University of Waterloo
ECE 355 - Software Engineering

Software Design Document

G3 Security System

Group 3
Ayush Gupta, 20292953
Ankit Srivastava, 20298545
Usman Khan, 20266360

Table of Contents

List of Figures

List of Tables

Introduction

Executive Summary

This document contains the software design of the G3 Security System. The document is based on the requirements elicitation, analysis and object modeling that was performed in the Software Requirements Specifications [1]. Some of the diagram and information presents in the software requirements specification is further refined in this document. This refinement is to allow developers that will be implementing the system greater insight into the design of the system. This software design document relates to the design and implementation of a home monitoring and security system. The system is designed to use a wide range of sensors to monitor a property and alert the owners of events such as break-ins, fires, floods, etc.

The goal of this document is to describe the overall design of the system. The document begins with first describing the architecture to decompose the system into subsystems. It discusses how these subsystems interface with both internal and external components. The document then describes various aspects of system design. This such as access control and security, boundary conditions and data resource management are discussed in detail in this section. The document then moves onto describing the external and internal interfaces of the system. A large portion of this document is dedicated to the object design. This section contains information about the design patterns employed to increase code re-use. The algorithms used to solve computational problems. This section also deals with how the objects that comprise the system interact with each other. The design evaluation section analyzes the design choice of the previous sections with regards to things such as trade-offs, re-use and optimizations. Finally, the document finishes with a discussion of the environment in which the system will be developed and then operate. Synchronization and fault handling are also discussed in this section.

Purpose

The purpose of this document is to provide a written description of the G3 Security System. The description contained within this software design document is intended to be used by the software developers who will be implementing this system. This document will be used by development team to ensure all members have the same vision for the overall project. The development team will rely heavily on this software design document during the implementation phase of the project.

This software design document contains several sections intended to cover all relevant aspects of the design of the system. The following document is separated into several sections, which each section addressing an important software design issues. The subsequent sections are, System Design, Interfaces, Object Design, Design Evaluation and Operating Environment. Together these sections completely describe the design of the G3 Security System.

Design Evaluation and Operating Environment. Together these sections completely describe the design of the G3 Security System.

Scope

The scope of this document does not differ greatly from the scope described in the earlier software requirements specification. The fact remains that the team has decided to focus on a subset of the overall problem to better tackle the problems that they encounter. Like the software requirements specification, this software design document deals specifically with the design of the security aspect of the system and puts less emphasis on the environmental and family monitoring aspects. However, as described in the earlier document, extensibility has been made a key design goal to make adding additional features an easy and rapid process. The generality with which certain concepts or the system are modeled with allow it to adapt easily to changes. The developers will be able to easily add additional functionality in future iterations of this system.

Assumptions

This software design document focuses on the software aspects of the G3 Security System. Certain aspects of the system such as the central control unit which is responsible for receiving and sending signals to and from the sensors are not modeled with as much detail. It is assumed that systems capable of doing this already exist and are readily available and don't need to be included in the software design. Similarly, things such as the inner workings of the various types of supported sensors are not detailed in this document and it is assumed that they are able to communicate with the central control panel which then interfaces with the software. For sections that deal with these externally available systems, a greater emphasis is put on the interface the software needs to communicate with these systems. Thus this software design document focuses on the main controller software and its communication with the relevant sensors and hardware and less on the inner working of the hardware itself.

Design Goals

The following is a prioritized list of the strategic aims for the functionality of the G3 Security System. The list covers design goals from the perspectives of all three relevant stake holders: client, end user and developer.

1. Security: The system will continue to function correctly even when being tampered with by a malicious user.
2. Extensibility: The system should be able to easily accommodate additional sensors and sensor types.
3. Maintenance cost: The system should be inexpensive to maintain, requiring minimal time, effort and money to continue functioning.
4. Modifiability: Behaviour/handling of sensors should be modifiable without the need for large changes to the entire system.
5. Portability: The system should be able to work with hardware from a wide range of suppliers.
6. Usability: Both the smartphone app and the control panel should be easy to use for the end user, allowing them to accomplish necessary tasks with minimal number of steps.
7. Availability: The system should be as close as possible to 100% up-time.
8. Fault tolerance: The system should be able to tolerate conditions, such as loss of primary power, loss of telephone lines, loss of sensors, etc.

Prioritization of Functionality

Priority	Functionality	Description
Critical	Arming/disarming the system	User should be able to arm and disarm the system
Critical	Sensor event detection	System should be able to interface with and subsequently detect sensor events
Critical	Smartphone app	User should be able to arm/disarm the system through their smartphone and receives alerts on their smartphone as well
Critical	Alarm functionality with appropriate delay	System is able to trigger an alarm once an sufficient event is detected, with the appropriate delay included between the event and the alarm trigger
High	Touch panel configuration	User should be able to configure the system using a touch panel
High	Redundancy	Should be resistant to attempts to tamper with the system
Medium	Deal gracefully with low batteries	System should be able to detect and deal with components that are running low on batteries
Medium	Provides secure way for user to authenticate themselves	The user is able to securely authenticate themselves on the smartphone app to allow communication with the main system
Medium	Green energy management	System should be able to monitor things such as environmental conditions within the home and control appliances
Medium	Family safety	The system should be able to family to ensure their safety via sensors such as GPS bracelets and geo-fences
Low	Billing system	System should be able to interface to an external billing system

Terminology and Definitions

Define any uncommon terms used in this document.

Access Level: A numerical value associated with the privileges afforded to a user.

APK: A variant of the JAR file format, used by Android to distribute and install applications on Android devices.

DLL: A Microsoft shared library

Fatal crash: A severe fault usually resulting in the application being force to shutdown or restart.

IDE: Integrated development environment

JAR: Java archive - an archive file format used by the Java platform.

Schema: A blueprint for the structure of a database

SDK: Software development kit

Zone: A user defined collection of sensors. This can comprise anything from a single room in a home, to all the outside sensors to even the entire home itself.

References

[1] Ayush Gupta, Ankit Srivastava, and Usman Khan, Software Requirements Specification, 2012.

Architecture

The system comprises of three major components, the central controller, the sensors and the user interface. Both the user interface and the sensors communicate to the central controller. Thus, the controller is the main component of this project and thus has been divided into two sub components, one dealing with the sensors and the other with the user interface.

Overview

As mentioned, the controller has been divided into 2 subcomponents. Thus, we have chosen 2 different architectural styles to implement the communication of the controller with the sensors and the user interface.

Controller and Sensors:

To implement the communication between the controller and sensors we are modelling this as a modification of the client server model architecture. In client server architecture, there is one or more servers, which provide requested services to the clients, and there are multiple clients communicating with the server(s) to request the services. In or case the sensors will be modelled as the clients as they will be controlled by the controller which would act as the server.

In a normal client server architecture the client is the front end which deals with the user and the server is the back end and is invoked only on the request of the client to process some data or information.

In our modelling, the sensors periodically keep in contact with the controller telling their current status, example battery and sensor values. The sensors once they sense a motion or get activated will immediately communicate with the controller their state and then the server can request more data from that sensor, like video/images from a camera. These can be processed by the server to detect if it's an actual intrusion or a false alarm. This would enable the server to keep monitoring and giving high priority to other sensors in the same zone and process their signals first. The modification to the general client server architecture in our case is that the controller could also make a connection with the sensor to switch it on or off. The following figure shows the relationship between the sensors and the controller.

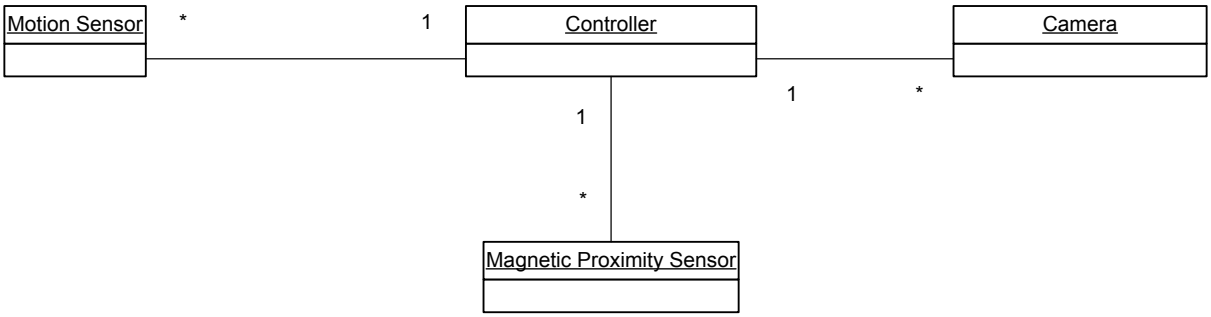

Figure 1: Relation of Sensors and Controller

This system will have a low coupling among the 2 subsystems. The reason to that is that the sensor will just send the signals to the controller and won't care how the controller interprets those signals and what actions do the controller performs. The sensors need no knowledge of the underlying hardware/software of the controller. Similarly, the controller doesn't need to be worried about how the sensor gets activated but has to deal only with the data coming from the sensors. The interface for this data is a standardized interface and would be known to both the controller and the sensors and that would make the basis of communication between them.

This style also helps in usability as there will be multiple usage of a single kind of sensor in one zone and all zones combined. Thus, each instance of that sensor will send similar information to the controller and thus, the different kinds of requests that the controller needs to be handled will be very limited.

This style helps in having a reliable solution as due to constant communication it will be easy for the controller to know if a sensor is not responding and that can be termed as an error in the system. The owner could be then informed to check that sensor and take the necessary to make it work.

As in a client server design, to extend the system you can add another client to the system or add a new service to the server. Similarly, in case of addition of a new zone or a sensor within a zone, would be similar to adding a new client to the whole system and the controller will deal with it in the same manner. The fact that adding these components won't change the controller also signifies the low coupling between these two subsystems. In case, in future if there is a new kind of sensor that seems feasible to add to the system, then it would be a matter of creating a new type of request in the controller without modifying any other components or classes.

Moreover, as the client server architecture is well documented, using it would lead to fast development of the systems.

Controller and User Interface:

To implement the controller and deal with the user interface, the Model/View/Controller (MVC) architectural styles suits this system the best as it effectively deals with the user interaction, changes due to those user interactions and then updating the display for the user to view it.

A MVC architectural style is decomposed into three components:

- Controller subsystem: This lets the user interact with itself and based on the interaction or user events, notifies the model what action needs to be taken.
- Model subsystem: This subsystem handles data and events in the application domain. This is controlled by the controller which sends appropriate signals/commands to the model to perform a particular action.
- View subsystem: This is the subsystem that displays information to the user. This displays the changes that have been made to the model subsystem

This helps to simplify the architecture as it decouples the models and views and let the controller take care of all the actions and processing.

For this subsystem, this is a very a good architectural style as the controller will be modelled as the model subsystem, the user interface and interaction will be modelled as the controller subsystem and the screen will be the view subsystem. The user would enter information like activating/deactivating a sensor/zone or requesting to view the percentage of the alarmed portion of the house. This would lead to the main controller sending signals to turn on/off the sensors via the modified client server architecture mentioned above or calculate the percentage of house alarmed using the stored last state of the sensors. This information would then be displayed on the screen of the sensor.

The following diagram how the MVC design for our system. Here the user interface perform the functionality of interacting with the user, thus, make the controller subsystem. The model subsystem consists of the the controller of our system. The display is the view subsystem.

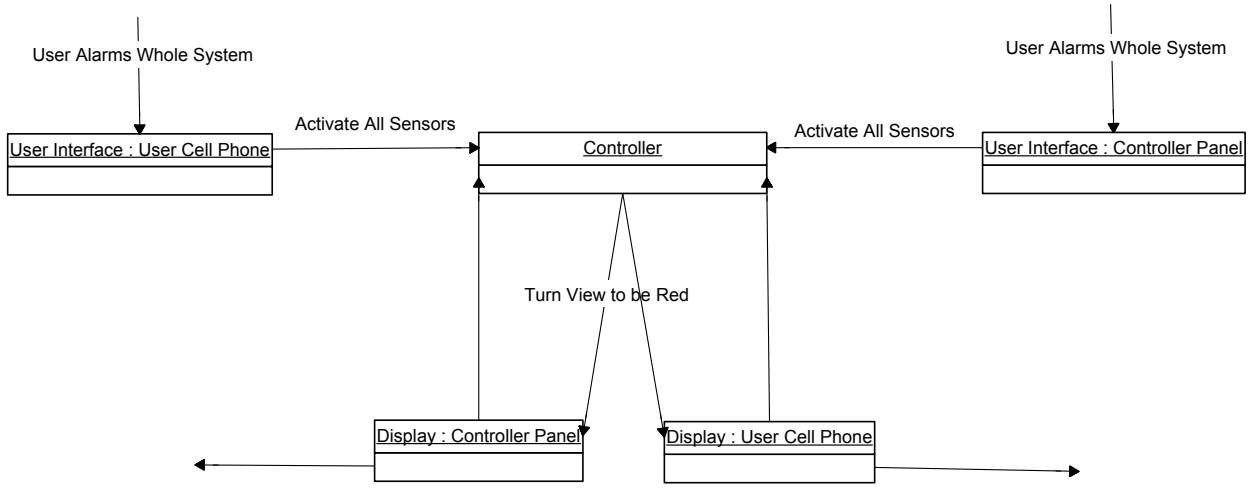

Figure 2: MVC Diagram of the subsystem

As MVC has been designed to create low coupling between the individual subsystems, the result subsystem has a low coupling to it.

The MVC is a well-known design implementation, thus, this will help in easy and rapid implementation of the design. For usability, it will provide a clear/concise user interface to the user, hiding all the details of the controller, thus making it an easy to use system. If in future, this device supports another kind of interface that could be easily modelled as part of the views and controller subsystems and thus be easily extended without changing the implementation of the controller.

Controller:

As mentioned above the controller has been divided into two parts, one dealing with the sensors and the other with the user interfaces. To patch the controller together we would use an adapter design pattern which would be explained in detail in the later sections of the document.

Subsystem Decomposition

As mentioned above, the whole system has been decomposed into 3 major subsystems, the central controller, the sensors and the user interface.

The following diagram is the component diagram of the overall system. It contains the three subsystems described and their interfaces.

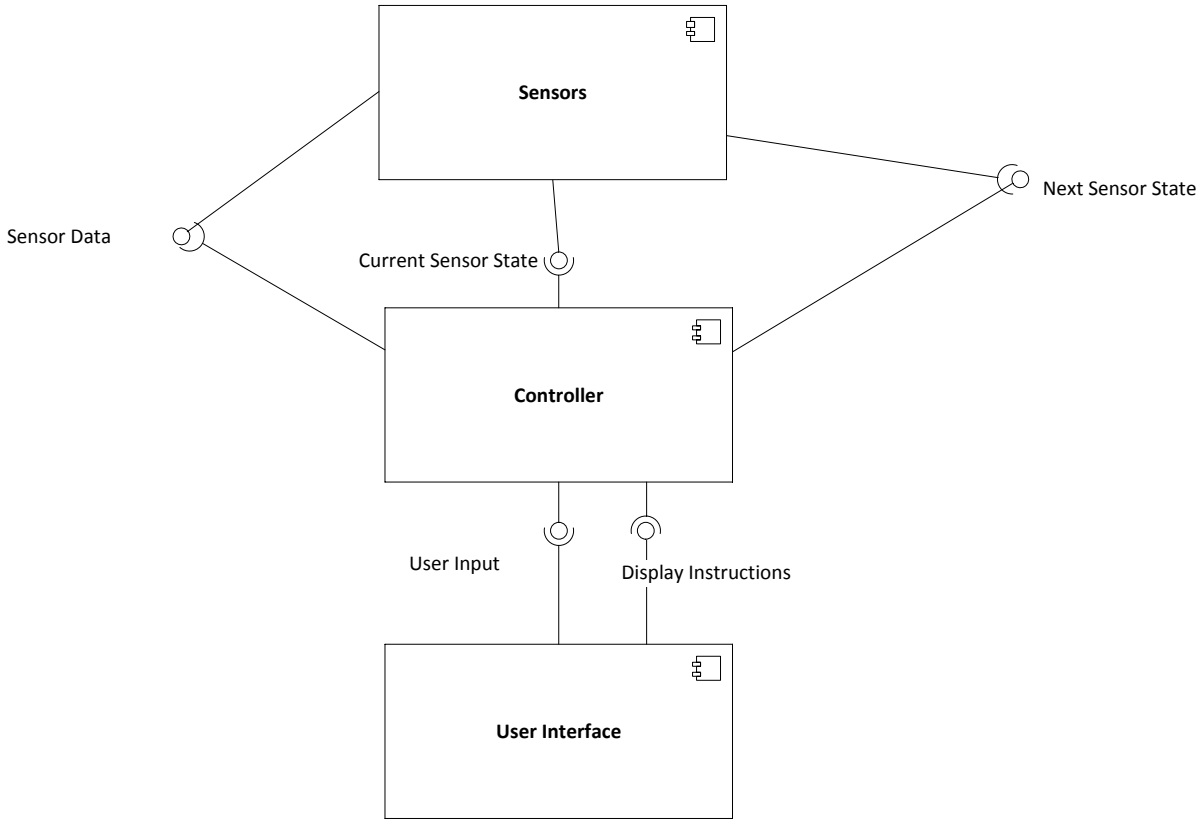

Figure 3: Component Diagram

The following figure provides the complete deployment diagram for the system. The major hardware parts are the phone, control panel and the sensors themselves.

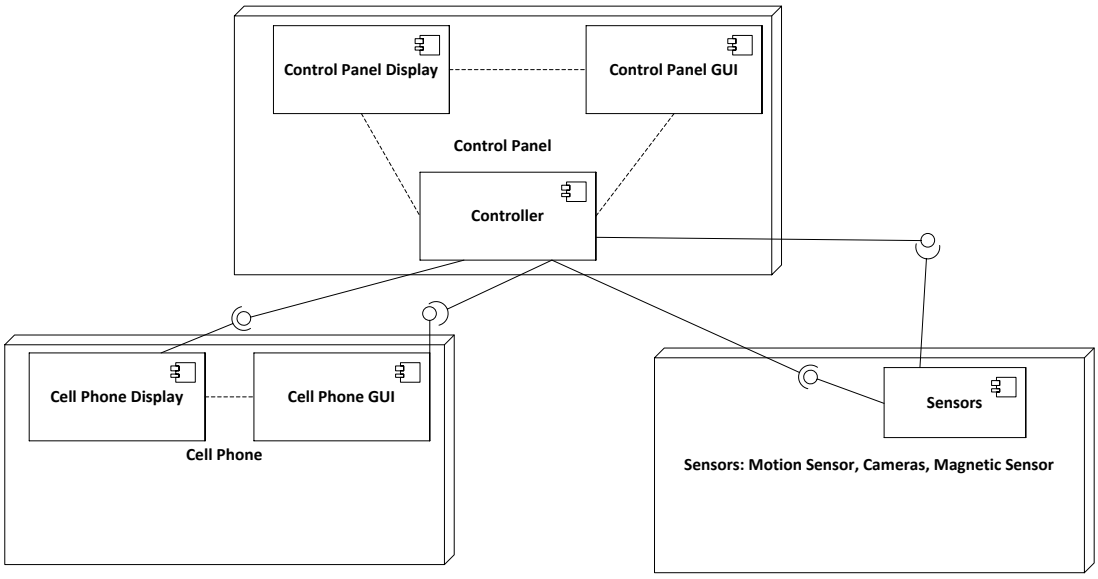

Figure 4: Deployment Diagram

The Sensor component provides the sensor data and the current sensor state to the controller. It is supposed to evaluate the intrusion and based on that give the information to the controller.

The User Interface component exists in two places, the cell phone and the controller panel. The main functionality of this is to provide the user, the owner and his family members, an interface to control the level of security and view how much portion of the house is secured and any other important information from the controller. Both the controller panel and the cell phone have the same interface architecture to them so that it is easy for the user to move from one to another.

The controller is the brains of the whole system which communicated with both the sensor and user interface components. On the sensor side, it deals with the information coming in from the sensor and decides if it is indeed an intrusion or a situation worthy to alert the owner. It also periodically checks the state of the sensor to make sure it is active. It also controls the sensor power state when the owner modifies the level of security in the home. On the user interface, it deals with all the user interactions. It evaluates the user inputs and depending on it modifies the change of the sensors or updates up the view for the user on the display.

System Design

Hardware/Software Mapping

As discussed in some of the other sections of this document the system has been broken up into multiple threads of execution. This means that certain aspects of the system are able to execute concurrently. In hardware terms, the majority of the system logic is located within the main system control software that runs on a Windows or Linux machine. This machine is the main control center of the system and coordinates all activities and tasks from this point. One very important hardware component is the central control unit. This unit is responsible for bridging the gap between the logical representation of the system as a set of objects and the physical signals that are required for the system to function. Additional, the interfaces with which the user will interact with have been assigned their own devices for the most part. Almost all the interactions of the user with the system will take place through two means. Either through the touch panel located within the home of the user, or through the smartphone app. The Synchronization subsection in the Operating Environment section discusses the need for coordination between the smartphone and the main system controller. These two nodes need to communicate in order to allow the user to view and modify the state of the system through their smartphone. These two nodes of the system use the concept of messages and commands to efficiently communicate with each other.

Data Resource Management

Information about users and their respective access levels are persistent data that needs to be stored in a database for the system to be able to function. The user table and related access levels tables will be stored in database tables according to the schema show below in Figure 5. The use of access levels in elaborated on in the following section.

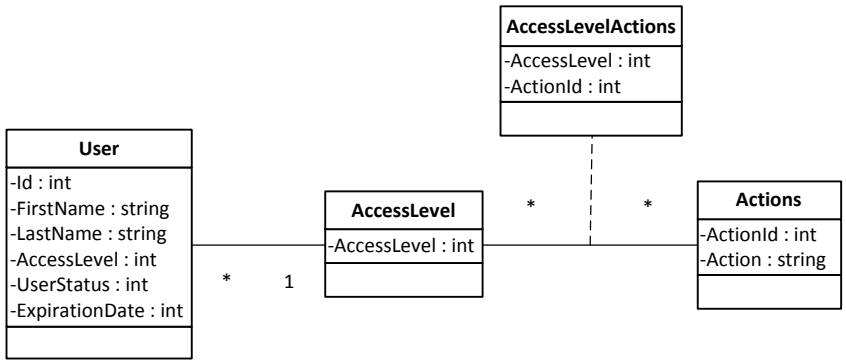

Figure 5. Selected database schema represented using UML class diagram

The system will also utilize a simple table to safely store system configuration parameters. These parameters, are values that would usually be stored in plain text files on the system itself, however in order to make the system less prone to interference from malicious users, the system will use the safety features built into a database to safely store these parameters. Storing these parameters in the table also allows a user with the sufficient access level to modify them. The types of parameters that the table is able to store are very open ended with the casting of the parameter value to the appropriate data type being taken care of by the application code.

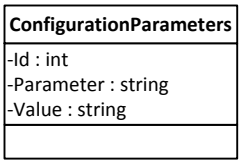

Figure 6. UML diagram of ConfigurationParameters table

Part of the system start-up process involves instantiating all the zones and sensors and adding those sensors to their respective zones. Figure 7below shows the structure of the tables used to store the zone and sensor data. At system start-up the system reads the values from the tables and creates the appropriate system structure based on the table data. The data in these tables also allows the system to recover from possible fatal crashes by reading in data from these tables and restoring the system structure to what it was before the crash. If these tables were not present, all sensor and zone association data would be lost in the event of a system shutdown or fatal crash.

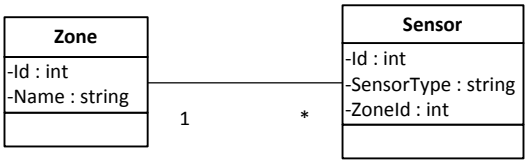

Figure 7. UML diagram of Zone and Sensor tables

The use of the database to store persistent data also allows the system to easily implement redundancy through the use of nightly backups. Incremental backups will be created of the entire database, allowing the system to be easily restored to a previous state if necessary. The nightly backups also allow the system to recover from fatal crashes which might have resulted in data loss.

Access Control and Security

Access control to the system will be managed through pre-defined access levels. Each system user will be assigned an access level which will govern the actions that they are able to perform. The action levels used by the system are as follows:

- **Level 0**: This is the most privileged of all access levels allowing the user perform any action on the system. These actions include, arming/disarming the system, changing other user's credentials/access level, adding/removing users, adding/removing sensors, configuring zones. Users with this access level will also receive system alerts on their smartphones regarding events such as intruder or fire detection by the system.
- **Level 1**: This is a more restricted level of access. A user with this access level will be only be able to arm/disarm the system, however they will be able to do so from both a control panel and the smartphone app. Users with this access level will also be able to receive alerts from the system on their smartphones.
- **Level 2**: This access level is very similar to the previous one in that it only allows the user to arm/disarm the system; however they can only do so through a control panel and not through the smartphone app. Users with this access level will not receive system alerts on their smartphones.
- **Level 3**: This access level is meant to be used for guest users and only allows a user to arm/disarm the system through the in-home control panel. Unlike the other access level, users with this access level are meant to be temporary and thus will have their access automatically revoked after a specified amount of time. Level 3 users will not receive system alerts on their smartphones.

Each access level is design with a specific type of user in mind. Level 0 is intended to be used by a single person in the house most likely the home owner. Level 0 users effectively control all other users and are also able to configure the system. Most other users in the home will be either Level 1 or 2. These two levels will allow users to perform most of the necessary actions such as arming/disarming of the system, however only Level 1 users will be able to interface with the system through their smartphones. Smartphone interfacing includes being able to alter the system state through the smartphone app and also being alerted to events that are detected by the system. The least privileged of the access levels, Level 3, is meant to be used for guests who may stay at the home. User accounts with level 3 access will have a limited lifetime, determined during user account creation. This means the home owner doesn't necessarily have to micromanage various users and worry about forgetting to deactivate guest users as this will be done automatically.

The inclusion of the ability to alter the system state via the smartphone app means that additional security measures need to be in place. These measures need to ensure that any communication that happens between system and the smartphone app is safe, secure and not prone to manipulation by malicious entities. To accommodate this, all communication between main system and the smartphone will be encrypted. In order to enable communication between the main system and the smartphone the user or system installer will manually need to place the encryption key on their smartphone, with a corresponding key residing on the main system. This key will allow the smartphone to encrypt all communication and only allow the main system to decrypt, and vice versa. This will aide in preventing malicious entities from interfering with or influencing the communication between the smartphone and the main system.

Global Software Control

The handling of all requests is centralized to the main controller of the system. This controller is responsible for sending and receiving sensor data, displaying data on an interface and handling user input. Requests made via the smartphone app are packaged, encrypted and sent over the internet to the controller. The controller then decrypts the received message, extracts the requests contained within and performs the necessary actions. The controller may also send data back to the smartphone that initiated the request to inform it of its completion or any possible exceptions that were encountered.

Synchronization is an important tool that helps ensure that the system stays in a consistent state at all times. The fact that the system supports multiple users and that these users are able to make changes to the system state via their smartphones means that synchronization is needed to keep the system state consistent. This is in addition to the general synchronization required due to the concurrency aspects of the system. One of the ways in which synchronization between the smartphone app and the main system is achieved is through the command design pattern. The smartphone app is treated as another interface to the system akin to a wall mounted in-home control panel. This means that the smartphone app does not duplicate the functionality of the main controller, but rather takes user commands and forwards them to the main controller. Thus changes to system state are only possible in this single location in the system, enforcing the necessary synchronization. This implementation does however come at the cost of performance. The responsiveness of the actions performed by the user is negatively impacted by this scheme. For requests that involve interactions with the data stored in the database, the system employs database locking to aide in synchronization. When completing a request the system uses database locks to ensure that only one user is able to access the information at a time. This prevents situations where certain orders of operations on the database cause the system to be left in an inconsistent state.

Threads are used in situations that require the system to respond quickly to an event. For instance, the communication between the smartphone app and the main system has a dedicated thread to improve the response time. Traditional synchronization techniques are employed, such as semaphores, for inter-thread communication.

Any attribute that is possibly modified by multiple threads will have a corresponding semaphore associated with it. The semaphore will enforce mutual exclusion and help the system achieve synchronization.

Boundary Conditions

Describe significant activities involved in the start-up, configuration, and shutdown of any subsystem or component. Also discuss how faults and errors are handled. A refined use case diagram may be provided that shows boundary activities.

System start-up for both the main system and the smartphone app is a very involved process. Figure 8 below shows a refined use case diagram regarding system start-up and configuration. Please ignore the multiplicities between the user and the use cases; this is an artefact of Visio which was used to create the diagram. For system start-up, the system needs to establish a connection to the database, read in necessary configuration parameters from database and verify that all sensors are functioning and able to communicate. The possible faults that may be experienced by the system are also shown in the figure below. In the situation where the system is unable to establish a connection to the database, and thus unable to read in configuration parameters, the system will display the appropriate error message to the user and fall back to a default system configuration. This default system configuration will allow to the system to continue to function albeit with some limitations and also allow the user/technician to debug the issues. In the case where the system is unable to verify one or more of the sensors, the system will still complete the start-up process. It will alert the user of the issues and continue to function normally without the faulty sensors.

The smartphone app start-up process also involves establishing a connection to the database. This allows it to read in the necessary configuration parameters that the smartphone app needs to function. The user is then either automatically logged in or asked for login credentials depending on the configuration parameters. In the case of the smartphone app, failure to connect to the database throws a more fatal exception, in that there is no default fall back mode. This is done for security and also due to the fact that the smartphone is less crucial to the functioning of the system than the other components.

System shutdown is not shown in the figure below, and is simpler process than start-up. Shutting the system down completely requires the authentication of the highest access level user. When a request to shut the system down is received the user is prompted with the password of a Level 0 user. This is to ensure that only a select few individuals are able to completely shut the system down and prevents malicious or accidental shutdowns.

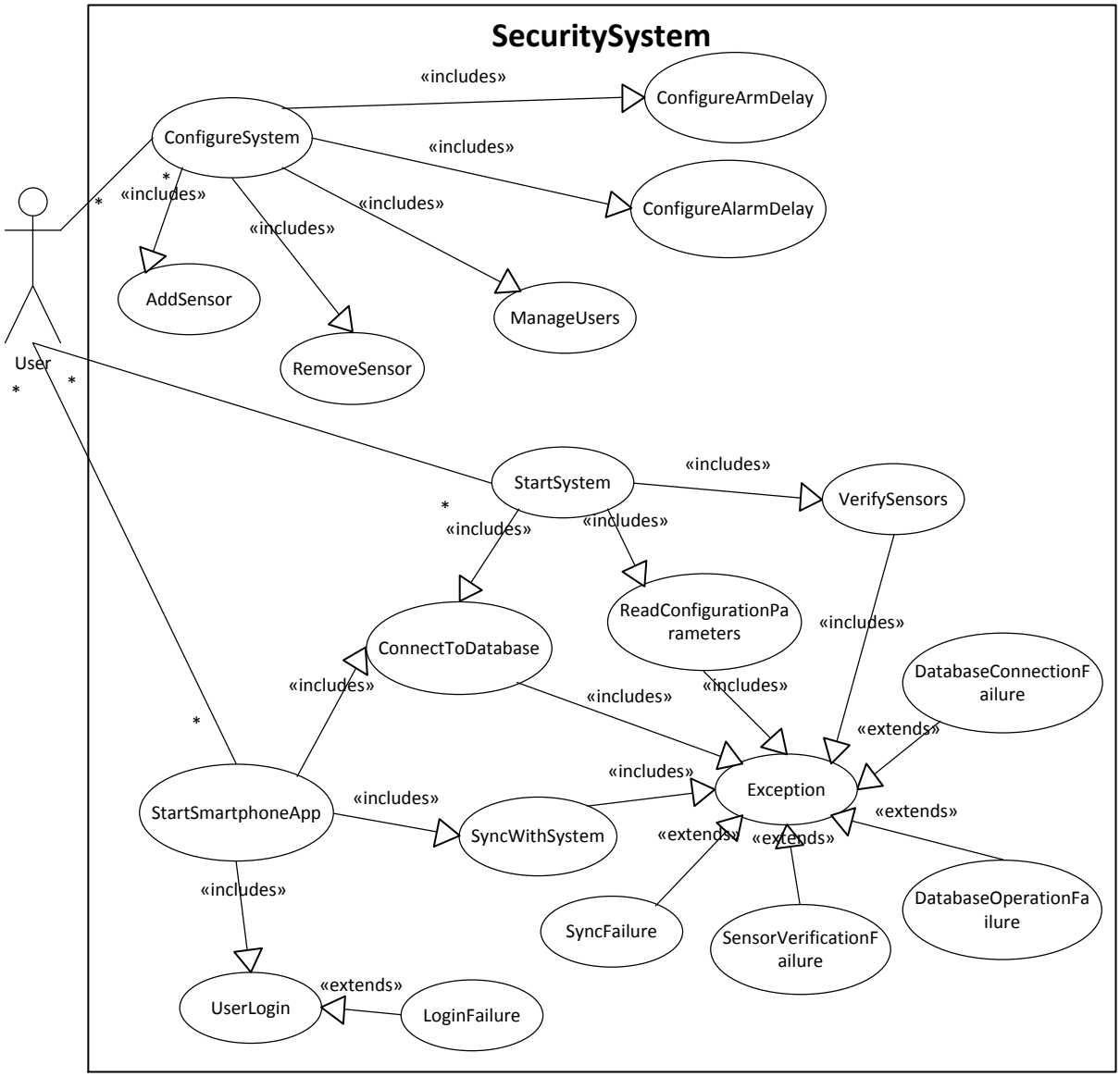

Figure 8. Refined use case diagram showing boundary conditions for system start-up and configuration

Interfaces

External System Interfaces

The alarm system interacts with the emergency response services and the cell phone service providers in order to facilitate the working of the system.

The Interaction of the system with the emergency services is made in order to describe the emergency situation in hand. The alarm system not only calls the response team telling them about the appropriate sensor which has triggered the event while at the same time giving them the GPS co-ordinates of where the house is located. The interactive functions which take place between the EmergencyServices() class and the Alarm system are shown in figure given below.

The alarm system also interacts with the phone service provider of the house owner. Since it sends and receives messages to the owner in order to alert him. The user can also choose to see a live view of the house in which case the system instead of transmitting textual data will have to transfer video data over the video frequency bandwidth. Thus, the functions and parameters needed by the class PhoneServiceProvider() are shown below. These parameters and functions allow the system to effectively interact with the alarm system.

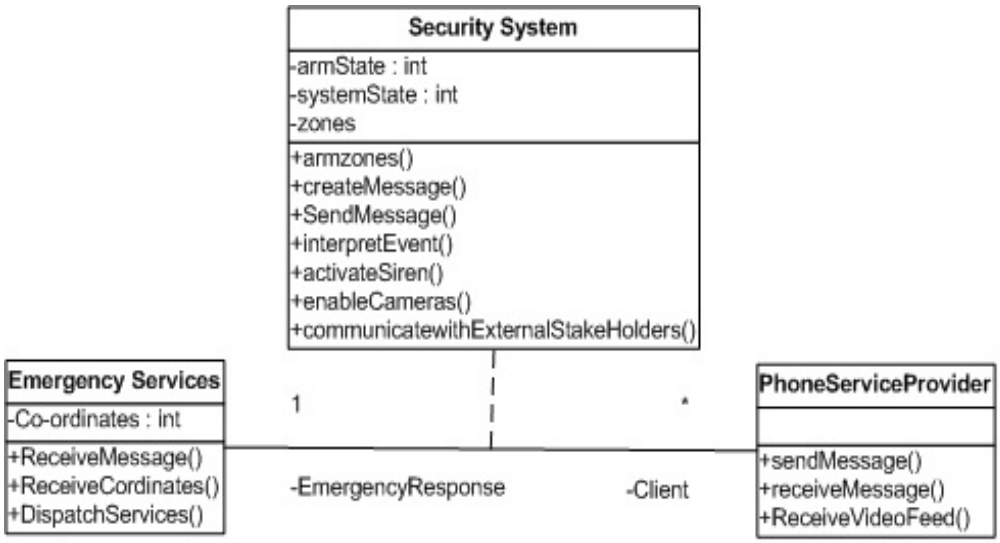

Figure 9: External System Interface

Internal Subsystem Interfaces

The security system communicates with the external classes EmergencyServices() and PhoneServiceProvider() as shown in the above section. The internal model on the other hand has 3 major object classes. The Parent class known as ‘SecuritySystem’ has two children classes ‘Zone’ and ‘Sensor”. Each sensor reports to the corresponding zone, which in turn alerts the security system. The security system then uses the SendMessage() function to inform

the emergency services and the phone service provider of the security breach. On the other hand if the enableCameras() function is invoked by the home owner, then the system responds back to the 'PhoneServiceProvider()' class with the live video feed.

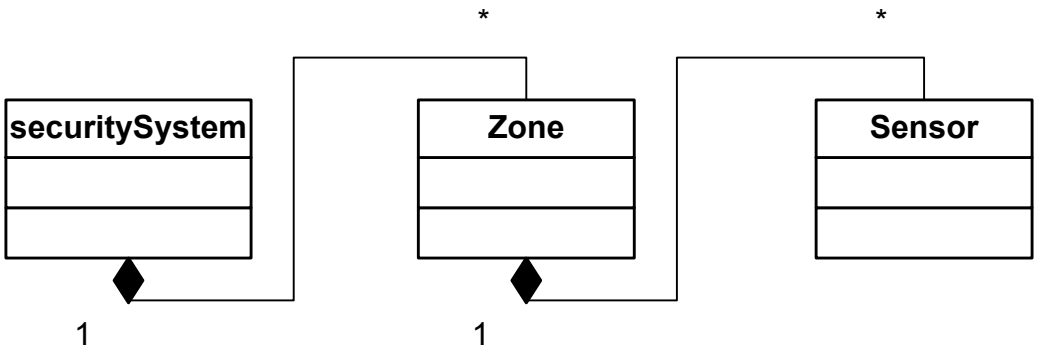

Figure 10: SecuritySystem class diagram

The Security System, which resides in the central control unit, is in turn connected to the different sensor classes. These sensor classes are shown in the following figure:

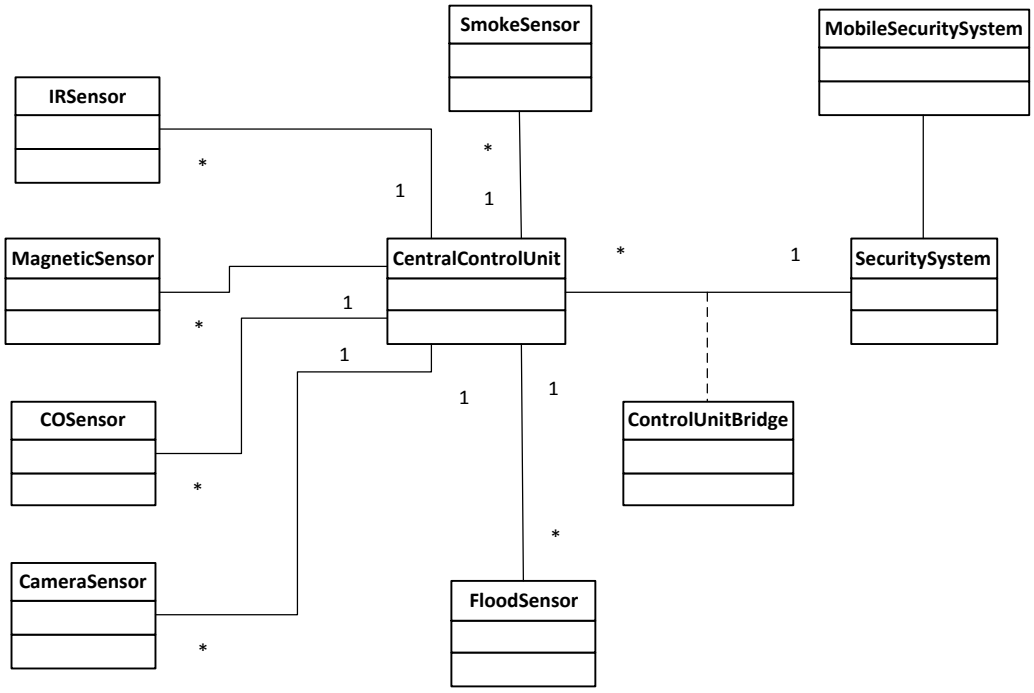

Figure 11: Overall system class diagram

Object Design

Design Patterns

There are a few design patterns that will be used for the implementation of this project and they are:

1. **Adapter Pattern:** Used for the Controller
2. **Bridge Pattern:** Used for the communication between controller and sensor and user interface
3. **Abstract Factory Pattern:** Used for the User Interface component

Adapter Pattern:

Adapter pattern is like a wrapper pattern which transforms one interface of a class into another interface of the class so that it can communicate across interfaces. This helps in creating independent versions of the subsystems and then a wrapper could be written to combine them to form a common subsystem.

As mentioned earlier, the controller will be split into two parts, one dealing with the sensors and one with the user interface. The idea is to create an adapter interface between these 2 components of the controller for them to communicate with each other.

The following figure shows the adapter pattern that would be used in our design. It shows a subsection of the controller subsystem which will be affected and used in implementing the adapter design pattern.

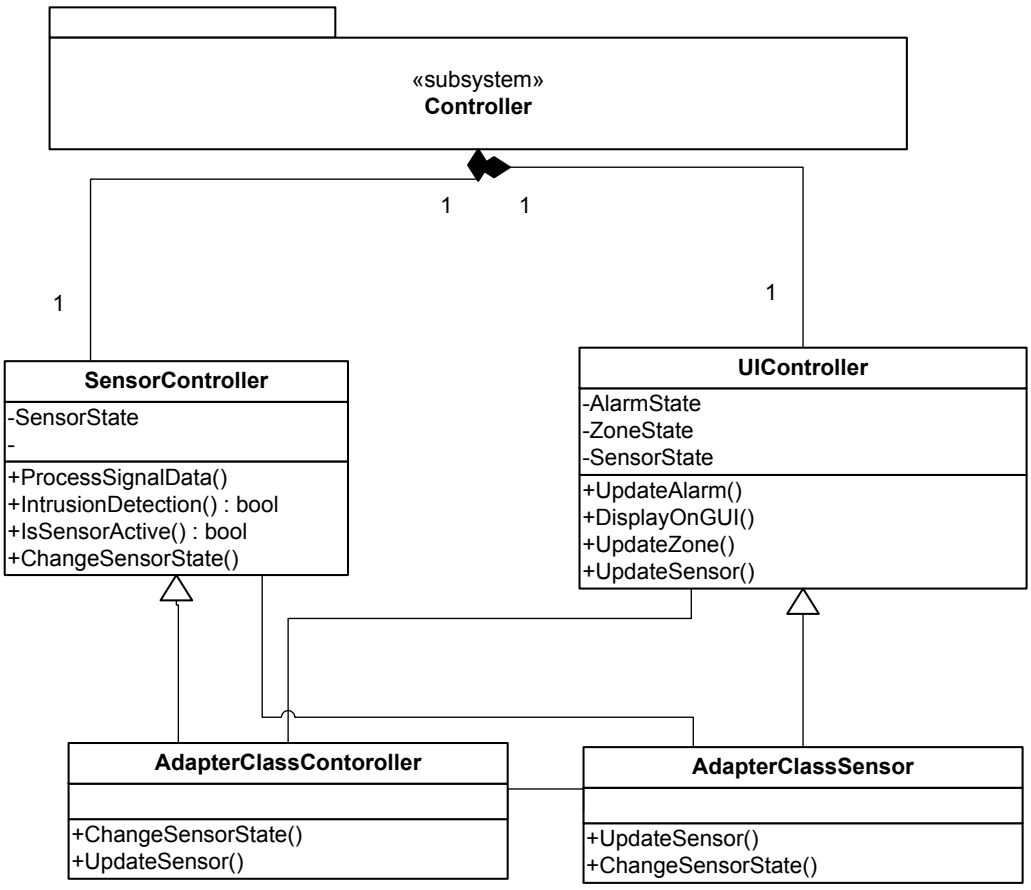

Figure 12: Adapter Implementation of the Controller Subsystem

The rationale behind this is that, since these are two individual parts of code which will be residing in the controller, we would like to keep them as separated as possible. This will help them to be created by different developers with their own unique styles (described below). The interface would not have to be necessarily generalized as during the development, the interface and requirements change and it would be more challenging to then change the whole system. If these 2 subsystems are completely independent, then the respective developers could in future change their own parts without disturbing the whole code base and still maintain the existing functionality. This would also lead to less dependency among classes as the user interface would be completely independent of the sensors. Any change/addition to number of sensors would need no change in the user interface, but only updating the adapter class. Similarly, addition of a new user interface would require no change of the controller part dealing with the sensors. Thus, this would ease in development and also keep things separated.

This would also help in testing of the system as they are smaller in nature and could be individually tested. Once, tested individually, then during integration phase it will be only the adapter class that would need to be tested and would need to be debugged. Thus, this would save a lot of time during integration phase.

Moreover in future, if additional alarm systems or monitoring systems needed to be added to the whole system, then keeping these separate will of immense use. In that case, the user interface won't change much and this user interface could be then reused to implement for the complete system. Thus, this would help in extendibility of the system.

Bridge Pattern:

Bridge pattern provides an interface to unimplemented component. It provides a bridge between the abstractions of the component to be developed in future to the existing system. This helps in developing the existing without worrying or testing the future component and that component can be added at a later time. It thus, delays the actual binding between the client and the interface during run times.

We will separate out the sensor and user interface and implement them as a bridge with the controller. The following diagram illustrates the bridge pattern for our system.

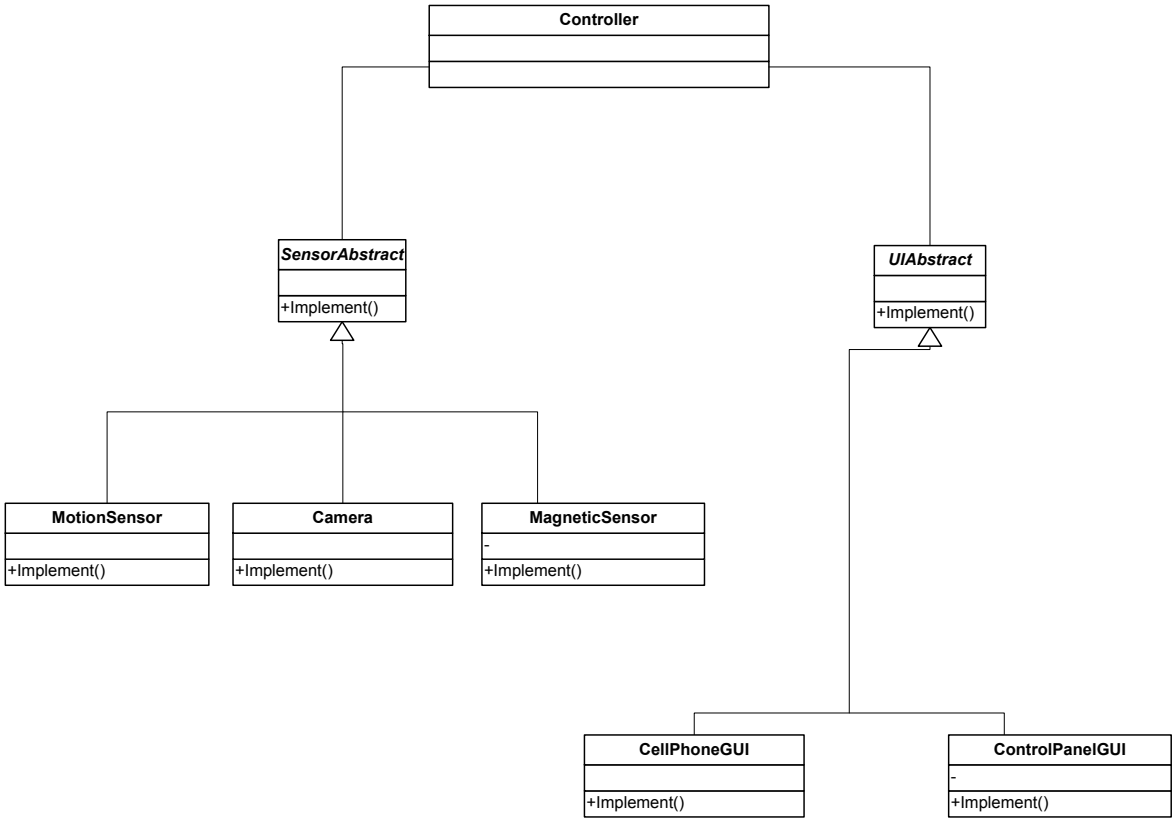

Figure 13: Bridge Pattern for Communication

The reason we have chosen this pattern is to model our communication of the controller with the user interface and the sensors. As all three of these subcomponents are independent of each other in implementation, but highly depend on each other during run time, it is beneficial to separate them out while creating a common interface between them. This would help in easy and individual development of each of the components. Moreover, the standard interface for all the sensors would be the same and thus in future adding another sensor would be very easy. Similarly adding a separate user interface in future would be easier as then the controller would not need to know how the user interface works but would just deal with the interface of it. Thus, this would both help in rapid development and extendibility of the system in the future system.

Abstract Factory Pattern:

This pattern encapsulates a group of subclasses that have a common theme. The implementation of is mainly to interface the abstract interfaces of the classes to the system and then implementing a concrete version of these abstract classes. The way this is implemented helps in creating a scenario where at run time the program can decide which abstract factory to use to create an object. This helps in creating a unified interface which is independent of initialization and representation and is manufacture or platform independent. The different factories can have very different implementations which could be platform or service specific.

Abstract factory fits well with our system of user interface. The user interface is consistent across the control panel GUI and the cell phone. However, both of them are very different, as the control panel is an embedded system and the cell phone GUI is a smartphone based system modelled on top of iOS, Android, QNX, Windows or Blackberry. The main aim for us is to create a consistent layout across all the platforms and being able to re-use all the common classes while deciding at run time which kind of graphic layout to use.

Another advantage of this pattern is that we could easily then extend it over different Operating Systems which are getting popular in smart phone market. Thus, this would help us in extendibility and re-using some of the code that is developed right now.

Algorithms

There are two major algorithms that will be used to solve some computational problems. These are image processing and RSA Key encryption.

Image Processing:

As mentioned in the SRS document, that the controller would process the images coming from the camera to determine the difference between an intruder or a pet [1]. The way this algorithm will work is that once the motion sensor for a zone sends a motion signal, then the camera will start sending a video feed to the controller. The controller would then look for the motion patterns in the video and segregate the part that is moving. It would then do a analysis on how tall, big the area where motion is. It would locate its color and does a pattern match with the stored photographs of the owners pet. If these pattern/colors match with the pet it would deduce it's the pet moving. Otherwise it would inform the owner of an intrusion and send him a live feed of the zone.

If a burglar tries to camouflage himself as a pet then it might succeed in initial trespassing however, as soon as the burglar tries stands up or opens a closet or any other mischief the camera would recognise and deduce it to be a burglar. Hence, the system will still work.

RSA Key Encryption:

As mentioned in the SRS document, that during the installation of the system a RSA key would be generated and transferred to the mobile device [1]. This is necessary because all the communication that will happen between the controller and the cell phone will be encrypted to ensure if due to network errors the message is delivered to a wrong cell phone, the person can't decipher it. Thus, in the controller a RSA key Encryption will be done and on the cell phone side the same decryption would be done for messages going from controller and cell phone and vice versa going from cell phone to controller. The RSA key used will be a random 2048 Bit key encryption, thus which is very secure encryption.

Packages

The system can be divided into multiple sub-systems and further into certain packages. These subsystems have certain common packages, which are being used all over there system. These packages allow the system to carry out its processes while maintaining certain standard across the whole system.

Authentication Package - The authentication of the client before it can access its own home security system is of outmost importance in the system. This must be standardized and highly secure. RSA encryption would be used in order to encode and decode messages, which would be sent on the network. The RSA key on the client and the addition of that key on the server end allows only a specific client to de-encrypt the message being passed around each time a new client/user is added to the system, the RSA encryption key generated by the clients cellphone needs to be added to the control server. Also, the user interface or the app running on the client's cellphone would require a separate authentication for the user to able to access the system.

Messaging- The messages sent from the system to the clients cell phone and the messages sent back by the client, all would be encrypted using RSA encryption. The messages sent out between Central Control system and the sensors would be just signals and would not require any kind of encryption.

Interpretation- every system needs a certain amount of logic or artificial intelligence. In the following system the although most of the interpretation is carried out in the Central Control Unit, still it is important for every sub-system in the architecture to know the signal which has triggered the alarm and perform it's function accordingly. Thus the package of logic allowing the sensors to interpret and choose their functionality based upon this interpretation is consistent between all the sub-systems.

Interdependency & connectivity- all the subsystems are highly dependent upon each other. Thus a standardised package for establishing communication protocols between them is necessary. The protocol will tell them, which port to use to for interconnectivity between them and which error to display in case a communication channel is not established. This would also standardise the errors and actions, which should be taken in case an error is received.

The figure below shows the dependency of the subsystems on the above mentioned package and how they can be used to standardize the system.

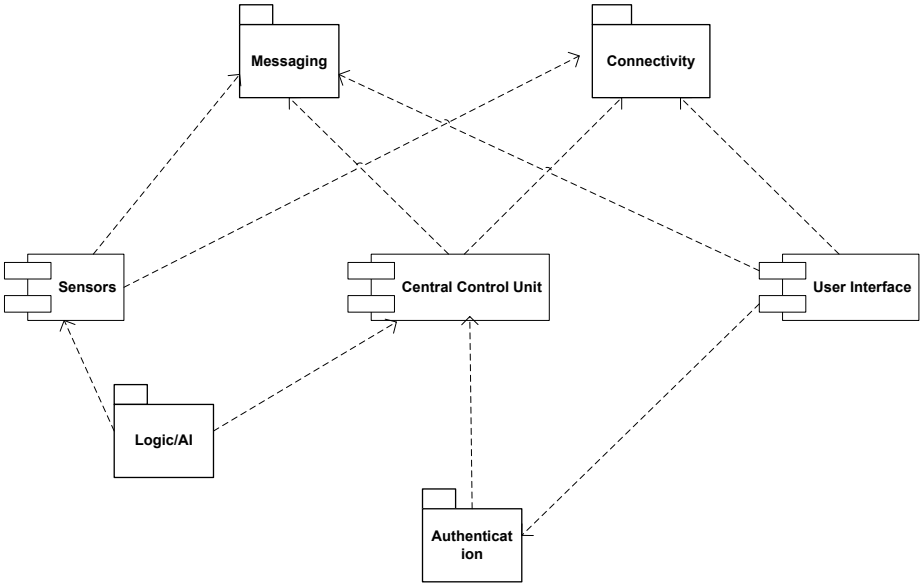

Figure 14: Shows dependency of subsystems on packages

Object and Interface Design

We will break down the object diagram according to the subsystem:

Sensor Subsystem:

Figure 11: Overall system class diagram, represents the overall sensor diagram in conjunction with the central control unit. All the sensors communicate with the central control as clients as discussed above. The following diagram shows the interface and class diagram for one of the sensors.

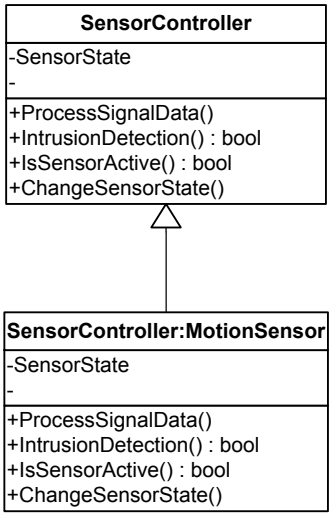

Figure 14: A high level Diagram of the Sensor Class Implementation

The above diagram shows the high level implementation of the sensor interface and the implementation of the motion sensor.

Table 1: Decription of SensorController

Attribute	**Description**
SensorState	Tells if the sensor is active or not. The sensor can be inactive if the user has not turned it on.
Operation	**Description**
ProcessSignaldata	It processes any incoming signals from the controller. This is generally to arm or disarm the sensor
IsSensorActive	Sends the active state signal to the controller when requested.
ChangeSensorState	This will update the state of the sensor depending on signal from controller
IntrusionDetection	If sensor sends a hardware signal to itself this will send the intrusion detection to the controller.
The exceptions that must be handled are in case the sensor is running out of battery and can't make adequate connection with the controller it should try and keep sending "battery low" signal to the controller to make sure the controller is receiving that signal.	

User Interface Subsystem:

The user interface subsystem has been depicted in the diagram on the next page. As mentioned above the user interface subsystem has been implemented user abstract factory.

GUI Application is the application running on the control panel or the cell phone which will manage the complete user interface.

Table 2: Decription of AbstractGUIClass

Attribute	**Description**
Button	Specifies a type of button that can be drawn on the GUI.
TextBox	Specifies a type of textBox that can be drawn on the GUI.
Operation	**Description**
CreateButton	This method creates a button on the GUI
CreateTextBox	This method creates a textBox on the GUI
CreateList	This method creates a list on the GUI which is an array of textboxes and buttons
EnterPassword	This method creates a textbox which is special as it takes inputs and then verifies it against the password stored in the database.

Table 3: Decription of Button, TextBox, List, EnterPassword

Operation	**Description**
Paint	This method paints the kind of object on the screen

These in reality are implemented in concrete classes as shown in the diagram.

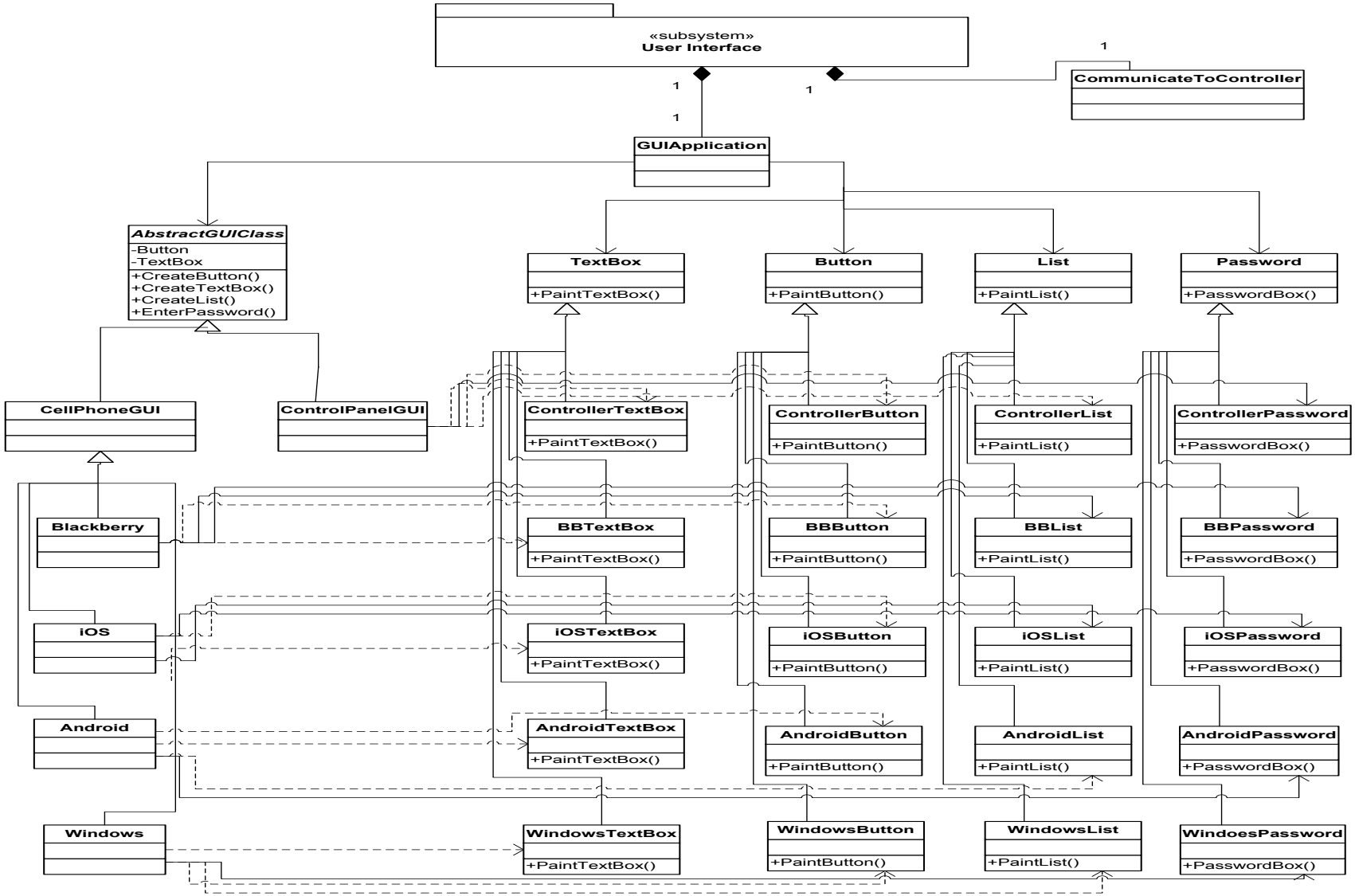

Figure 15: Abstract Factory for user Interface

Another class which the User Interface subsystem comprises of is the communication with the Controller. The details of this has been provided the Controller Subsystem section.

Controller Subsystem:

The controller subsystem has been broken down into 2 major subsystems, one dealing with the sensor and the other dealing with the user interface. Another aspect of the controller is the data processing which happens in the SecuritySystem class of it.

The following figures show the complete controller:

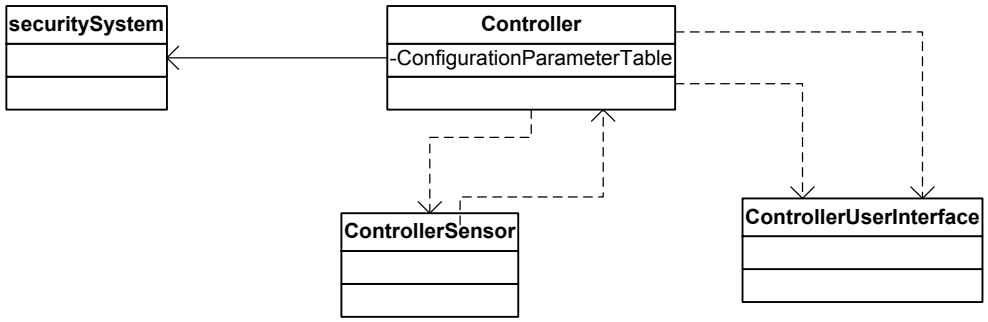

Figure 16: A high level diagram of decompostion of Controller

The persistent data management will be done by the configurations parameter table explained in **Error! Reference source not found.**. This table is a part of the Controller Class.

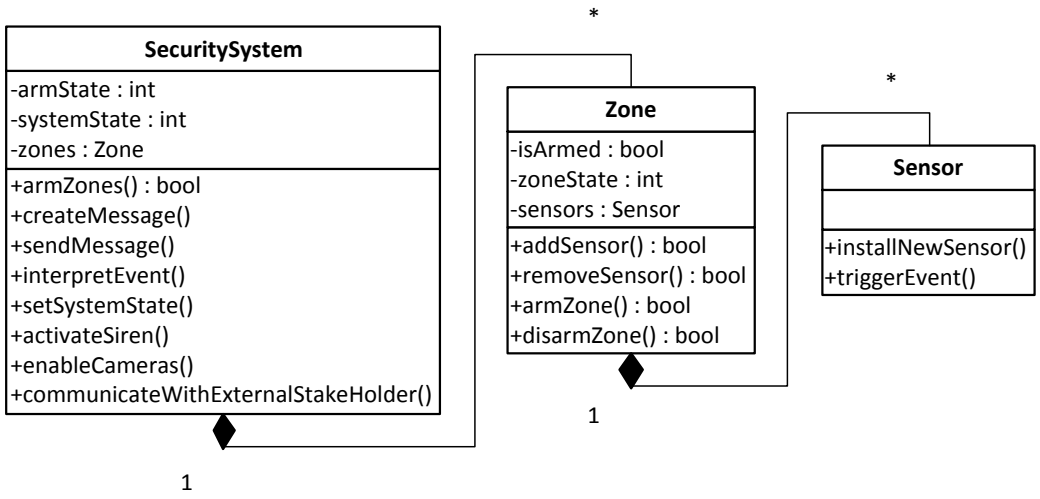

Figure 17: Detailed Security System Design

The SRS provides a detail description of all the objects related to the boundary conditions of the Security Subsystem on pages 17-21 [1].

Table 4: Decription of SecuritySystem

Attribute	**Description**
armState	Tells if the security system is armed or not
systemState	Tells the percentage of system armed
zones	Lists all the possible zones
Operation	**Description**
armZones	Arms a certain zone
CreateMessage	Create a message to send to contollerUI class
sendMessage	Sends a message to send to contollerUI class
interpretEvent	Interprets event from the controller and maps them to sensor specific signals
activateSiren	Activates the Siren
enableCameras	Cameras can be shut off but in case of intrusion all of them will be enable to capture all the feed.
SetSystemState	Vary systemState
communicateWithExternalStakeholder	Creates the message that needs to be sent to the emergency services

The Zone class consists of the zone state, number of sensors and if its armed or not. You can add/remove a sensor or disarm/arm a zone.

The Sensor class can be used to add a new sensor or trigger an external event in case of intrusion.

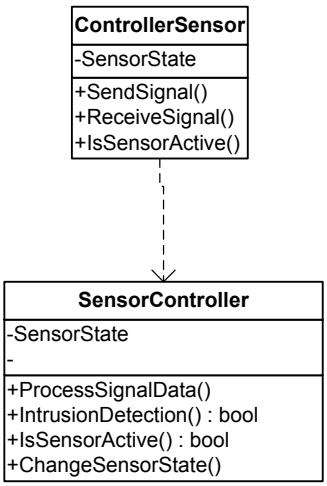

Figure 18: Controller Dealing with Sensors

Table 5: Decription of ControllerSensor

Attribute	**Description**
SensorState	This is the current state of the sensor. If the sensor state just changed this will be the updated sensor state and that will be updated to the sensor.
Operation	**Description**
SendSignal	This method sends the state signal to the sensor to arm/disarm it.
ReceiveSignal	It receives the signal and data from the sensors, processes it to see for a possible intrusion.
isSensorActive	This method calls the sensor method to check if sensor is active or not.

The exception that needs to be handled is if the sensor stops responding. Then the controller should inform the own as soon as possible.

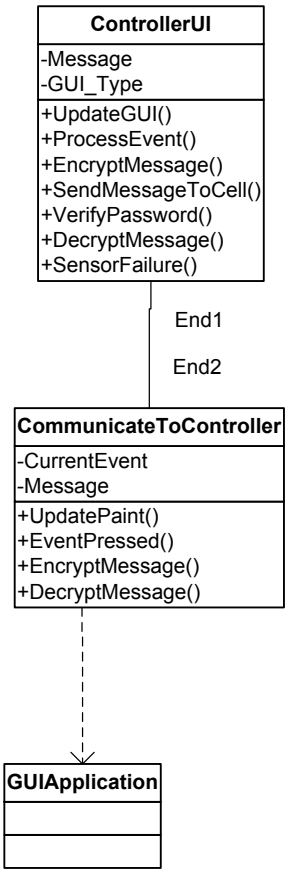

Figure 19: Controller Dealing with the User Interface

Table 6: Decription of ControllerUI

Attribute	**Description**
Cuurent Event	This specifies the event which has been pressed by the user. It could be button to arm/disarm the system, zone or sensor, a keyboard input or list of cameras
Message	This is the message that is received from the cell phone or control panel to change the state of the system or the message made by the controller in case of a warning or when the system state changes.
Operation	**Description**
UpdatePaint	This method is called when the controller wants to update the paint on the system. This will be resulted if the user interacted with the system and changed its state or a sensor failure changed the state of the system
EventPressed	This is a callback done when the user presses a button on the screen
DecryptMessage	Decrypts the message using the RSA key that is arrived from the control panel
EncryptMessage	Ecrypts the message using the RSA key that is being sent to the control panel
The exception that needs to be handled is if the controller cannot create a network with the user cell phone in case of a warning. At that time it needs to try calling the user cell phone or auxiliary numbers stored in its memory by the user to call in emergencies.	

Table 7: Decription of CommunicateToController

Attribute	**Description**
Cuurent Event	This specifies the event which has been pressed by the user. It could be button to arm/disarm the system, zone or sensor, a keyboard input or list of cameras
Message	This is the message that is received from the controller in case of a warning or when the system state changes. It could be the message being sent to the controller when we are updating the system remotely
Operation	**Description**
UpdatePaint	This method is called when the controller wants to update the paint on the system. This will be resulted if the user interacted with the system and changed its state or a sensor failure changed the state of the system
EventPressed	This is a callback done when the user presses a button on the screen
DecryptMessage	Decrypts the message using the RSA key that is arrived from the controller
EncryptMessage	Ecrypts the message using the RSA key that is being sent to the controller
The exception that must be handled if the event that is pressed on the cell phone is also being pressed at the control panel at the same time. In this case the cell phone would be given precedence as the control panel can be modified by the intruder in the house.	

Dynamic Design Model

The dynamic design model shows the dynamic behaviour of the system and its associated classes. The following section utilizes activity and sequence diagrams from the previous documentation. These diagrams have been further optimized in order to aptly represent the border conditions and the exceptions which the system might face during execution.

The sequence diagram stated below shows the sequence of events which would take place if a burglar breaks in through a window of the house. The window sensor would send in a trigger event to the central control unit, only if it is properly connected to the system. The Central Control Unit (CCU) would in turn process the event and send the appropriate information to the client. If needed also provide the client with a video update of the house. After this interpretation has taken place, assuming the client is able to correctly interpret it as a burglary the system then sends a signal to the siren. This siren located in the house would only start blowing off if it is correctly connected to the system. Most of the exceptions stated here can be clubbed under the category of communication error, which can be dealt in standardized manner by having a package for messaging as stated in the above section.

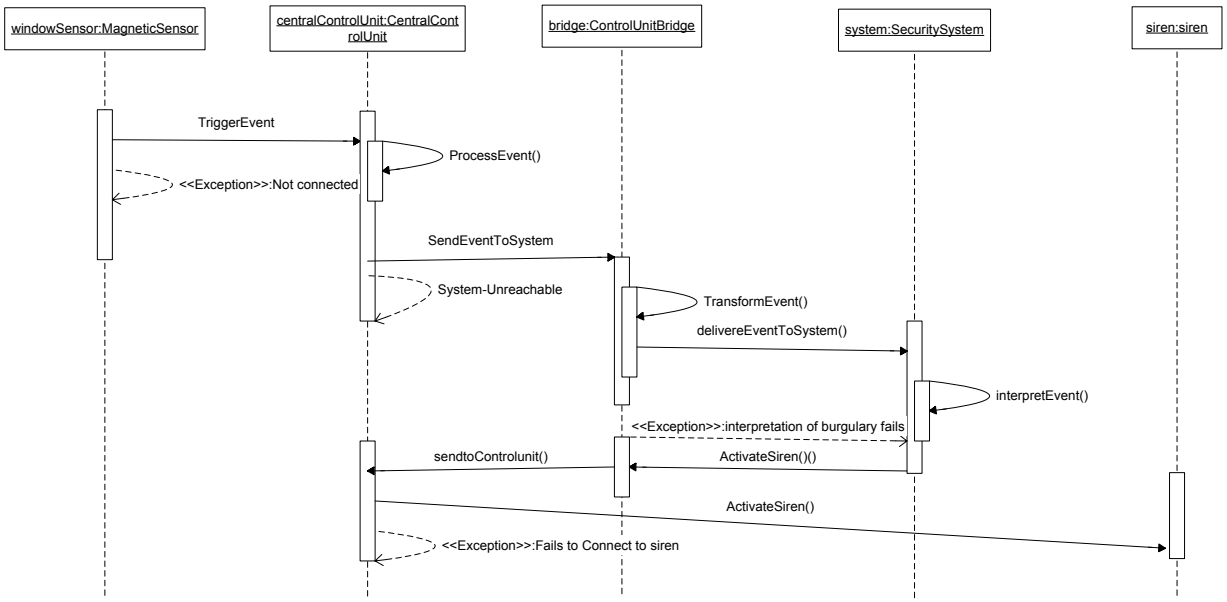

Figure 20: Sequence diagram representing the flow of events during burglary

Further we have effectively analysed the system with the help of an activity diagram. The activity diagram also states the boundary conditions and the exceptions which we have analysed in this report. The figure analyses the scenario when an object is noticed in front of a camera or an IR sensor, provided that these sensors are connected correctly to the system. The system then decides whether the object is a pet or a human trying to intrude into the house. Based upon it's analysis it then contacts the client and the emergency services at the same time. This step might face an exception if the system is not able to connect to the client or the emergency response team. Also, if the client replies back asking for a live feedback of the house, then a large amount of constraints can come into play. The video processing can take time, the data transmission can be really slow or the camera could not be connected properly, all of these scenarios can equally take place in the system. Thus these constraints and exceptions have been aptly shown in the following activity diagram.

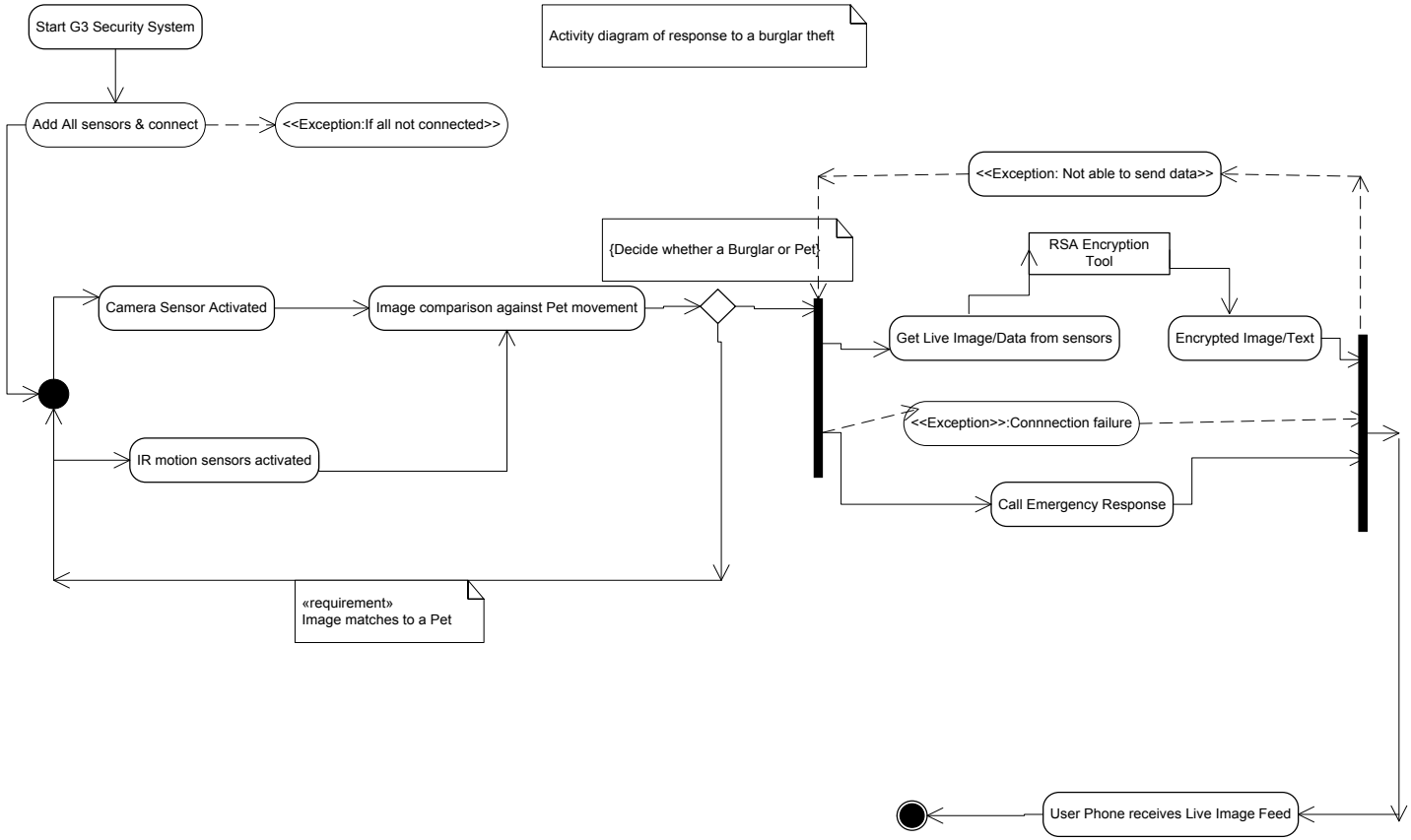

Figure 21: Activity diagram representing the flow of events during the detection of an unknown object

Design Evaluation

Design Trade-Offs

In G3 security system a lot of trade-offs that can easily be observed and stated since certain functionality of the system has been given higher priority than others. These trade-offs have indeed been made to the system to make it more efficient and robust while satisfying all the needs of the client.

The main trade-offs of the system are:

Security v/s Speed – The system will be programmed to have very secure connections for transferring data between sub-systems and the user. For this, RSA encryption and secure port connections are used. This although will drastically increase the security of the connection, but would also lead to decrease in overall speed of the system. Since establishing a secure connection between two sub-systems would take more time, thus messages will take more time to transmit, thereby halting the system for a much longer time. Making such a trade

Extendibility v/s Development – The system that we are creating is focussed on easy extendibility on future. This adds much more work to finish in the present as interfaces need to be created in such a manner to make sure in future with less code change more features can be added. Thus, this requires good architectural and design patterns and longer time to implement and test. Thus, the development process becomes harder.

Buy v/s Build – The security components that we are looking into are to support components from various manufacturers. We would assemble the hardware components and interface them with our controller. The controller is one part in addition to the software that we will be creating ourselves. This is because there are a lot of good off the shelf sensors out there and it's easy to just use those. However, the controller is the brains of the project and we want full control over it. Thus, developing that from scratch is a better option.

Scalability v/s Performance – As the system has an object oriented design; theoretically it will be capable of handling any number of zones and any number of sensors. However, as we have only one controller, the more the number of sensors or zones, the more processing the controller would need to do. In such a case the controller would become a bottle neck and it won't be able to respond to the sensors in a real time manner. Hence, we will focus on having a high performance for a limited number of sensors.

Security v/s Number of Cell Phone Users – Our system supports interaction via the owner's cell phone. However, this could also lead to security breach quite easily. This can happen in the scenario the owner lends his phone to someone with the application running in behind and the owner already logged into it. Or the owner's phone gets stolen and the wrong doers are able to get the security zone information before the owner is able to contact us about this. Thus, to maintain security we would limit only 1 cell phone per controller for a home owner on the basis that the head of the household needs to be responsible enough and also he/she should be only the one who should control the security of the house.

Memory v/s Response Time – The higher the memory of the system the more information it can store about the current states of all the sensors and also about the patterns and images of a user's pet. Having a database of this information would help the system to respond to user queries faster and distinguishing an intruder from the pet. Thus, we will choose high memory as we want a system that is very responsive to user interface and works in real time.

Memory v/s Availability – Having a high memory would enable us to store the failure rate and period of each sensor in past and with time the controller will be able to predict when the sensor is about to fail, so that the user can take appropriate action before that. As we want the system to be available for the maximum time possible we would go with higher memory.

Cost v/s Performance – As mentioned in the above trade-offs, we are choosing higher memory and expensive off the shelf components, this brings the cost of the system to be relatively high. However, as the system is protecting a home, business or office which has a lot monetary worth to it, it's important that it should be reliable and really secure. Thus, increasing cost to maintain its accurate performance is the choice that we would make.

Re-use

The design patterns that have been chosen in this system have been chosen keeping re-use in mind. All the three, adapter, bridge and abstract factory use inheritance and delegation as their major components.

The system is a modular system based on object oriented design. All the classes have been first modelled as interfaces and then divided into base classes. For example, in the user interface subsystem, the GUI is an abstract class that has cell phone and controller as its children. The cell phone has all the various operating systems as its children. The GUI layout has text box, button, list and password as its children and each of them is implemented as an object of the cell phone or operating system. Having such a structured design helps to re-use common functionality between classes and only re-implement the things that from one platform to another.

Moreover, the MVC pattern for the User Interface is a very common pattern to create user interface and hence we would be using third party libraries for GUI interface to create the user interface. This would help us to not create the system from scratch and re-use existing code.

The RSA Key Encryption is already implemented for all the programming languages and they would be easily added to our system to generate keys, decode and encode data.

For image processing, there are external libraries like OpenCV which would be used to perform the basic image processing manipulations. Hence, this would reduce the effort of our developer and reuse the code written by the community.

Optimizations

In order to optimize the final iteration, the basic approach used was to standardize code and remove any repeated components from the system. For this we heavily analysed any hierarchical structures and combined them to make a parent and child class.

Moreover we used the concept of packaging in order to bring together common elements of the object model. By doing so we were also to standardise the errors which can be caused by the system.

We were able to make a separate block for RSA encryption which would readily do the encryption of messages being sent and received. This would certainly optimize the system speed by reducing the expensive delay which is caused in establishing a connection with the system.

Extensibility

The design of the system has been made keeping extensibility as a major factor. Both our architectural design and design patterns easily support any extended features.

Modelling sensors as clients and with a Bridge design pattern means that we will be creating a common interface for all the sensors. In future addition of any sensor to the system would require writing a library or sub set of code for

that particular and interfacing it with the current interface. The controller doesn't even need to know the kind of sensor till the time the signal interface matches. Also, this would allow having sensors from different manufacturers who might make improved sensors in the future. This would be done by adding a wrapper to the interface to make sure the new designs are compatible with our system. The wrapper is a relatively easier operation and would not require any modification to our system.

The user interfaces have been created as an abstract factory model. This gives us the option to add a new factory depending on the platform and use it during run time. Thus, the existing system doesn't even need to recompile if a new interface is added. Thus, this eases up the process of adding a new interface to the system.

Operating Environment

Development Platform

The entirety of the system will be developed on either a Windows or Linux machine using the Eclipse IDE. The language used for the implementation will be Java. Eclipse deep support for Java makes it a natural fit for almost any project being developed in Java. Eclipse provides a wide range or built-in features and the ability to easily extend functionality through plug-ins, making it a power tool for writing Java code. The Java language was chosen as it provides a wide feature set to work with. It has things like concurrency support, a simple yet powerful object model and "write once, run anywhere" functionality. The fact that Java code executes on a virtual machines means that the code can be easily ported to different architectures as long as a Java virtual machine exists for that architecture. This will aide in improving the portability of the system, by allowing it to be deployed on a variety of platforms.

Another reason for developing in Java using the Eclipse IDE is the fact that the smartphone component of the system will be based on Android. Application development in Android is done in Java and Google provides excellent Eclipse plug-ins as part of the Android SDK. This will allow the entire system to be developed using a single language and IDE, reducing the need to create adapters or dynamically link to DLLs.

Runtime Platform

The runtime platform for the system will consist of a small Windows or Linux machine running the main security system code. This machine will be located in a secure location with the premises where the system is deployed in order to prevent tempering. The fact that Java was chosen as the implementation language allows the system to be easily deployed to either a Windows or Linux machine. In addition to the Windows or Linux machine running the main security system software, there will also be a physical central control unit with which all the sensors will communicate. The Windows or Linux machine will then connect to this central control unit bridge. The role of the central control panel is to transform electrical signals being sent and received by the sensors into digital data that can be modeled using object-oriented principles. Any action that the main security system needs to perform involving the sensors will need to go through this central control unit. The central control unit will comprise both wired and wireless connectivity to allow it to support a wider range of sensor types. Like the Windows or Linux machine running the main system application, the central control unit will also be placed in a secure location within the premises where the system is installed to prevent tampering.

In addition to the above, the system will also generate Android packages knows as APKs which the user will be able to install on their Android smartphone. This APK will contain the smartphone app that interfaces with the system. The smartphone needed to run the application will be provided by the user of the system.

The system will also have one or more wall mounted in-home control panels. These control panels will allow the user to perform a wide range of actions. Everything from arming/disarming the system to adding new sensors, to managing user will be done through the control panels. The control panels will communicate directly with the main system controller allowing the system to respond to user input. Error message and alerts will also be displayed to the user via these control panels.

Process Model

The system will be multi-threaded. Using a multi-threaded approach allows the system to easily share resources between threads of execution. If concurrent processes were used instead, another means of sharing data would need to be used such as inter-process communication. This would significantly complicate the design of the system and create much greater overhead. While a system using concurrent process may be able to have true parallel execution, from the software requirements specification we know that performance is not a vital non-functional requirements [1]. The system needs to be relatively responsive to user input, and this can be accomplished using a multi-threaded approach. Thus, the increased complexity and overhead of a concurrent process approach was not selected. Similarly, during the requirements elicitation and then during the subsequent design phase, the team did not find a pressing need to implement a distributed system. The fact that the system control is centralized does create a single point of failure, however it greatly reduces system complexity and the overhead of communication between systems nodes is eliminated.

The system is broken into zones, with a zone being defined as a collection of sensors. Each zone is assigned its own thread to execute within. This allows the system to concurrently monitor all zones. Additionally, communication with the smartphone app is also relegated to its own dedicated thread. This is to help improve the responsiveness of the smartphone app, especially is situations where there is a great deal of back and forth communication between the two. The thread dedicated to communicating with the smartphone app maintains a message queue. The messages contain commands that need to be executed by the main system. On the smartphone itself, the majority of the application will reside within a single thread of execution, however a service will be created that is able to receive messages from the system. In cases where an event is detected, the system will send the smartphone app a message containing the alert and the relevant information. The service that is running on the user's smartphone will receive this alert and start the smartphone app's main thread to process this message. The smartphone app will then alert the user and display the relevant information.

Synchronization

The system will semaphore where necessary to enforce mutual exclusion. The fact that the system will be multi-threaded means that the changes in the state of an object need to synchronized to ensure only one thread at a time is able to modify the object's state. Semaphores for the attributes that comprise the state of an object will allow the system to maintain a consistent system state.

In addition to the semaphore used for synchronization between threads, the system will also utilize message queues for communicating with the smartphone app. Both the main system and the smartphone app will maintain message queues allowing them to communicate with each other. Since the smartphone app will act as an interface, similar to a wall mounted control panel, all commands received from a user will be encrypted and sent as a message to the main system for future processing. Message queues will allow the system to efficiently receive and handle these commands.

Fault Handling

There are situations where the system may encounter a fault and generate an exception. The way in which this exception is handled will depend on where it was generated and its severity. Examples of possible exceptions of varying severity can be seen in Figure 8 in the Boundary Conditions. For instance, it is possible for the system to generate an exception during start-up if it is not able to successfully connect to the database. This is a severe fault as

this means that the system does not have important configuration and user data available to it. The system responds to this situation through graceful degradation. Rather than shutting down and leaving the premises unsecured the system defaults to a default state with more limited functionality. In addition, the user is also alert of the error through the control panel, so that they may be able to further diagnose the problem that led to the exception. In this way the system is able to continue to provide at least minimum level of protection even in the case of a severe exception.

An example of a less severe, but still important exception would be one were a sensor stop transmitting data or stop responding to commands. This failure of the sensor could be due to a variety of reasons, everything from a dead battery, to a faulty wire, to malicious tampering. The system in this case, will continue to function normally, however it will alert the user of the sensor failure. The user is then able to investigate the problem and remedy it accordingly. Generally, the system will be designed such that lower level exception are caught be higher level processes and handled according to the exception type.

For hardware components that operate on battery power, the specific component will periodically send a keep-alive message to the system. This message will allow the system to know that the component is functioning. If it does not receive this message from a component then an exception will be generated and handled according to steps discussed in previous paragraph. In addition to this keep-alive message, a user will be able to trigger a manual verification process. The process will involve the system entering test mode and the user going through the premises triggering all the installed sensors one by one. A report will be generated and viewable by the user, showing which sensors responded and which failed to so. This allows both the system and the user to spot any faulty hardware and fix/replace it if necessary.