcon(s.getNormalIcon());
	} else {
		b.setIcon(s.getDisabledIcon());
	}
}
```

4.2 Strategy Pattern

As discussed earlier, we have different billing rates and policies depending on the type of user (in turn, level of security provided.) Thus, while creating bill, we need to consider User type and then apply an appropriate bill strategy. This requirement itself brings in a need to have strategy pattern implemented for this scenario. For now, we have just implemented default billing strategy however; design and code have been done in a way to add more strategies at later point in time.

Billing strategies have been implemented using GOF Strategy design pattern where interface BillStrategy is an Abstract strategy, class BillStrategyDefault is a concrete strategy and UserEvents is the context.

Below code illustrates the BillStrategy interface and a concrete strategy class BillStrategyDefault:

```
public interface BillStrategy {
	String generateTxtDetails(Bill b);
	double getTotalCost(Bill b);
}
public class BillStrategyDefault implements BillStrategy{
   private String txt = null;
   private double cost = -1.0;
   private Bill lastBill = null;
   private static double DOOR_SENSOR_COST_PER_MSECOND = 30000000.0/(24*60*60*1000);
   private static double DOOR_SENSOR_COST_PER_ALARM = 1.0;
   private static double FIRE_SENSOR_COST_PER_MSECOND = 10000000.0 / (24*60*60*1000);
   private static double FIRE_SENSOR_COST_PER_ALARM = 2.0;
   public String generateTxtDetails(Bill b) {   …   }
   public double getTotalCost(Bill b) {   …   }
}
```

Below code illustrates the how to apply the BillStrategy interface in the context:

```
public synchronized Bill getBill() {
		// Generate the bill with the billing strategy.
		bill = new Bill(user, lastBillDate);
		lastBillDate = Calendar.getInstance().getTime();
		bill.addEvents(unBilledEvents);
		bill.applyStrategy(strategy);
```

4.3 Mediator Pattern

Mediator pattern is required for loose coupling of a lot of interacting classes. We have used mediator in following scenarios for our implementation.

4.3.1 UserEvents Mediator

As mentioned earlier, we consider the alarm events to calculate bill which constitutes the variable part of bill. We have a class AlarmEventManager which is an Observer on Sensors. It captures all the AlarmEvents and dispatch them to the UserEvents object of current user. Then it organizes these events per user and then passes it to the Bill for bill generation for that specific user and to the ServerThread to dispatch message to all remote clients. Thus, it is clear that this class UserEvents works as a Mediator.

We have used slight variation to the GOF Mediator design pattern to implement this scenario where we DO NOT have abstract mediator. UserEvents is a concrete Mediator class and Bill as well as ServerThread are concrete classes. Also, UserEvents is an Observer on AlarmEvents which is a subject.

Below code illustrates the implementation of pattern.

```
Code in the UserEvents (Mediator) class:
public class UserEvents {
```

```
	private User user;
	private Vector<ServerThread> sockets = new Vector<ServerThread>(); // For
synchronization. No ArrayList.
	private Vector<AlarmEvent> unBilledEvents = new Vector<AlarmEvent>();
	private Vector<AlarmEvent> billedEvents = new Vector<AlarmEvent>();
	private Vector<Bill> bills= new Vector<Bill>();
	private Date lastBillDate = null;
	private Bill bill = null;  // A latest bill.
	BillStrategy strategy;


private synchronized void dispatchMsgToSocket(String msg) {
	for (ServerThread st: sockets) {
		st.writeMsg(msg);
	}
}
```

(There are more other methods for mediating the behaviors when get the new alarm events.

4.4 Factory and Factory Method Pattern

4.4.1 SensorFactory

As stated earlier, Sensor is an abstract class extended by DoorSensor and TempSensor. To create a Sensor Object, we have to use the FactoryMethod pattern.

We have designed and implemented the FactoryMethod with a slight variation to GOF factory method. We have one Factory that dynamically checks the type of object to be created and creates a corresponding class object.

To explain further, we ask user to tell which type of sensor to create (information passed through UI). If user wants Door Sensor, we create an object of DoorSensor else in case type passed is TEMPERATURE_SENSOR, we create an object of TempSensor class.

Also, we have implemented the Factory Method together with Singleton pattern in the SensorFactory. to make sure there is only one instance/entrance for creating the other Sensor objects. .

Below code illustrates the implementation of the FactoryMethod pattern in the SensorFactory class:

```
	public synchronized Sensor addNewSensor(Position p, SensorType type, User u) {
		String uid = u.getUID();

		// Get the Sensor group for the current user.
	..... (init code)
		Sensor s;
		if (type == SensorType.DOOR_SENSOR) {
			s = new DoorSensor(p, u);
		} else if (type == SensorType.TEMPERATURE_SENSOR) {
			s = new TempSensor(p, u);
		} else {
```

```
                System.out.println("Unknown sensor type");
                return null;
        }
        ..............
}
```

4.4.2 UserFactory

A user is created using UserFactory as we needed a mechanism to validate user information before giving access.

Below code illustrates the implementation of the FactoryMethod pattern in UserFactory class:

```
public String addUser(String uid, String pass, String map, UserType type) {
        // If the UID exists, simply return error message.
        if (users.containsKey(uid)) {
                return "User ID already exists";
        }

        users.put(uid, new User(uid, pass, map, type));
        return null;
}
```

4.5 Singleton Pattern

Singleton pattern has been used to create objects for below classes:

1. AlarmEventManger
2. BillStrategy
3. SensorFactory
4. UserFactory
5. All UI frames
6. EventSimulator

All these classes have either a single instance or a very few instance requirement.

4.6 Security Pattern – Future Implementation

We have used some of the security patterns in our design and for some we are proposing how it can be incorporated. Below is the discussion of security patterns with respect to our design and implementation.

4.6.1 Single Access Point

Login Frame in our system works as Single entry point. This is the form where user logins to access its profile or creates a new profile in case of new user.

Login frame is implemented using Singleton design pattern.

4.6.2 Check Point

UserFactory acts as a check point. Once user enters a username/password in the Login frame (SAP), validUser method in UserFactory validates if the user credentials are correct or not. Only an authorized user is allowed to access user profile.

UserFactory is implemented using Singleton and Factory method design patterns.

4.6.3 Role

SoSafe system application can be used by end user to see profile details or by SoSafe system administrator to manage user profile, generate bills etc. GUI console is used by a user with role system administrator and remote login client is used by end user.

We have identified this scenario to be implemented using Role security pattern.

4.6.4 Session

Each user can have multiple sessions by logging in through different devices. These sessions are maintained through sockets and thread. Each client creates a new socket by registering to the default port 12345. All the UserEvents are buffered and dispatched to the socket which then broadcasts to all the registered clients.

4.6.5 Full view with Errors/ Limited View with errors

As explained in previous sections system administrator has more control over the user profile as it helps add/remove/disable/enable sensors and also shows current bills and all past bills.

End user, on the other end has limited view where only alarm events are showed and current bill is accessible.

4.7 Global Settings Class

In our implementation we have defined a GlobalSettings class that defines global constants to be used. The idea is to decouple dynamic features from the concrete implementation. This helps in easy extension without opening any other class.

We have implemented the concept for bill strategies. Define all possible bill strategies in GlobalSettings class. For now we have defined default strategy. In future if any new strategy is added, only a new class will be made and a new strategy will be added in GlobalSettings class. No other class needs to be changed.

```
static private Map<String, Class> billStrategy = new HashMap<String, Class>();
	// If there are new policies, add them into the map so the system can
automatically load them when start.
	static {
		billStrategy.put("Default Strategy", BillStrategyDefault.class);
```

```
};
static public BillStrategy getStrategy(String strategyName) {
	BillStrategy bp = null;
	try {
		bp = (BillStrategy)(billStrategy.get(strategyName).newInstance());
	} catch (InstantiationException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	} catch (IllegalAccessException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}

	if (null == bp)
		bp = new BillStrategyDefault();
	return bp;
}
```

5. Source Code

Submitted as deliverables as well.


AlarmClient.zip


AlarmSystem.zip

6. Possible Commercial Usage

The code of this project can be used in commericial environment with only very minor changes:

When creating sensors, the alarm sources can be got from the real Zigbee sensors insteard of calling the method of sensor events emulator.

The remote client can ring alarms when the sensor reveives the alarm, and can change to be silent when the sensor return back to normal state.

Acknowledgement

Thanks professor Rani for offering the lectures about the Design Pattern course. All the patterns and big rules used in the system design of this project reflect the ideas she taught us in the Design Patterns classes.