Community Gate Keypad Project

Software Architecture Design Document

SAD Version 1.0

Team T02

02 March 2021

Tanner Evans (manager)
Thomas Bowidowicz (documents)
Robin Acosta
Marcos Lopez
Jared Bock
Jacob Varela

CS 460 Software Engineering

TABLE OF CONTENTS

Introduction

1.1 Purpose

The purpose of this document is to create a high level outline of the software's functionality as well as illustrate the relationships between components and how those components interact during operation. This document will also outline the constraints and limitations of these same components to make clear the role each one has.

Section two covers the design overview of the project which includes a description of the approach used to come up with the design, the design diagram which details the software, and a description of the various elements of the design diagram. In section three, we give a detailed description of the components of the project including enums for the state of the program, the type of sound played, and identifier for the keys. Also described in section three are the classes that will define the keypad, speaker, and gate interfaces as well as a driver class and the subsequent methods in each class. Section four is a sample use case where we describe the possible outcomes based on the input as and the scenario in which the keypad is being used. Section five covers the design constraints placed on this program.

1.2 Project goals

The goal of this project is to create a community keypad that will restrict access to specific users which will include residents, community workers, and public services. It is intended to be easy to use and secure (from a community standpoint and a software standpoint). The client of this keypad and software hope to implement this into gated communities across the country.

Design Overview

2.1 Design approach

The keypad system is designed according to the Actor model. In this model, input received from a user interacting with the keypad is treated as a message. In order to handle asynchronous input events we send all input messages into a message queue. This means that when a user hits a button on the keypad that message is sent to a message queue. The messages in that queue are then interpreted by our main class, CommGateDriver.

We have implemented several enumerator classes in order to define a collection of constants that relate the different components of our system. These components help define the logic of the software and allow the program to react differently to received messages depending on the system's current Mode and State enums. The main class has

sole control over its Mode and State. Each component has a minimum number of public functions in order to minimize the risk of one component altering the data of another component.

2.2 Design diagrams

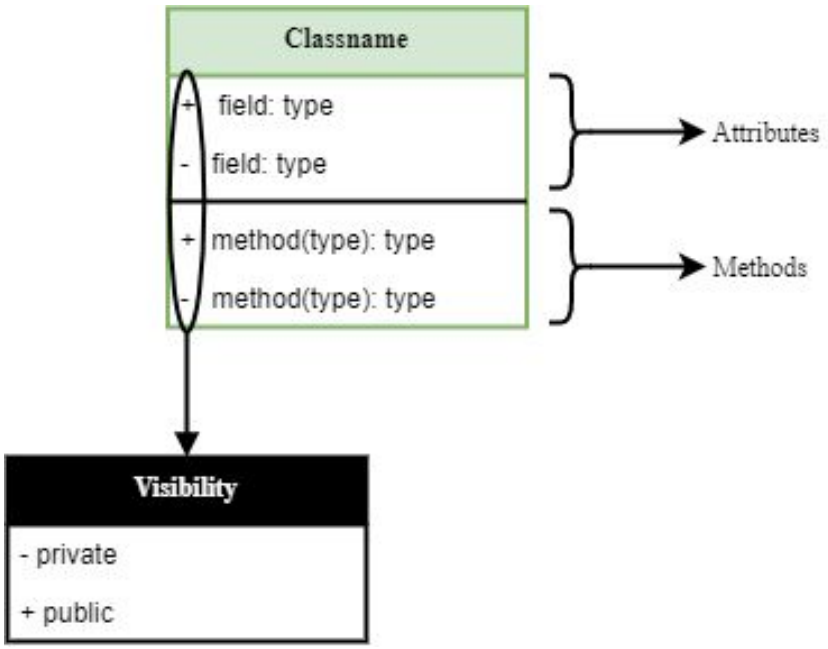

Figure 1 - Class Legend

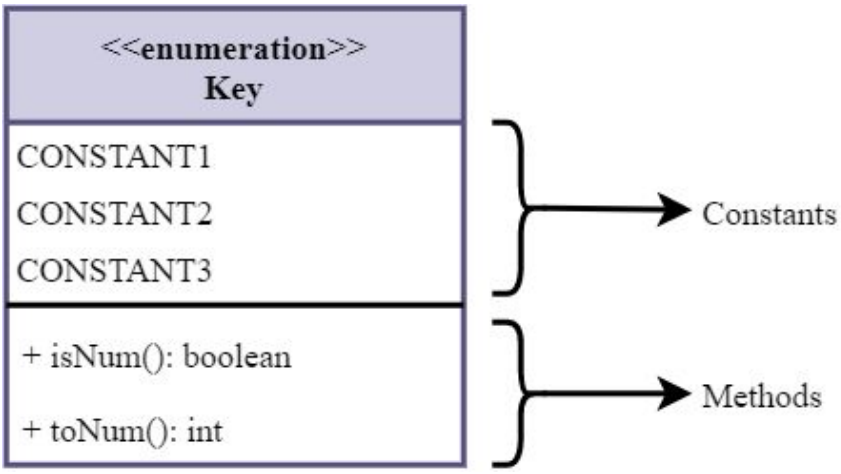

Figure 2 - Enumeration Class Legend

Both Figure 1 and Figure 2 showcase the layout for the normal class and enumeration classes in the design diagram. Each class may have any number of attributes, constants, or methods and they will be represented by a '+' or a '-' symbol to denote if that component is publicly visible or private. Note that the enum classes will have predefined constants, while

the normal classes will have public or private attributes. Attributes will have a name field, as well as a type field to indicate what type of variable that attribute is. A method will have a method name, a return type, and indicate the variable type of any input variables it may have.

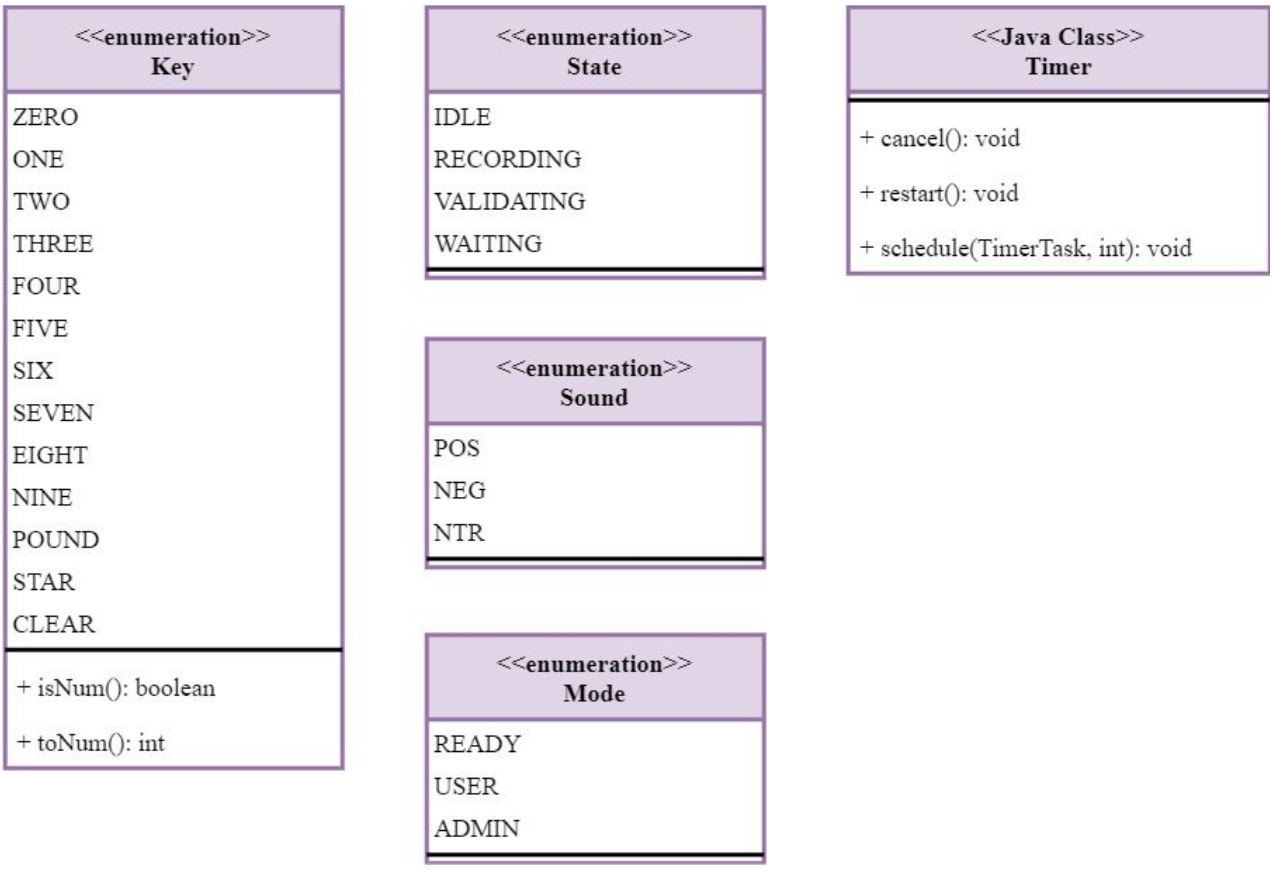

Figure 3 - Enumeration Classes

Figure 3 depicts the enumeration classes our software will have, as well as any components or methods associated with those enumeration classes. Note that this figure is not a depiction of the relationship between the enumeration classes and normal Java classes, that is represented in Figure 4.

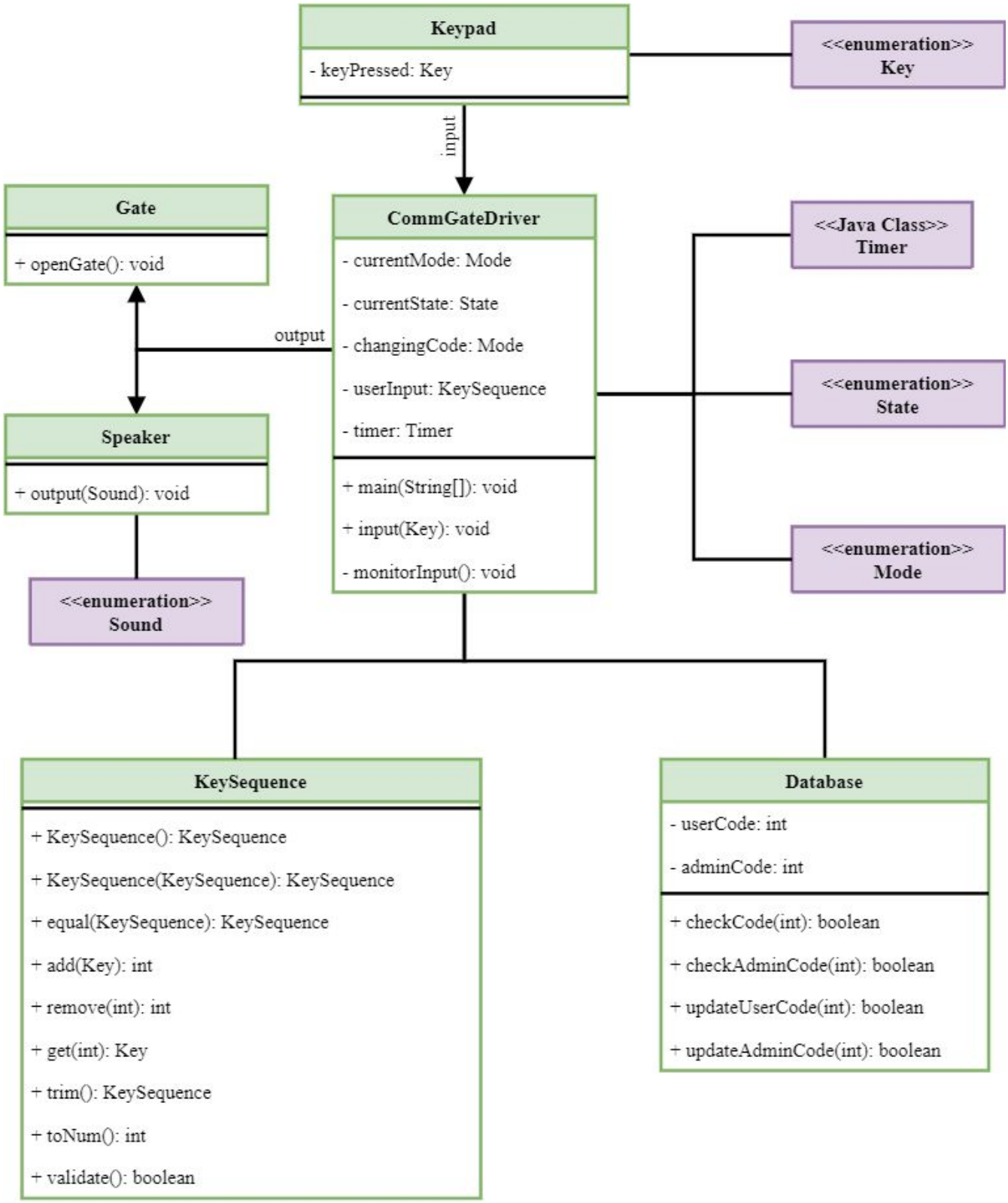

Figure 4 - Class Diagram

Figure 4 represents a class diagram for the keypad software. Note that the enum classes do not list their constants or methods in this figure, this was done to save space as those elements can be viewed fully in Figure 3. The CommGateDriver is the main class of this software. As keys are entered on the keypad they will be interpreted as input messages. Based on the current Mode and State that the system is in, the messages will be processed appropriately. The main class also starts a timer when a new key is pressed. This timer is used to automatically return to an idle state in the ready mode so that the system's message queue does not get bogged down by old messages.

The KeySequence class contains many methods that allow the software to convert the entered key buttons to integer values that we can manipulate. It also builds the full code from the individual keys entered, based on the order the messages are received in the message queue.

The Database stores the most up-to-date version of the userCode and the adminCode, as well as related methods to check or update these codes.

If the user entered a correct code to unlock the gate then an output signal will be sent from the CommGateDriver class to call the openGate() method. The CommGateDriver will also send output signals to the speaker interface to play an appropriate tone for audio-feedback to the user.

Component Specification

This section describes various top-level components of the software. Since the software is being implemented in Java, we will use Java-specific terms for the various components. The program will utilize classes and enumerators as its top-level components.

CommGateDriver is the main logical component, and therefore its general internal implementation has been outlined. However, the other component descriptions detail only the required functionality of the component in relation to other top-level components, including all functions accessible to other top-level components (excepting constructors of no arguments with no special initializations). As long as this functionality is supported, the internal implementation of these classes is left to the programmer to decide.

Programmers are encouraged to create helper components as needed to implement their components. As such, additional components may be included in the final implementation.

3.1 Enumerators

To simplify communication and abstraction, enumerator constants will be passed between classes where information must be exchanged. There will be an enumerator containing constants for each of the keys, and an enumerator containing constants for each of the speaker output types. The gate interface does not require an enumerator, since its public function needs no information.

3.1.1 Key Enumerator

The Key enumerator will provide constants for each key supported by the system. The Keypad interface must map the specific input information obtained from the keypad hardware to one of the constants. Hereafter, Key will refer to a constant in this enumerator. Java supports the comparison of enumerator constants by means of the '==' operator, as well as the inherited .equal() method, and these should be sufficient for our needs.

Non-Private Functions:

`boolean isNum() (non-static)`

Returns a boolean indicating whether the Key it is called on represents a digit.

`int toNum() (non-static)`

Returns the integer representation of the Key it is called on.

Key	Constant	isNum()	toNum()	Key	Constant	isNum()	toNum()
0	ZERO	true	0	7	SEVEN	true	7
1	ONE	true	1	8	EIGHT	true	8
2	TWO	true	2	9	NINE	true	9
3	THREE	true	3	#	POUND	false	-1
4	FOUR	true	4	*	STAR	false	-1
5	FIVE	true	5	Clear	CLEAR	false	-1
6	SIX	true	6				

Table 1 - Key Enumerator Summary

3.1.2 Sound Enumerator

The Sound enumerator will provide constants for each sound output by the system. The Speaker interface must take any of the Sound constants and communicate with the speaker to produce the associated sound. Hereafter, Sound will refer to a constant in this enumerator.

This enumerator has no non-private functions.

Constant	Output Sound
POS	A positive feedback sound.
NEG	A negative feedback sound.
NTR	A neutral feedback sound.

Table 2 - Sound Enumerator Summary

3.1.3 Mode Enumerator

The Mode enumerator provides constants for indicating the mode the main program is in. It is used by CommGateDriver in conjunction with the State Enumerator to track and update the state of the program, allowing a more streamlined process for determining what to do with input.

This enumerator has no non-private functions.

Constant
READY
USER
ADMIN

Table 3 - Mode Enumerator Summary

3.1.4 State Enumerator

The State enumerator provides constants for indicating the state the main program is in. It is used by CommGateDriver in conjunction with the Mode Enumerator to track and update the state of the program, allowing a more streamlined process for determining what to do with input.

This enumerator has no non-private functions.

Constant
IDLE
RECORDING
VALIDATING
WAITING

Table 4 - State Enumerator Summary

3.2 Classes

The project's classes will consist of several interfaces to external systems, a main CommGateDriver class, and several utility objects.

3.2.1 Keypad Interface

The keypad interface maps specific data received from the keypad to a Key, in accordance with Table 1. When it receives input information, it calls the input() function of the CommGateDriver class with the appropriate Key as an argument.

Since input is received asynchronously, this class will need to run as a separate process for the duration of the program. As such, it will need to be a runnable, non-abstract class, and it will be instantiated by CommGateDriver when the program is loaded.

This class has no non-private functions.

3.2.2 Speaker Interface

The speaker interface maps sound constants to the specific processes needed to instruct the speaker to output the intended sound.

Non-Private Functions:

`void output(Sound)`

Takes a Sound and causes the speaker to emit the associated sound.

3.2.3 Gate Interface

The gate interface implements the specific process required to instruct the gate to open.

Non-Private Functions:

`void openGate()`

Instructs the gate to open.

3.2.3 Key Sequence

A Key Sequence is a utility object which stores a sequence of Keys. It provides several utility functions.

Non-Private Functions:

`KeySequence KeySequence()`

Constructs and returns an empty KeySequence.

`KeySequence KeySequence(KeySequence)`

Takes a KeySequence and returns a copy.

`boolean equal(KeySequence) (non-static)`

Tests if the KeySequence it is called on and the provided KeySequence are equal.

`int add(Key) (non-static)`

Appends the provided Key to the KeySequence it is called on, and returns the number of Keys that KeySequence now holds.

`int remove(int) (non-static)`

Removes the Key at position int in the KeySequence is it called on, and returns the number of Keys that KeySequence now holds.

`Key get(int) (non-static)`

Returns the Key at position int in the KeySequence is is called on.

`KeySequence trim() (non-static)`

Returns a KeySequence of the first four Keys of the KeySequence it is called on.

`int toNum() (non-static)`

Returns the integer translation of the KeySequence it is called on. Returns -1 if it contains non-number Keys.

`boolean validate() (non-static)`

This returns true if the KeySequence it is called on is comprised of four number keys.

3.2.4 Database

Database is the class which accesses the stored user and admin codes. The codes will be stored in a secure format that persists when the program ends, accessible only to itself. Its functions will be access-restricted to the program.

Non-Private Functions:

`boolean checkCode(int)`

Returns true if the provided integer matches the user or admin code, and false otherwise.

`boolean checkAdminCode(int)`

Returns true if the provided integer matches the admin code, and false otherwise.

`boolean updateUserCode(int)`

Updates the user code to the provided integer.

`boolean updateAdminCode(int)`

Updates the admin code to the provided integer.

3.2.5 CommGateDriver

CommGateDriver is the central class of the program. It contains the main method, and tracks the modes and states of the program. As such, its implementation is described in more detail than others.

Inner Classes:

`Timer`

Used by CommGateDriver to trigger a timeout when input ceases for too long.

Required functionality:

- Runs in parallel to main program
- Can be cancelled with non-static method
- Length specified in number of seconds in constructor
- Passes Key.CLEAR to input function of CommGateDriver when done

Variables:

```
Mode currentMode := Mode.READY
State currentState := State.IDLE
Mode changingCode
```

Used to mark which code is being changed when currentMode is ADMIN and currentState is RECORDING.

`KeySequence userInput`

Where the user input sequence is stored. Accesses should be atomic.

`Timer timer`

The current timer object. Whenever a new timer is created, the old timer is cancelled, and this value is updated.

Functions:

The following functions are critical implementations. It is expected that a variety of functions will be implemented which abstract groups of actions when common and conceptually advantageous.

`main`

- Starts the Keypad Interface process.
- Initializes any necessary values.
- Calls the monitorInput() method once all setup is complete.

`private void monitorInput()`

Constantly and persistently monitors the state of the userInput KeySequence. When a value is added, checks currentMode, currentState, the value entered, and sometimes changingCode, to determine what to do with the input. A detailed outline of its decision-making is included separately in 3.2.5.1.

`public void input(Key)`

Takes a Key and appends it to the userInput KeySequence.

3.2.5.1 - Program Behavior Based on State:

The following pseudocode describes the major actions that the program will take when input is detected, given various markers of state and tests of the input. The testing of these markers will reside in the monitorInput() function, but it is likely many of the actions performed as a result will be abstracted into a variety of helper functions, which we have left to the programmer to implement as they choose. State transitions are specified with the Mode, State, and ChangingCode (sometimes) that are associated with the state being entered. The listed actions are not necessarily comprehensive, since changes of state may grow to require more complex behavior.

```
if Mode = Ready:
	if State = Idle:
		if input is a Digit:
			start Timer
			transition to Ready/Recording
		else:
			clear input
			continue
	if State = Recording:
		reset Timer
		if input is a Digit:
			continue
		if input is Pound:
			transition to User/Validate
		if input is Star:
			transition to Admin/Validate
		else:
			cancel Timer
			return to Ready/Idle
if Mode = User:
	if code entered is valid:
		unlock gate
		output Positive Feedback
		return to Ready/Idle
	else:
		output Negative Feedback
		return to Ready/Idle
if Mode = Admin:
	if State = Validating:
		if code entered is valid:
			start Timer
			output Neutral Feedback
			transition to Admin/Waiting
		else:
			output Negative Feedback
			return to Ready/Idle
	if State = Waiting:
		restart Timer
```

```
        if input is a Digit:
                continue
        if input is Pound:
                transition to Admin/Recording/Admin
        if input is Star:
                transition to Admin/Recording/User
        else:
                cancel Timer
                return to Ready/Idle
if State = Recording:
        restart Timer
        if input is a Digit:
                if four Digits have been entered:
                        if ChangingCode = Admin:
                                change Admin code
                        else:
                                change User code
                        output Positive feedback
                        cancel Timer
                        return to Ready/Idle
                else:
                        continue
        else:
                output Neutral feedback
                transition to Admin/Waiting
```

Sample Use Case

The purpose of this section is to walk the reader through a specific use case from the perspective of the user. This will provide context of how the system should work. The example case is informed by the architectural diagram and component specifications sections. This sample use case will follow a primary user of the keypad interacting with the keypad and unlocking the community gate.

4.1 Sample Use Case

Use case: Unlocking the community gate
Primary actor: A member of the local gated community to which this keypad is installed.
Goal in context: To unlock the gate to the community center and enter the premises.
Preconditions: The keypad and software have been installed on the gate and activated by a system administrator. The community has also been informed of the general user password.
Trigger: The user correctly enters the four digit password and presses the [#] key.
Scenario:

- The member of the community or user is returning to the community and approaches the locked gate and its accompanying keypad.

- The user approaches the keypad and presses the first key of the community password. No sound is emitted from the speaker.
- The user presses the second key of the password, again hearing no audible sound.
- The user presses the third key of the password, again hearing no audible sound.
- The user presses the fourth and final key of the password without hearing any sound from the speaker.
- Finally, the user presses the [#] key which confirms the password. The keypad begins to check if the password entered corresponds to the password that is stored in memory.
- The password is correct and matches the stored password. A positive sound is emitted from the speaker to indicate that the correct password was entered.
- The keypad sends the gate an unlock signal and the gate begins to open.
- The user walks through the open gate. As they clear the range of the gate, the gate's internal motion sensors determine that there is no one in the path of the gate.
- The gate closes itself and relocks upon closing.

Exceptions:

- The password is incorrect. In this case, the speaker emits a negative sound and the user would have to try and enter the correct password.
- The user forgets the password halfway through entering it and the system times out while they are trying to think of the correct digits to enter.
- The user fails to press the [#] pound key to confirm the password and the system times out after ten seconds.

Priority:

- This use case is the primary function of the gate keypad system and therefore is essential.

Frequency of use:

- This will be used many times per day as people enter the gated community. It will be in use every single day.

Channel to actor:

- The keypad buttons and speaker server as the interface with the user.

Design Constraints

This section lists the specific constraints that are placed on the design and implementation of the keypad and that has informed the design.

5.1 Performance

- The keypad will have thirteen buttons to enter the password and debug (the buttons are listed above).
- The passwords must be four digits in length.

- Once a valid code is entered followed by the [#] button, the keypad must immediately send a signal to the gate through a physical connection between the keypad and the gate. If a correct code is not entered, the keypad must remain locked.
- If the code is not completed, including the [#] button, within a few seconds the timer must reset and the keypad will clear and continue in ready/idle.
- Users must be able to clear mistakes and try again by pressing the clear button anytime before they press the [#] button. This will erase their current input and the keypad will remain in ready/idle.
- If a valid code has been entered, there will be positive feedback in the form of a sound. If an incorrect code is entered, there will be negative feedback.
- The administrator must have a unique code that when entered, will switch the keypad into the admin state, allowing the administrator to change the community code as well as the admin code. A valid admin code will result in a neutral feedback

5.2 Security

- There may be safety concerns regarding emergency personnel needing to gain access to the community. There should be a protocol in place for granting non-permanent members access.

5.3 Implementation

- The keypad must be implemented using Java 8 or higher.
- It must be implemented on a processing unit capable of running Java-compiled code.
- It must be able to handle up to twenty key presses a second.

Definition of terms

- **Actor Model:**
  - In response to a message it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received.
- **Admin/Administrator:**
  - A person or persons, who interact with the gate keypad with the intended purpose to access and modify the passcodes stored in the keypads memory.
- **Asterisk/Star:**
  - One of the keys on the keypad, denoted by the following symbol: [*]
- **Asynchronous**:
  - Not occurring at the same time.
- **Class:**
  - The blueprint from which individual objects are created in Java.
- **CommGateDriver**
  - Shorthand for Communication Gate Driver.
- **Enumerator / Enumeration:**

    - A special data type in Java that enables for a variable to be a set of predefined constants.
- **Gate:**
    - Any point of access to a walled community that the keypad will interface with.
- **Hashtag/Pound:**
    - One of the keys on the keypad, denoted by the following symbol: [#].
- **IsNum:**
    - Shorthand for “is Number?”.
- **Keypad:**
    - Physical hardware device with a set of buttons used to pass input into the software.
- **Member:**
    - A user.
- **Signal:**
    - A sequence or flow of messages passed between devices.
- **State:**
    - A representation of the system's position in a logical control flow. Specific states result from a series of inputs, and can be considered a representation of the data received up to that point.
- **System:**
    - A set of interconnected components, some of which are systems themselves.
- **toNum:**
    - Shorthand for “to Number”.
- **Transition:**
    - A representation of the system's movement from one state to another. Transitions are caused by input events or the completion of internal actions. They result in a change in how the system will handle new input.
- **User:**
    - A person or persons, who interact with the Gate Keypad with the intended purpose of opening the gate to gain access into the community.
- **Passcode/password/code:**
    - A specific four digit string of numbers that is defined to unlock the gate when entered.
- **Pseudocode:**
    - Simplified**,** non-functional description of the program code and logic.