PiClock

Software Design Document

Jacob Alspaw
Arbazuddin Ahmed
Kareem Taleb
Benjamin Young

An open source, smart alarm clock platform that aims to improve upon the functionality of traditional alarm clocks by integrating new and useful technologies to motivate users. PiClock increases productivity by helping users overcome their early morning exhaustion and providing general information about their upcoming day.

Version 1.1

October 17, 2018

History of Changes

Version	Description of change(s)	Initials	Date
1.0	Initial draft	BY, JA, KT, AA	10/14/2018
1.1	Removed News Widget due to time constraints	BY, JA, KT, AA	10/15/2018
1.2	Made Inspection Report	BY, JA, KT, AA	10/17/2018

List of Figures

1.Introduction

1.1 Purpose

This Software Design Document provides detailed guidance for the intended implementation of the PiClock by giving a complete and thorough overview of the PiClock's software architecture. It builds upon the Software Requirements Specification by describing how the software components of the PiClock satisfy its requirements and the rationale for the methods used. It then describes the major classes and their interfaces and the interactions between them.

1.2 References

PiClock Software Requirements Specification, 23 September 2018.

2. Design Overview

2.1 Software Requirements

The PiClock's software requirements are listed in section 2.2.2 of *PiClock Software Requirements Specification*. They are listed below for convenience.

F01:	Basic clock with time and date, present on every screen	
F02:	Alarm rings at time specified by user and continues ringing until user completes a task or failsafe is triggered	
F03:	User plays a simple game to shut off the alarm	
	F03.1:	Tic-Tac-Toe
	F03.2:	Concentration memory game
	F03.3:	Simon memory game
	F03.4:	Trivia game
	F03.5:	Math game
F04:	User can view the current weather conditions on home screen	
F05:	User can view the week's weather forecast on home screen	
F06:	User can view a random motivational quote on home screen	
F07:	User can view times in different cities around the world	
F08:	User can observe how many days until next major holiday	
F09:	User can view Raspberry Pi's system statistics	
F10:	Home screen rotates continually rotates between F04 - F09	
F11:	User can adjust application settings to change which game is to be completed when alarm rings, game difficulty, which applications appear on home screen, and API configuration	

2.2 Design Approach

PiClock is a UI heavy application and because of this will implement both procedural and event driven processes. Event-driven design is an appropriate approach when there is an external source of events to which the application must respond. At any time, PiClock users and the application itself can supply an external source of events to which the application will need to handle. The PiClock developers will develop event-handlers and process coordinators to deal with eventing. The PiClock application will need to handle eventing whenever the user supplies touch input or an alarm begins sounding.

PiClock, however, cannot just be built upon event driven design concepts because of the requirements outlined in the Software Requirements Specification report. The PiClock development team will equally need to consider a procedural driven approach where the design is conceived of as sets of procedures. At all other times when eventing is not happening, PiClock is a continuously and sequentially updating system. Its widget processes will take steps towards updating their user interfaces through the use of fixed iterative steps within indefinite loops.

PiClock is going to be a complex application by the time its implementation has finished. Because the application is going to be complex, divide conquer design techniques will be a powerful tool for problem solving. To solve and address many of the requirements outlined in the Software Requirements Specifications report, the system's design will need to be recursively decomposed into a set of simpler design subproblems. Developers will be able to implement these subproblems easier once the design is broken down into lower-level components. For example, Pi Clock will be broken into three levels to separate complex ideas: application, coordinators, and components. These ideas will be discussed in more detail below. The PiClock development team leader will then be able prioritize low-level subproblems and assign them to multiple developers.

The PiClock's software will be designed using object-oriented programming (OOP) principles. The design is split into classes which serve specialized purposes. Each class encapsulates its implementation details behind an external interface. Classes will access other classes' methods only by the accessed classes' interfaces, independent of how those methods are implemented. Hiding implementation information in this way reduces the coupling between classes because changes to the implementation of a class' methods will not necessitate changes to the classes which access those methods. OOP also promotes cohesion within the PiClock's software architecture, as related classes, such as the different games and widgets, are grouped as subclasses of a common parent class.

2.3 Principal Classes

2.3.1 PiClockApp

The PiClockApp is the main application class that will act as a frame on which all other process rely. It is the highest-level class and is not directly responsible for any of the above requirements. This process will act as the main parent process to all other managed processes that the application will need to invoke. It will contain the *main* method to the application that will be invoked to create and start the application.

The process will initialize and manage the user interface (UI) and give inter-agent services access to specific components in the UI. Hiding the system's user interface structure from other agents promotes encapsulation of processes by limiting the user interface components to which they have access. The main application process will also be responsible for putting the application to sleep by controlling the touch-screen's backlight. The PiClockApp class will need to put the system to sleep by invoking system commands to toggle the backlight after the user has not provided input for a configured period of time.

2.3.2 Coordinator

A coordinator process is a class which will help facilitate communication between two different processes or components of the PiClock's application software. Each coordinator is responsible for checking and configuring the system and therefore have exclusive read access to the database. No other process will have read access to the JSON configuration file. They will also have the primary responsibility of initiating application subsystems, and monitoring the status of the system while simultaneously handling system errors. PiClock has three different types of coordinators that are further discussed in section 2.3.

2.3.3 Widget

Widget is an abstract parent class to the six widgets described in requirements F04 - F09, of which more detailed descriptions will be given below. The Widget class itself is responsible for initializing these widgets and updating them at regular intervals set within each widget's child class. Each widget will need to be able to initialize its components in the user interface and inject data into these components after regularly updating. If the widget encounters an error, it is expected to handle the error appropriately by calling an abstract method provided by this class, but implemented individually by the widget child class.

2.3.4 Alarm

Alarm is a class that will act as the embodiment of the clock's alarm and is responsible for facilitating communication between the PiClock software and the Piezo buzzer. When initialized, it will use the WiringPi library and establish a connection to the Raspberry Pi's GPIO pins. Once told to sound, the class will send a sine signal to the pin at which the Piezo buzzer is connected, so the buzzer will begin to ring. At the same time, a countdown will be triggered lasting only five minutes. Once the countdown has finished, the alarm will disconnect from the GPIO pins and stop emitting a signal. No alarm will last longer than five minutes because of the design alarm failsafe. If the alarm is told to stop sounding, it will bypass the countdown and stop sounding immediately.

2.3.5 Game

Game is a class responsible for requirement F03. Like Widget, Game is an abstract parent class, in this case to the five games listed in requirements F03.1 - F03.5. Game is responsible for control and play of the alarm games and implements methods which all the games have in common, such as initialization, starting the game, adding points to the user's score, and determining if the user has won, and ending the game. If the game encounters an error, it is expected to handle the error appropriately by calling an abstract method provided by this class, but implemented individually by the widget child class.

2.4 Inter-Agent Processes

2.4.1 Settings Manager

The Settings Manager process controls the PiClock's user settings and is responsible for requirement F11. It will be responsible for allowing users to toggle which widgets appear on the home screen slideshow, and to set up API configurations and timeout of the touch-screen's backlight. It also allows users to set game difficulty and which game is played when the alarm rings, and to set alarm the time at which the alarm will trigger. It saves the settings modified by the user and applies those settings when the user clicks save by reading and writing data to a JSON configuration file. To ensure data integrity of the JSON configuration file, the settings database will be fully rewritten every time the user requests a save. The process will then need to restart the system, so other processes can use the new settings.

The settings manager process is responsible for a good portion of the application's initialization. Each of the application's managed processes are reliant off of how the settings manager operates. Because the settings manager is responsible for setup and initialization of the entire PiClock system, it must know the overall structure of the system. It is the only process that should have write access to to any of the configuration information. Hiding this information is important because it ensures that all processes are encapsulated within their own operation and functionality, meaning that one process will not be able to affect the ways in which any other processes initializes or operates. Therefore, the settings manager must be capable of reading data from the user interface, configuring that data into a reusable format by other processes, writing the data to a configuration file, and restarting the system using system commands.

2.4.2 Widget Manager

The WidgetManager class is responsible for control of PiClock's widget processes and facilitates each widget's positioning and display in the user interface. It rotates the slideshow between widgets and thereby satisfies requirement F10. It decides which widgets to initialize for inclusion in the slideshow. The process will read the application settings once to use for determining which widgets are enabled. It then delegates which widget is assigned to each slide container in the user interface and then initializes the widget processes in the context of that container using the application settings. The process manager further limits user interface access to each widget by passing in a refined user interface reference during widget initialization to widget-specific components only. The process manager will continue to rotate the slideshow indefinitely or until the main application process is killed. It will do so by regularly hiding one widget container and showing the next widget container in the sequence. Each slideshow sequence will be in the form of a circle queue to ensure that all widgets are shown in consistent and regular intervals without giving preference to one widget over any other.

2.4.3 Alarm Manager

The AlarmManager class controls the behavior of the alarm to satisfy requirement F02. The class will check for an alarm at regular intervals and triggers the alarm to ring at the user-specified time. It starts the preconfigured game and ensures that the alarm only stops ringing when the user completes the game or the failsafe is triggered. Once the user has completed the game, the AlarmManager will end both the game process and the alarm process, and then repeat the process of waiting for another alarm to trigger.

3. Class Interfaces

This section details the interfaces of the PiClock's primary classes and their subclasses. These lists are likely to be subject to revision in the future.

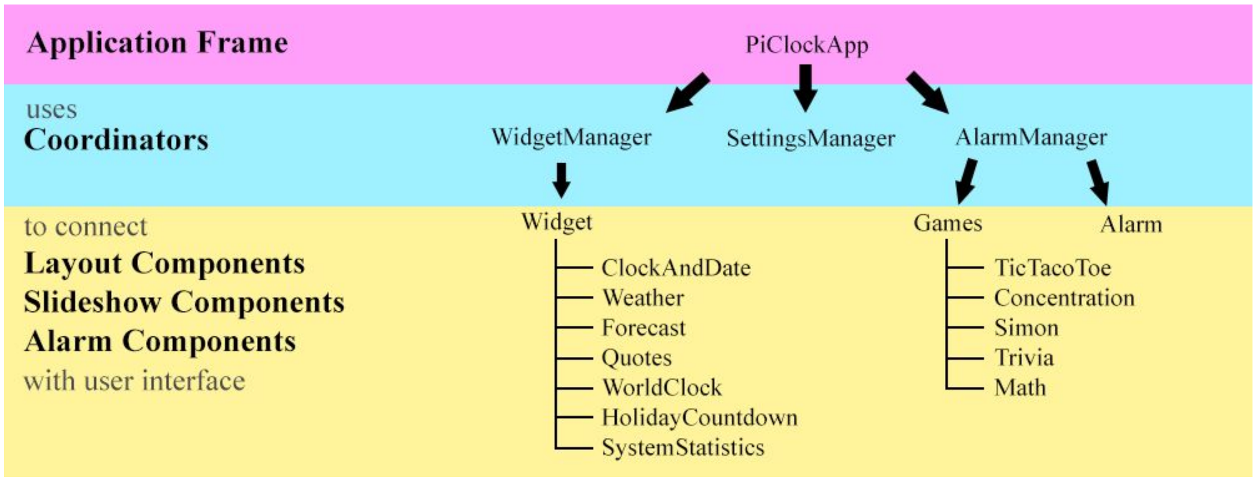

Fig. 1: PiClock class diagram

3.1 Class PiClockApp

PiClockApp is the highest-level class in the PiClock software architecture. It will be responsible for handling the XML that is the structuring component of the user interface.

- *public void main()*
  The application's primary main method. This method initiates and starts the application.

- *public void initializeUI()*
  Initializes the PiClock's user interface, which is accessed by the user via the touch-screen.

3.2 Abstract Class Widget

Widget is the parent of classes 3.2.1 - 3.2.7. It outlines and implements methods which are common to all widgets.

- *public abstract void Init()*
  Initializes the widget program. Each type of widget provides its own implementation

- *protected abstract void Update()*
  Refreshes the information displayed by the widget. Each type of widget provides its own implementation.

- *protected void OnError(exception)*
  Hides all user interface components of this widget and instead displays an error message that corresponds to the system error. This method will not affect the overall behavior of the widget, but is intended as a way to handle unexpected error conditions. The next call to the *Update()* method will reveal any hidden user interface components belonging to the widget as long as *OnError()* has not been called again.

3.2.1 Class ClockAndDate extends Widget

This widget displays the time and date of the day.

- *public void Init()*
  Will create a loop that calls the *Update()* method once every second.

- *protected void Update()*
  Retrieves the current time and date according to the system and updates the time and date labels in the user interface.

3.2.2 Class Weather extends Widget

A widget that displays the weather of the current day.

- *public void Init()*
  Will create a loop that calls the *Update()* method once every minute.

- *protected void Update()*
  Retrieves the current local weather using an HTTP request to a third party API. The response data will be parsed and used to update the weather specific components of the user interface.

- *private void gatherWeatherData()*
  Will format an HTTP GET request header and send the request to the required endpoint by using a socket and TCP connection. The method will read back weather data into memory.

- *private void CalculateGraphic()*
  Determines the infographic that should be used in the user interface based off of the current weather data retrieved from the API.

3.2.3 Class Forecast extends Widget

A widget that displays the weather for the next four days. An array will be used to store information about each day's weather data.

- *public void Init()*
  Will create a loop that calls the *Update()* method once every minute.

- *protected void Update()*
  Retrieves the current local forecast for the next four days using an HTTP request to a third party API by calling *GatherForecastData()*. The response data will be parsed and used to update the forecast specific components of the user interface.

- *private void GatherForecastData()*
  Will format an HTTP GET request header and send the request to the required endpoint by using a socket and TCP connection. The method will read back forecast data and parse it into memory using an array of JSON strings.

- *private void CalculateGraphic()*
  Determines the infographic that should be used in the user interface based off of the suspected weather data for each day retrieved from the API.

3.2.4 Class Quotes extends Widget

The inspirational quote widget will display randomized quotes from a variety of authors to inspire early morning motivation. An array will be used to store a list of unused quote-author pairs.

- *public void Init()*
  Will create a loop that calls the *Update()* method once every slide rotation.

- *protected void Update()*
  Updates the quote label every rotation so that the next time this widget appears, a different quote appears. Once the widget runs out of quotes to use, the update method will also call the *GatherQuoteData()* method to repopulate its list of unused quotes.

- *private void GatherQuoteData()*
  Will format an HTTP GET request header and send the request to the required endpoint by using a socket and TCP connection. The method will read back quote data and parse it into memory using an array of JSON strings. The request will retrieve many quotes at a time, so we can limit the overall number of requests made by the system.

3.2.5 Class WorldClock extends Widget

The clock's functionality will be extended as a widget that periodically displays times in six major cities around the world. The set of city-time pairs will be stored in a dictionary of string key-value pairs.

- *public void Init()*
  Will create a loop that calls the *Update()* method once every minute.

- *protected void Update()*
  Updates the world clock specific components of the user interface using the UTC offset calculations. Calls *CalculateTimeZones()* once every time before updating the user interface.

- *private void CalculateTimeZones()*
  Calculates the time in each of the supported cities using UTC offsets. The time in other cities is determined by using the difference in UTC from the current time zone to the new time zone and then applying that difference to the current time. The result is the time in the new city. Stores each result in memory as a dictionary of (city, time) pairs.

3.2.6 Class HolidayCountdown extends Widget

The Holiday Countdown widget will help users account for and track upcoming holidays by displaying the number of days until the next configured holiday. The widget is comes pre-packaged with major holidays with support through the year 2030. Holidays are stored in a JSON configuration file.

- *public void Init()*
  Will create a loop that calls the *Update()* method once every minute.
- *protected void Update()*
  Updates the holiday countdown specific components of the user interface using a list of prepackaged holiday dates stored in a JSON file. Calls *CalculateCountdown()* once every time before updating the user interface.
- private void *CalculateCountdown()*
  Reads the holiday JSON configuration file into memory and then finds the name and date of the next upcoming holiday. Compares the next upcoming holiday's date with the current date and then calculates a difference in days. The next holiday and days until that holiday are stored in memory as strings.

3.2.7 Class SystemStatistics extends Widget

The system usage statistics widget will display information about the user's Raspberry Pi system under load.

- *public void Init()*
  Will create a loop that calls the *Update()* method once every second.
- *protected void Update()*
  Updates the system usage statistics specific components of the user interface. Calls *GatherUsageStats()* once every time before updating the user interface.
- *private void GatherUsageStats()*
  Uses system calls and POSIX API to determine CPU and memory consumption from all running processes on the machine. Returned values will be stored in memory as floats.

3.3 Abstract Class Game

Game is the abstract parent of classes 3.3.1 - 3.3.5. It outlines and implements methods which are common to all games.

- *public abstract void Init(difficulty)*
  Initializes a Game object with input difficulty. Each type of game provides its own implementation.
- *public abstract void Start()*
  Loads game and displays it on the screen. Prompts user to begin playing the game.
- *protected void AddPoints(numPoints)*
  Adds numPoints to the user's total. Based on the game's scoring system, the user can never lose points. The method is triggered by an event in the game which causes the user to gain points.

- *public boolean IsWinner()*
  Is called after every call of *AddPoints().* Returns true ff the number of points accumulated by the user exceeds the threshold determined by the game's difficulty, indicating that the user has won the game. Returns false otherwise

- *public abstract void End()*
  Terminates the game. Primarily called when a game's failsafe is triggered. Otherwise triggered when a user has determined to be a winner. The method is responsible for freeing all memory that a game allocates and returns the game to an "unplayed" state.

- *protected void OnError(Exception)*
  Hides all user interface components of the game and instead displays an error message that corresponds to the system error. In instances of unexpected game errors, alarms will continue to ring until the alarm failsafe triggers.

3.4.1 Class TicTacToe extends Game

This game and its rules will conform to the popular children's game Tic-Tac-Toe. The game is played by two players, *X* and *O*, human and computer. Each player will take turns marking the spaces in a 3x3 grid. The player who succeeds in placing 3 contiguous marks wins. The game will be shown as a 3 by 3 table where the user selects the specific square when it is their turn.

- *public void Init(difficulty)*
  Initializes a 3x3 array of characters into memory that is used to store the movements of computer and user. Sets the required number of points to win based off of the difficulty.

- *private void AutomatedSelections()*
  Computer uses O character as its letter. Randomly selects coordinates in the 3x3 array to place its letter. The coordinate that is chosen cannot already be occupied by another letter.

- *public void UserSelections(x, y)*
  User uses X character as its letter. User selections are determined by the button pressed. Method is called using the zero-based index of the pressed button. The corresponding position in the 3x3 array is then marked with an X. If the coordinate already occupies a letter, then the move is rejected and the user is asked to choose a different spot.

- *protected void AddPoints(numPoints)*
  Two points are awarded for a match win. One point is awarded for a match tie. No points are awarded for a match loss.

- *public boolean IsWinner()*
  Finds if 3 X's or 3 O's are placed in a row and says who won the game respectively. If all squares are filled with no winner, the game ends in a tie. Method is called each time the computer or user submit a move.

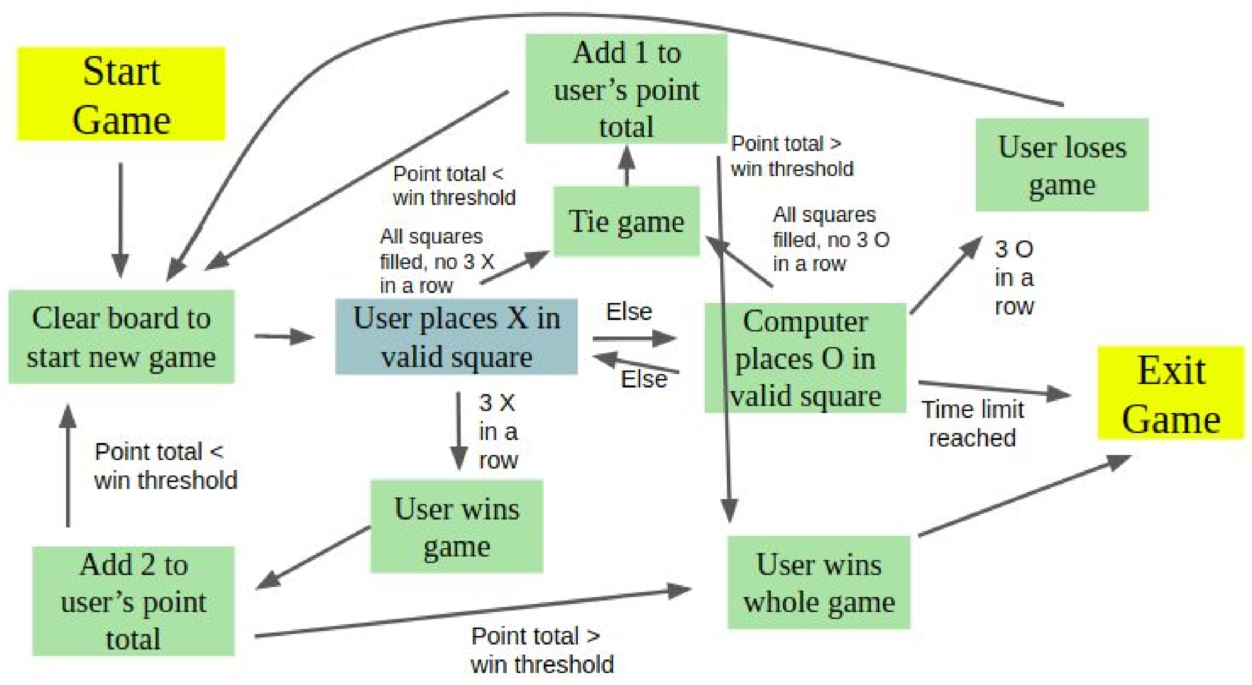
Start Game
Add 1 to user's point total
User loses game
Point total < win threshold
Point total > win threshold
All squares filled, no 3 X in a row
Tie game
All squares filled, no 3 O in a row
3 O in a row
Clear board to start new game
User places X in valid square
Else
Else
Computer places O in valid square
Time limit reached
Exit Game
3 X in a row
Point total < win threshold
User wins game
Add 2 to user's point total
User wins whole game
Point total > win threshold

Fig. 2: Tic-tac-toe activity diagram

3.4.2 Class Concentration extends Game

This game and its rules will conform to the popular matching card games. The game objective is to flip all the matching cards. Each turn, the user will select two cards, and if the cards match, then the user will have flipped the cards. Otherwise, the cards will be flipped back down and the user will get proceed to the next turn of flipping the cards.

- *public void Init(difficulty)*
  Initializes a 2x4 array of key-value pairs into memory that is used to store the positions of matching cards. The key for each pair is a GUID while the value is a boolean that tells if a card has been successfully matched or not. Each entry in the array will have a matching value. The *ShuffleCards()* method is then called to rearrange the cards into randomized locations. Sets the required number of points to win based off of the difficulty.

- *private void ShuffleCards()*
  The 2x4 array of cards are randomly ordered at different locations on the board.

- *public void FlipCards(x1, y1, x2, y2)*
  Flips two cards and checks if the cards are the same. If they are the same, then the card status for each is set true. If the card status of a card that is trying to be flipped by a user is true, the move is rejected and the user is asked to select two cards again. If two cards don't match, then nothing happens and the user is asked to select two cards again.

- *protected void AddPoints(numPoints)*
  One point is awarded for each new pair of matched cards.

- *public void IsWinner()*
  Determines if the game is over by checking if the number of matched cards on the board divided by two matches the number of required points determined by the difficulty.

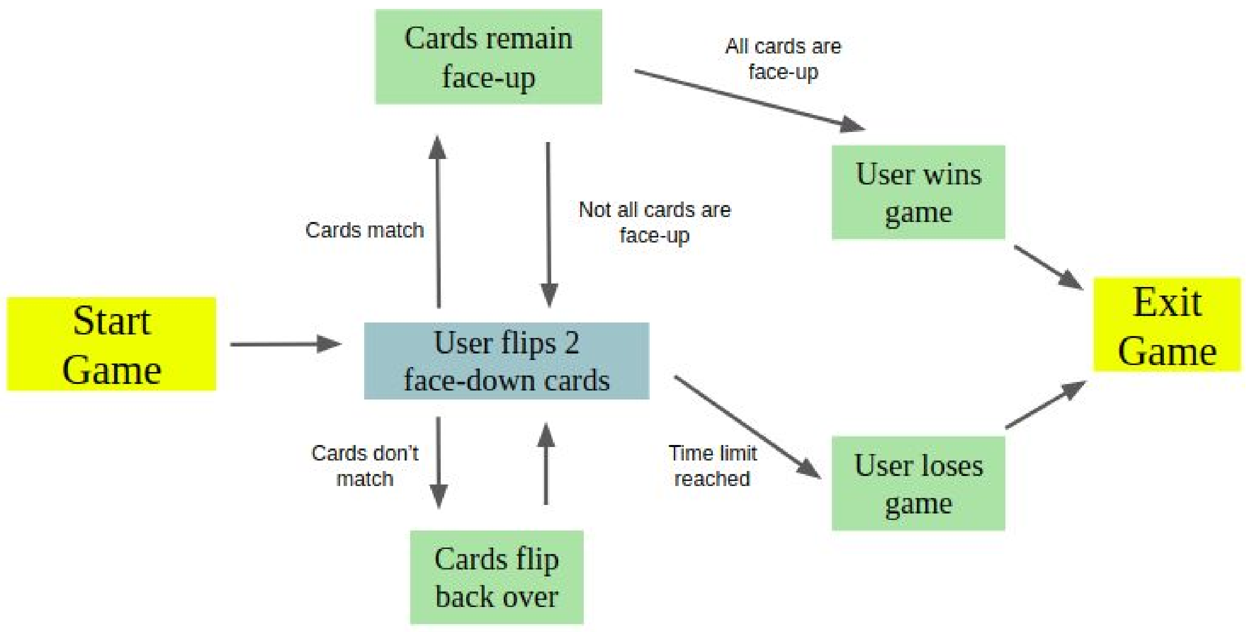

Fig. 3: Concentration activity diagram

3.4.3 Class Simon extends Game

This game and its rules will conform to the popular electronic game called Simon. Simon is a memory game in which the computer creates a series of button presses that the user is required to repeat. The buttons are illuminated in a sequence that the user must remember and then reiterate. Each round the sequence grows by one button press.

- *public void Init(difficulty)*
  Creates a linked list of moves to store the randomized sequence that the user must reiterate. The linked list is initially created with one move. Sets the required number of points to win based off of the difficulty.

- *private void NextLevel()*
  Appends a button press to the end of the sequence.

- *public void PressButton(colorCode)*
  Compares user input to the next expected button press in the sequence. Each button has a corresponding button color code that can be used to compare two button presses. If the user clicked the correct sequence, then the *AddPoints()* method is called. If the user did not click the correct sequence, the computer reiterates the sequence for the user and then allows the user to retry.

- *protected void AddPoints(numPoints)*
  One point is added to the user's score if they correctly reiterate the current sequence.

- *public boolean IsWinner()*
  Returns true or false based off of a win condition. Compares the user score to a value set in the *Init(difficulty)* method. The number of points required increases with increasing difficulty. If the user has passed enough levels, then return true.

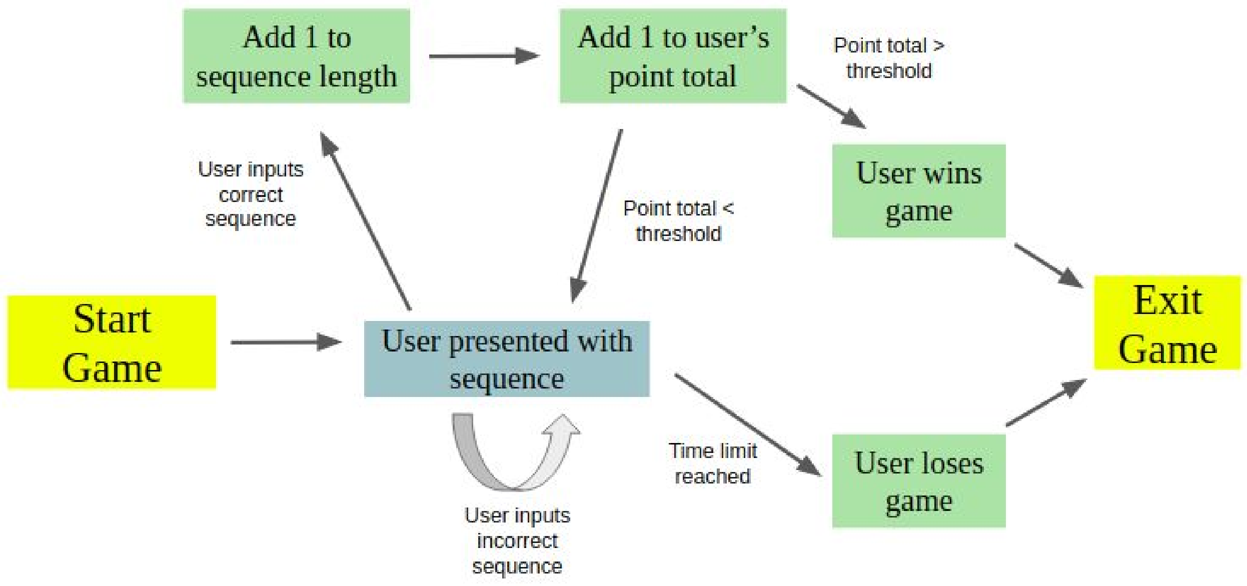

Fig. 4: Simon activity diagram

3.4.4 Class Trivia extends Game

In the trivia game, the user answers a sequence of randomly generated multiple choice questions. The game will ask the user several multiple-choice questions that the user is expected to answer correctly.

- *public void Init(difficulty)*
  Creates an array of questions in the form of JSON strings. Each question will have a bucket in the array. Sets the required number of points to win based off of the difficulty.

- *private void GatherQuestionData()*
  Will format an HTTP GET request header and send the request to the required endpoint by using a socket and TCP connection. The method will read back trivia question data and parse it into memory using an array of JSON strings. The request will retrieve many questions at a time, so we can limit the overall number of requests made by the system.

- *public string EvaluateChoice(int choice)*
  Retrieves the user input and evaluates it against the correct answer. If the user was correct, he is given a point and moves on to the next question. If the user was incorrect, the computer outputs the next question and does not give the user a point.

- *private void NextQuestion()*
  Displays the next question in the list. If the list of questions has been exhausted, then the

GatherQuestionData() method is called to repopulate the list.

- *protected void AddPoints(numPoints)*
  One point is added to the user's score if they correctly answer a question.
- *public boolean IsWinner()*
  Returns true or false based off of a win condition. Compares the user score to a value set in the *Init(difficulty)* method. The number of points required increases with increasing difficulty. If the user has answered enough questions correctly, then return true.

3.4.5 Class Math extends Game

The Math game and its rules will conform to a simple multiple choice questionnaire. The game objective is to correctly answer a set of simple arithmetic multiple-choice questions. The user will see the question and four boxes that include three incorrect answers and one correct answer.

- *public void Init(difficulty)*
  Sets the required number of points to win based off of the difficulty.
- *private void GatherQuestionData()*
  Randomly generates a question that follows the following format: "What is the answer to <integer> <operator> <integer>?" Corresponding multiple choice answers will also be made in this method call.
- *public void EvaluateChoice(int choice)*
  Retrieves the answer from the user and analyzes it against the correct answer. If the user was correct, this method moves on to the next question and gives the user a single point. If the user was incorrect, this method displays the correct answer and moves on to the next question.
- *private void NextQuestion()*
  Displays the next question. This game does not have a list of prepared questions, so it will need to call the *GatherQuestionData()* method before updating the user interface.
- *protected void AddPoints(numPoints)*
  One point is added to the user's score if they correctly answer a question.
- *public boolean IsWinner()*
  Returns true or false based off of a win condition. Compares the user score to a value set in the *Init(difficulty)* method. The number of points required increases with increasing difficulty. If the user has answered enough questions correctly, then return true.

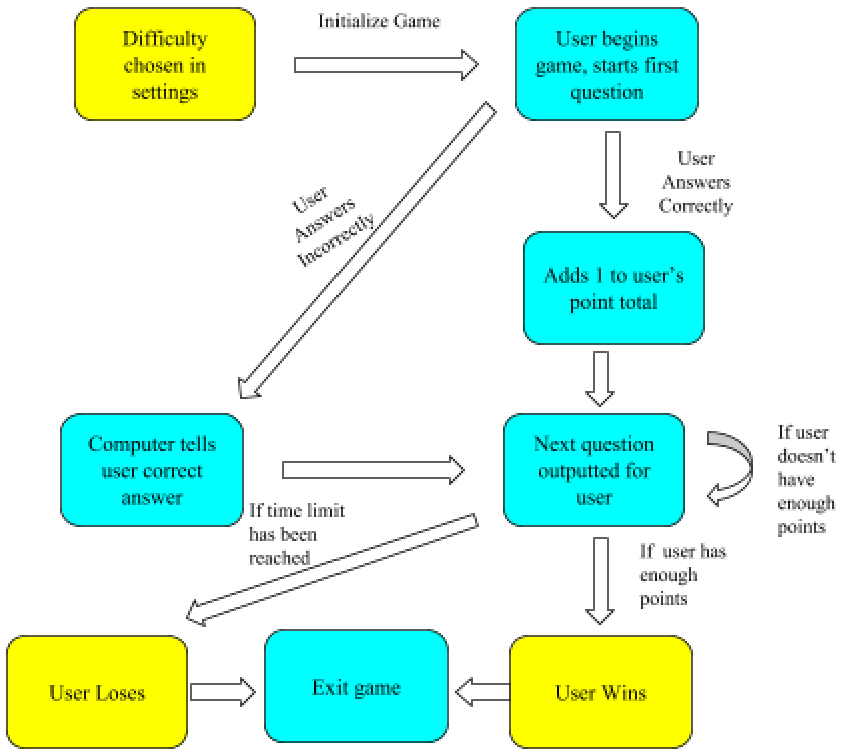
Difficulty chosen in settings
Initialize Game
User begins game, starts first question
User Answers Correctly
User Answers Incorrectly
Adds 1 to user's point total
Computer tells user correct answer
Next question outputted for user
If user doesn't have enough points
If time limit has been reached
If user has enough points
User Loses
Exit game
User Wins

Fig. 5: Math and trivia games activity diagram

3.4 Class Alarm

The user creates Alarm objects to ring at a specified time.

- *public void Init()*
  Initializes a new Alarm object. It creates a connection to the GPIO pins at which the Piezo buzzer is connected.
- *public void StartRinging()*
  Sine wave signal is sent to the Piezo buzzer in an indefinite loop.
- *public void StartCountdown()*
  Starts a countdown that will call the *StopRunning()* method after five minutes.
- *public void StopRinging()*
  Ends the loop which is repeatedly sending signals to the Piezo buzzer.

3.5 Abstract Class Coordinator

Coordinator is an abstract parent class to the three coordinator subclasses. All three types serve to coordinate the communication between different components of the PiClock's software.

- *public abstract void Init()*
  Each subclass provides its own implementation for initialization.

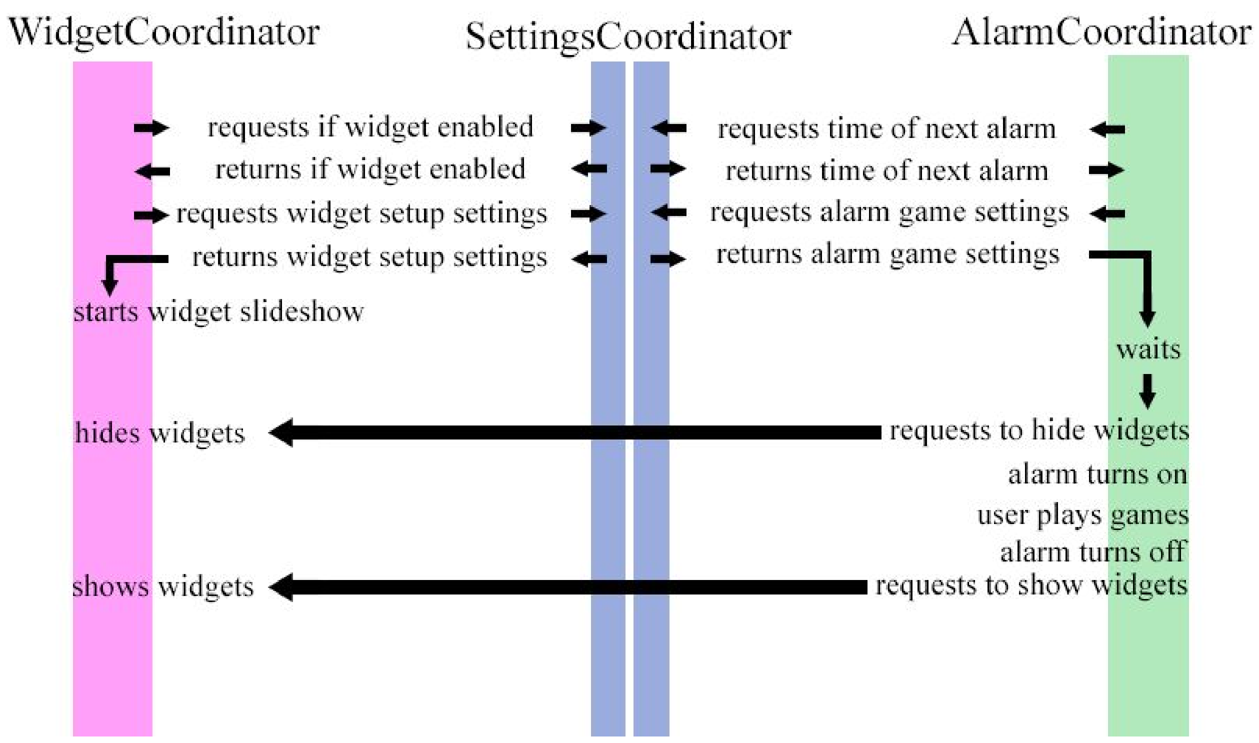
WidgetCoordinator
SettingsCoordinator
AlarmCoordinator
requests if widget enabled
returns if widget enabled
requests widget setup settings
returns widget setup settings
starts widget slideshow
requests time of next alarm
returns time of next alarm
requests alarm game settings
returns alarm game settings
waits
hides widgets
requests to hide widgets
alarm turns on
user plays games
alarm turns off
shows widgets
requests to show widgets

Fig. 6: Coordinator sequence diagram

3.5.1 Class AlarmManager Extends Coordinator

AlarmManager controls the PiClock's alarm

- *public void Init()*
  Initializes an AlarmManager object. Calculates how long until the next alarm in seconds. Puts the process to sleep until the alarm should be triggered using the calculated number of seconds. Calls the *StartAlarm()* and *StartGame()* methods immediately after the sleep call finishes.
- *private void StartAlarm(Alarm)*
  An alarm's StartAlarm() method is triggered when the clock reaches the alarm's time. It calls the Alarm's StartRinging() method to begin making sound and calls the startGame() method.
- *private void StartGame(Game)*
  StartGame launches a game. The type of game that is launched and the game's difficulty is determined by the SettingsManager
- *private void EndGame(Game)*
  EndGame terminates a game. AlarmManager calls this method when the game has indicated that it has been successfully completed, or the 5 minute countdown failsafe has been reached.
- *private void EndAlarm(Alarm)*
  EndAlarm() calls the Alarm object's *StopRinging()* method, causing it to stop making noise.

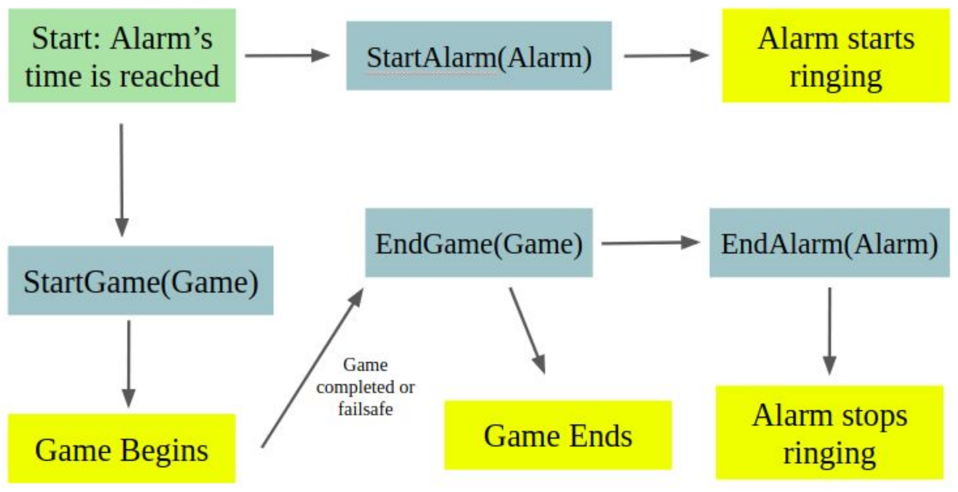

Fig. 7: AlarmManager activity diagram

3.5.2 Class SettingsManager Extends Coordinator

The SettingsManager class allows the users to change settings relating to the widget slideshow, games, and alarms.

- *public void Init()*
  Initializes a SettingsManager object. Loads and opens the configuration file for reading and writing.
- *public string ReadSettings(key)*
  Reads the settings from the configuration file and returns the values associated with the JSON string key.
- *private void SetGameDifficulty(int difficulty)*
  Sets the difficulty with which games are initialized.
- *private void SetGamePlayedOnAlarm(Game)*
  Defines which type of Game is to be initialized when the alarm goes off.
- *private void SetBacklightTimeout(int seconds)*
  Defines how many seconds of inaction must pass before the touch-screen's backlight turns off.
- *private void Save()*
  Saves the settings chosen by the user by writing them to a JSON configuration file. Restarts the system after the file has been saved and closed.

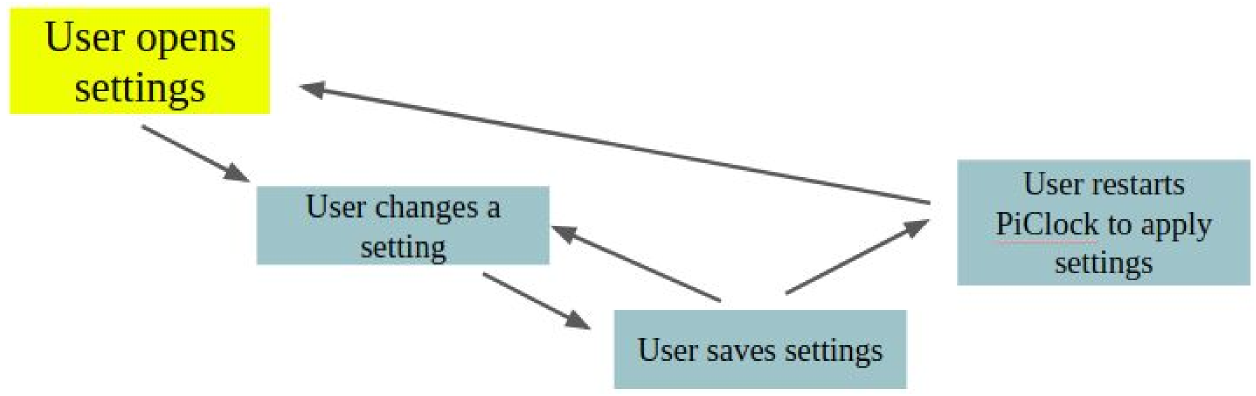

Fig. 8: SettingsManager activity diagram

3.5.3 Class WidgetManager Extends Coordinator

The WidgetManager class controls the widget processes and which widgets appear in the slideshow. It delegates widgets to slide containers and rotates the slideshow at set intervals.

- *public void Init()*
  Initializes a WidgetManager object. Upon initialization, the WidgetManager asks the settings manager for the application settings to determine which widgets should be included in the slideshow.
- *public void CreateSlideshow()*
  Creates the slideshow by sequentially ordering the Widgets to be included into containers and initializing these Widget's processes. The slideshow is a circle queue linked list, so the last item in the queue has reference to the first item in the queue. Calls the *AddWidgetToSlideshow()* method to add all widgets to the slideshow.
- *public void RotateSlideshow()*
  Progresses the slideshow to the next slide within an infinite loop.
- *private void AddWidgetToSlideshow(Widget)*
  If a widget of the type of the input Widget is not already set to be shown in the slideshow, set this Widget to be shown in the slideshow.

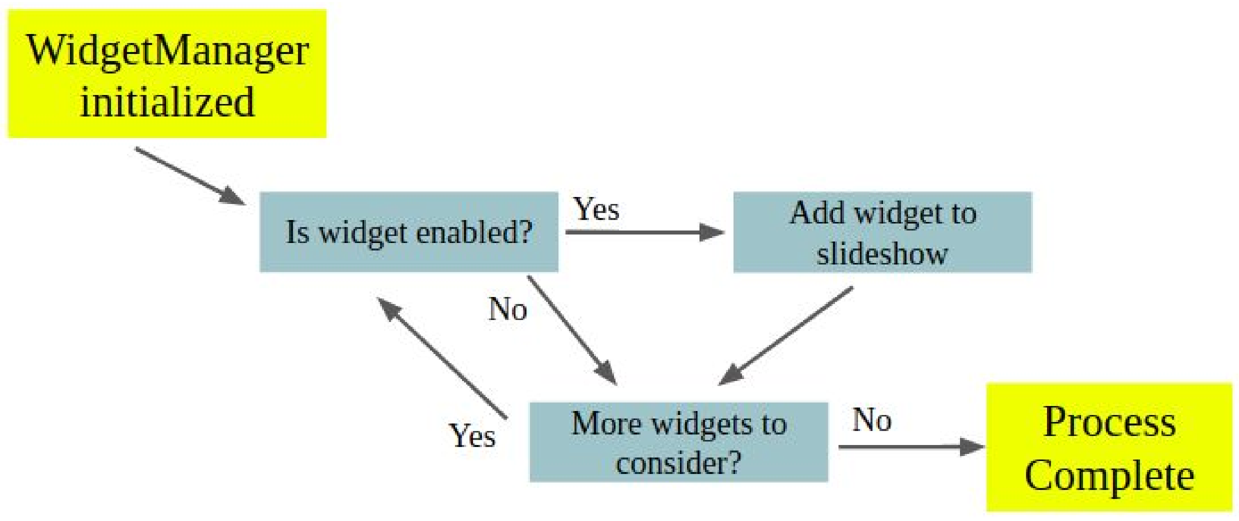

Fig. 9: WidgetManager.Init() activity diagram

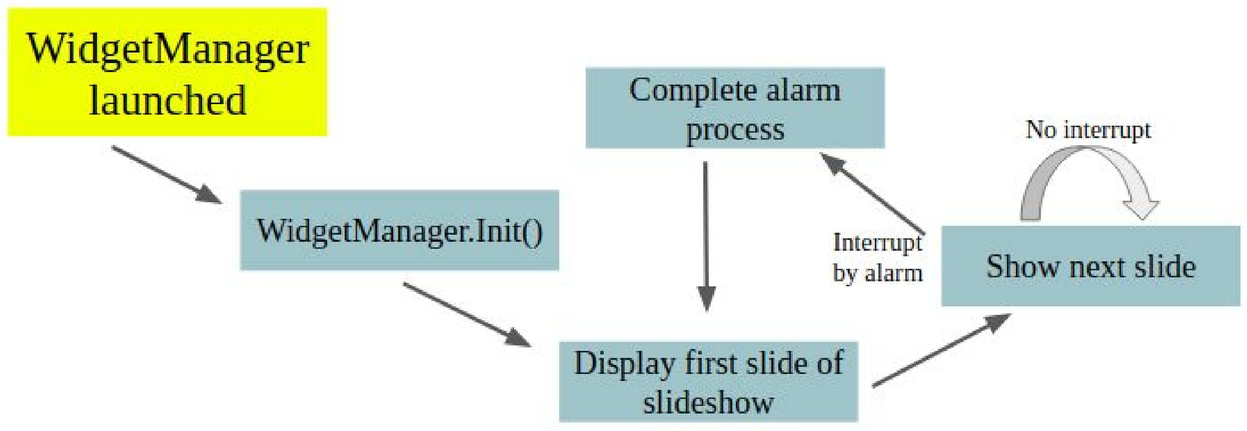

Fig. 10: WidgetManager activity diagram

4. Inspection Report

The inspection report describes the issues we as a team discovered throughout the construction of our design document and the resolution status of those issues.

Issue	Issue raised on date	Resolution status as of 10/17/2018
Document's communication could be improved by addition of more diagrams	10/17/2018	Resolved - Figures 6, 9, and 10 added
Implementing news widget did not seem feasible in time allotted to build the PiClock	10/15/2018	Resolved - News widget removed from design document
Section 3 did not describe interfaces in sufficient detail	10/14/2018	Resolved - Detail added to section 3.
Document did not include sufficient discussion of design approach and rationales for design decisions	10/14/2018	Resolved - Added section 2.2 - design approach, and added further discussion of rationale to sections 2.3 and 2.4