CSSE1001

23/05/13

Assignment 3 – Final Design Document

Student: Sean Manson

SID: 42846413

Project: GUI-based Riichi Mahjong implementation with A.I and Mahjong modules.

Description:

Riichi Mahjong is a variant of classic Chinese mahjong that is played primarily in Japan, and is based around a set of 136 tiles, which are divided into separate suits and numbers like cards. Unlike the commonly known solitaire versions of mahjong which are mainly focused around matching together similar tiles, this particular game is played by four players and has similar rules to the card game 'Rummy' (though, strictly speaking, this form of mahjong came first).

The rules of Riichi Mahjong are difficult to summarise, but the game essentially involves four players, each attempting to form a 'hand' of 14 tiles by drawing tiles from a 'wall' and then choosing a tile to discard at the end of their turn. In order to win a 'round', a player must form four 'melds' (a sequence of three tiles in the same suit, or a three/four of a kind of the same tile) and a pair of a tile. The hand is then scored based on what 'yaku' (special combinations of tiles, such as having all tiles in the same suit) the hand contains, and this amount is taken from the other players. Whoever has the greatest score at the end of the game (usually 8 rounds) wins.

This program is a visual implementation of Riichi Mahjong in Python using pygame. It is constructed from four major parts:

- A complex GUI using a combination of pygame and Tkinter that can take mouse and keyboard input and allows users to interact and play around with the game intuitively and effectively. This GUI has been programmed from scratch, and includes several animation classes and button elements which can be interacted with as a part of the menu system. This UI also has music and sound effects which are played back as a part of gameplay.
- A back-end element which is able to store all intricacies and algorithms used in the mahjong game, such as the involved scoring system or the actual game process containing rounds and each player's turn.
- Five different types of AI that the user can play against, each focused around achieving different purposes. While none of these AI can match up against an experienced player, they all can play the game at a basic level, imitating real actions and strategies that players may undertake, such as only discarding safe tiles, or attempting to hoard high scoring hands.
- Finally, there is also a fairly modular mahjong rules package which is independent of the actual game functions and UI. This package includes helper classes and methods defining tiles and melds, as well as methods for determining whether certain yaku have been fulfilled in a player's hand or whether or not players can call on specified tiles.

This program does *not* implement network features, and is wholly a singleplayer game played against the AI. It does, however, contain a fully working sound and music system (alongside volume controls) as well as a saving and loading system for users to keep their progress whenever they feel like it in the middle of play.

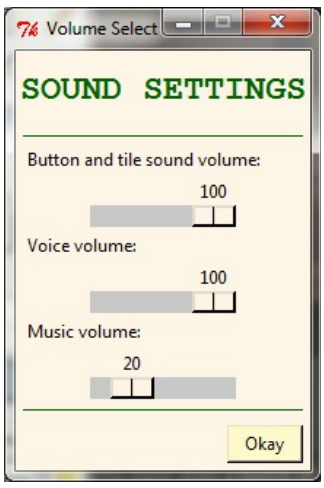
For more details as to the rules of the game, see here:

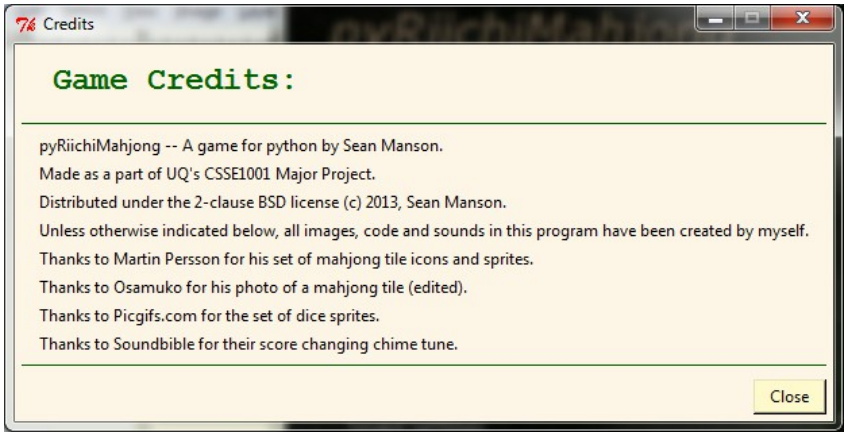
http://www.japanesemahjong.com/

User Interface:

Figure 1: Menu Screen

The menu screen (fig. 1) is what the user is faced with when first opening up the game. The text on the left are all Buttons (as defined in menuItems.py), which can be selected using either the keyboard or the mouse and can be clicked with the left mouse button or pressed with the enter key, playing appropriate sounds when doing so. The 'Load Previous Game' button attempts to start the main game using the user's currently stored save data. Pressing the 'Help' button tries to launch the user's default web browser and passes it the web page for the online mahjong help guide. 'Quit' exits the program, effectively the same as closing down the window. 'Sound Settings' pops up the Tkinter window shown in fig. 2 below, from which the user can change the main volumes for all audio controls. Similarly, 'Game Credits' pops up a window with a list of credits for the game, which is automatically loaded from a file called credits.txt in the game directory.





Figures 2 and 3: Sound Settings window and Game Credits window

Figure 4: Start New Game window

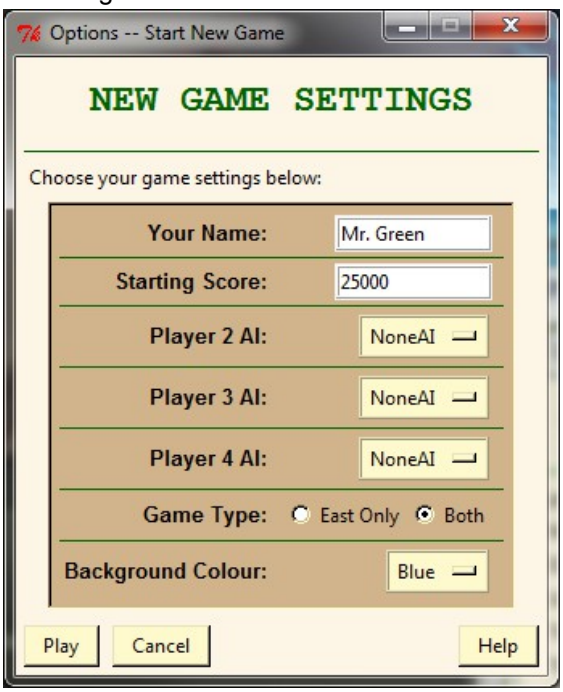


Fig. 4 on the left shows the new game window, which is displayed when the 'Start New Game' button is clicked.

This window allows players to enter their desired starting information, such as their name or how much score each player begins with. The list of AI that can be chosen from is determined using an external global variable with a list of AI classes, and can be chosen from using a Tkinter dropdown list. Game Type allows players to choose how many rounds the game lasts for through the use of a custom radio buttons class.

As you'd expect, 'Cancel' closes this window without doing anything, while 'Play' runs a new game in the main window using the data provided. 'Help' pops up a TopLevel Tkinter element which explains what each of these form terms mean to the user.

Figure 5: Main Game Screen

The main gameplay screen, shown in fig. 5 above, is displayed when the user starts up a new game or loads a previous game. It is made up of many pygame surface elements which automatically put together the graphical interface for the user to interact with.

The main screen part in the top left contains the tiles arranged as they would be in a real riichi mahjong game, displayed from a bird's eye view. The AI players have their hands in the left, top and right sides of this section, and inside this is the square of tiles for the wall, of which about a third is remaining. The darker tiles in this wall represent the dead wall, which are shifted downwards to differentiate them from the normal tiles. Closer to the centre, we have each player's discard pile—with a sideways tile in the user's discards representing when his riichi was declared—the dice and riichi sticks, and also a count of how many riichi/bonus sticks are remaining up for grabs from previous rounds.

Because this game is 2D, it was quite difficult to portray the wall, which is 2 tiles high, to the user in an intuitive way when looking from above. The workaround for this was to shrink the tiles in the wall when they were 'on the bottom', with the larger tiles on the top being 'closer' to the bird's eye view. For this to work, the tiles on the bottom also needed to be shifted along sideways to push against the next tile in the wall. The game screen automatically accounts for this in the dead wall as well.

The bottom part of the interface makes use of a series of buttons for the player's hand tiles. When it is the player's turn, they can click on one of these tiles to choose it for discarding. The buttons on the right, which are initially disabled unless the player can use them, allow the player to make various calls on discarded tiles as gameplay progresses. When these come up, gameplay pauses to wait for the player's response. If they do not wish to call on the tile, they can click on a 'Cancel' button or anywhere except the button, and the game will continue onwards.

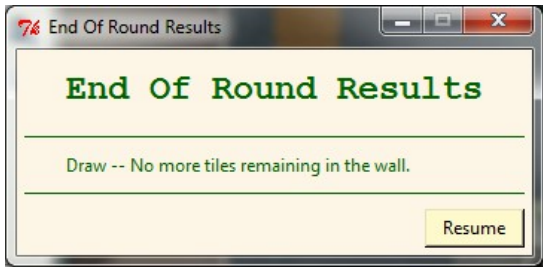
Figure 6: Game Menu window

The 'Menu' button, which only activates when it is the user's turn, opens up the window shown in fig. 6 when pressed. This window allows users to quit back to the main menu from the game, giving them the choice to save or not when doing so. Note that it is not possible to save and continue playing, though one may just save and quit and then load the save again immediately afterwards.

On the right and the bottom left are auto-generated pygame surface areas communicating useful bits of game information to the user. The player areas on the right change colour according to whose turn it is, and also display tiles showing which position the player is in (north, east, south or west).

As well as displaying these elements, this screen is also able to enact several kinds of animations. These include scrolling text down the screen, animating score changes for each player and fading the entire game area dark when necessary. Also, the game screen is able to play the appropriate sounds and voices when players discard, draw and call on tiles.

Figures 7 and 8: End of Round Results windows



The windows shown in figures 7 and 8 are displayed at the end of a round. Fig. 7 occurs when a player wins a hand, showing the hand they won with, the applicable dora for that round, as well as the fu, han and points that the hand is worth. This window includes an auto-generated list of yaku which the hand includes, along with how much han these yaku are worth. The names of these yaku can be pressed to pop up a description window explaining what they actually are for. Fig. 8 shows the window which is displayed on a drawn round between players.

Figure 9: Game End window

	Mr. Green	CPU Player 1	CPU Player 2	CPU Player 3
East First Round	24000	25000	25000	25000
(+- change)	10600	-9600	0	0
East Second Round	33600	15400	25000	25000
(+- change)	2900	0	0	-1900
East Third Round	36500	15400	25000	23100
(+- change)	3000	-1000	-1000	-1000
East Fourth Round	39500	14400	24000	22100
(+- change)	0	0	0	0
Final Total	39500	14400	24000	22100

The above window in fig. 9 pops up at the end of every game. It displays an auto-generated table of scores for each round of the game for each player, along with a breakdown of +/- changes in score over the playing period. This screen automatically accommodates for changes the amount of rounds that occur (due to including the South round or bonus dealer ante rounds), and simply stretches to fit them.

Design:

Because many of the private functions in these classes are often inseparable and meaningless when looked at individually, I have opted to simply break down each class by their *logic* and *purpose*, rather than spout off hefty amounts of useless information. I have, however, included descriptions of the less obvious functions in each class. For detailed and specific information about each individual class, I recommend viewing the module files themselves, as they contain a very comprehensive set of descriptions for each class and each method.

This program is run from a 'main.pyw' file, which then makes use of 14 custom module files, each of which fulfil one of two central objectives. Seven of these make up the gameplay and user interaction elements of the program, such as the pygame GUI, game logic and AI, while the other seven form a 'mahjong_rulebase' package containing definitions of core game class data structures such as tiles or yaku. The class breakdown for the central modules is as follows:

riichiMahjongApp.py:

Class RiichiMahjongApp(object)

Contructor RiichiMahjongApp()

This class is the main application object which defines the pygame display and clock, loading a 'MenuScreen' object onto this display. This class contains the main processing loop for the game under .run() which constantly is able to redraw elements to the screen and catch events such as mouse movements, key presses or resizing the window. This class is also responsible for the central music and sound channels, containing functions that can play certain sound effects and change music volumes.

As you'd expect, the _redraw() function here runs .returnSurface() on the current screen in order to obtain a Surface to copy to the game window. In addition to this, however, this function is able to cope with non-4:3 game resolutions by automatically generating black bars on the top and bottom or the sides when the window is resized.

playButtonSound(sound) and playVoiceSound(sound) are intriguing functions because they do not take actual pygame sound objects as their arguments. Instead, sounds are defined and loaded in the constructor of this class, and their variable names, as strings, are passed as arguments to these functions. While this is less secure than passing the sounds directly, it has the advantage of not requiring external modules to define and load their own sounds before playing them.

menuItems.py:

This file mostly focuses around defining central menu objects which can be subclassed for specific GUI elements. Each of them has a .returnSurface() method, which returns what they look like on the screen so that they can be blitted together for the final game window.

Class Button(object)

Constructor Button(master, xPos, yPos, func) where master is the app window that the button is associated with, (xPos, yPos) are the (x, y) position of the button and func is the function that is run when the button is pressed.

A button, in its distilled form, is a menu element that can be added to a 'MenuScreen' object at a certain position. Moving the mouse within the defined button area (set as 1x1 pixels for default buttons) causes the button to be 'hovered' and clicking on it causes it to be 'pressed'. These mouse movements are obtained by passing this class mouse movement event information. The function object passed to these buttons is run when the button is pressed, whether it occurs through mouse events, or through some external action defined in MenuScreen.

Classes ButtonText/ButtonImgText/ButtonInvis/ButtonTile:

Each of these are specialised subclasses of button with increased functionality, which can be seen through the greatly increased number of arguments they accept. For example, ButtonText allows users to provide a font and text to display as the button on the screen, which changes colour when hovered or clicked. ButtonTile is used as a part of the GameScreen class to have tile buttons which can be highlighted on command.

Classes AnimatedText/FadeInOut:

Both of these classes are used for internally-animated elements and surfaces on the screen. AnimatedText accepts text and font as uses it to draw text that can scroll down the screen easily for a given lifetime of frames. FadeInOut is similar, but has a dark transparent rectangle which fades in and out over the lifetime.

Class MenuScreen:

Contructor MenuScreen(master, bgImg) where master is the app window and bgImg is the background image Surface to use for this screen.

A MenuScreen object is, essentially, an object which contains a series of buttons on it, is able to parse mouse events to these buttons, is able to use the keyboard to see which button is selected, and can return a surface generated from these buttons and the given background image. What makes this class vital for this program, however, is its ability to be subclassed to add new and powerful mechanics to the screen.

The .update() function plays a major part in this. RiichiMahjongApp will call this function every turn for the current screen being displayed. While this function does nothing in this particular class, subclasses wishing to describe their own sets of onscreen action can use this to update their internal logic before running .returnSurface() and drawing everything to the screen.

.popupDialog(dialog, info) is a function which allows this screen to open up Tkinter windows on command. Essentially, it pauses the current running of the main window in favour of popping up a Tkinter app class, the specific class being given by the string argument passed to it. info is an optional argument that passes the provided info to the window on creation. After the dialog has closed, .destroyDialog() allows for the easy destruction of the Tkinter loop.

.changeScreen(screen) is a meta function as a part of this class that, in turn, asks the RiichiMahjongApp object to change which screen is currently displayed, effectively destroying the current MenuScreen object in the process.

popupDialogs.py:

This file mostly focuses around defining central menu objects which can be subclassed for specific GUI elements. Each of them has a .returnSurface() method, which returns what they look like on the screen so that they can be blitted together for the final game window.

Class Window(object)

Constructor Window(curRoot, curScreen) where curRoot is the root Tk class running as a part of the screen, and curScreen is the screen said class is running on.

This is the base class for all popup windows, which, essentially, are Tkinter windows that can be made to appear on command, containing useful buttons and framed areas for playing the game with. By using the curScreen variable, these buttons can change settings on the main screen from this window. A great variety of these dialogs are used in this program, as shown in the User Interface section of this document.

Class RadioTwoChoices(Frame)

Constructor RadioTwoChoices(curRoot, backcolour, textone, texttwo) where curRoot is the root Tk class running as a part of the screen, backcolour is the colour of the background and textone/two are the text to place in each text label.

This class is a simple two-choice set of radio buttons, which can be used as a frame in other Tkinter windows for any choices between two different True/False options. Similar to the other form items available in Tkinter, the current selected radio button can be obtained by using .get().

Class WindowRoundEnd(Window)

Constructor WindowRoundEnd(curRoot, curScreen, winInfo) where curRoot is the root Tk class running as a part of the screen, curScreen is the screen said class is running on and winInfo is a list of info relating to the most recently finished round.

This is the most complex Tkinter window class, and its design can be seen in figures 7 and 8 in the previous section. Essentially, it automatically accounts for how the previous round ended, and if it was finished with a player winning, it generates a canvas with the player's hand drawn on it, along with applicable dora and other useful data.

To accomplish this, this class makes use of two other custom frame classes. WindowRoundEnd_DrawHandCanvas utilises the external PIL library to load and convert tile image data to a format usable in Tkinter, constructing the requried canvas. WindowRoundEnd_YakuFrame creates a formatted list of yaku with customised buttons which pop up descriptions about how each yaku works.

mainMenu.py:

Class MainMenu(MenuScreen)

This is a subclass of MenuScreen, and is the first screen the user is greeted with when starting up the game process. It does not particularly expand upon the functionality of MenuScreen beyond adding a few simple buttons, as shown in figure 1 in the User Interface section above.

gameScreen.py:

Class GameScreen(MenuScreen)

Constructor GameScreen(master, startingValues) where master is the app window and startingValues is a list of flags, integers and strings that should be passed to the window for the initial setup.

The GameScreen can be considered the major logic class in this application, and is the largest of all these modules. This one class has a great many methods and attributes, which for the most part are not used outside of this module.

The basic structure of GameScreen is no different from a normal MenuScreen, with the slight exception that GameScreen has a fully defined .update() function which constantly updates each turn. Using a variable called _curStage, the gameScreen is able to tell where the currently playing game is up to, and what actions and class instances it should define at any one point. The GameScreen running cycle is described in detail later in this document.

.saveGame() and .loadGame() are two very similar functions used in this screen. In order to save the current status of the game, .saveGame() goes through every single required game attribute and generates a save file that, when executed line by line, should define these variables in the exact same state in which they were saved. For example, it would save the line 'self._playerTurn = 0' as a single line in this file based upon the value of self._playerTurn. After a save file has been constructed, GameScreen is able to obtain its startup attributes by simply executing every line in this file as it finds them.

mahjongGlobals.py:

This module merely contains long lists of global variables and file locations, which is useful for other mahjong modules that require predefined globals. This allows for the easy editing of heavily used numbers on the fly, such as the width and height of each tile.

AI.py:

Class NoneAI:

Constructor(self, gameScreen, playerID) where gameScreen is the screen which this AI player belongs to and playerID is the ID of the player on this screen who this AI is for.

A basic Mahjong AI class. Essentially, whenever the game logic in GameScreen arrives at a decision point for any player that is not the user, it then goes through all of its AI players and asks it questions using functions like .checkDiscard() to find how it would respond in its current situation. In this case, the default response of the AI is just to deny all queries and to always discard the first tile in its hand.

This particular AI can be thought of as a baseline class. All other AI classes build off this.

.checkChi(tile, poss) is a slightly unintuitive function in these AI classes. Essentially, it asks the AI whether they want to chi the given tile. Because there can be multiple possibilities as to how a tile can be called in a straight, the second argument, poss, gives a list of tuples containing hand index ID's for valid pairs of other tiles for the straight. This method must then discard one of these pairs at the end to answer the question on how they should act towards this tile.

Class GeoffAI:

The first actual mahjong AI. This AI simulates a complete rookie player who calls on every tile and doesn't understand the rules of the game. It does this by always randomly choosing its discards and always returning True for any of the checker methods.

Class HighHandAI:

This AI focuses entirely on achieving the high score 'single suit' hands. It does so by constantly calculating which suit it should focus on based upon the number of each individual suit, and then by trying to maximise its chances of obtaining a valid hand.

Class AttackAI:

This AI focuses entirely on trying to score a valid hand as quickly as possible, using cheap yaku like tanyao or pinfu. This is accomplished through the use of a long list of checkers as to which tiles are least likely to help this goal and discarding them first.

Class DefendAI:

This AI does not try to win, and instead attempts to avoid dealing in to another player's hand whenever possible. It first makes lists of safe tiles, then uses hand logic to weigh which tiles are the best choice to discard at any one point.

mahjong_rulebase package:

This package contains mainly data structures and independent conceptual mahjong classes, such as players or tiles, which have methods and properties that are vital for many of the above gameplay-based classes.

Class Tile:

Constructor Tile(tileID) where tileID is a two digit number, where the first digit represents the suit of the tile and the second digit is the numeric value.

A class for the conceptual concept of a tile, used mainly for the handy creation and methods that it provides. For example, isTerminal() can determine whether a tile is a 1 or a 9 instantly, and getNextTile() can quickly find and return the next tile along in a suit. As well as this, the class can automatically search up tile names using a given 'tile.txt'.

Class TileCollection:

Constructor TileCollection(setType, tileMain, side) where setType is a string, tileMain is the main meld Tile and side is an integer representing which player the meld was taken from.

A class representing melds of tiles, such as pons, chis, kans and so forth. Unlike simple lists, this collection just stores the main tile in the meld and the type of the meld, making it more efficient for functions which require knowing what the type of meld it is. As well as this, it provides a clean way to keep these sets of tiles together without needing some messy hacked-together solution that would be provided with just basic lists.

Class Yaku:

Constructor Yaku(yakuID) where yakuID is a string identifier used in yaku.txt

A class representing all the different yaku that can provide scores. When a yaku is named, all of its information is loaded from a file 'yaku.txt'. These include info like the full name of the yaku, how much it is worth and a description of it.

Class Player:

Constructor Player(handsize, suitnum, name) where handsize is the maximum length of the hand, suitnum is the number of numbered suits in the game and name is the player's name as a string.

Is effectively all the information and methods behind a player's hand. Tiles are divided into _mutable and _immutable lists, differentiating tiles that are already called and those which can be legally discarded. There are many, many useful functions in each players hand, for example:

.isValid() takes apart the tiles in the hand and sees whether they can be arranged into four valid melds plus a pair, returning a list of these possibilities if they can.

.isTenpai(listOfTiles) determines whether the hand is one tile away from being a valid hand by trying all of the tiles given in listOfTiles.

.canChi(tile) determines whether is possible to get a three straight using the given tile, returning a list of tuples, which each contain the hand indexes for the other two tiles required.

Class PlayerScore:

Extends the above, to add even more features. This class contains methods related to calculating score and determining what yaku are in the hand. Because these methods are a lot more arbitrary than those above (for example, adding nondescript amounts of points for various things), they have been separated out so that those wishing to write their own scoring rules merely have to overwrite this class, not the entire Player class.

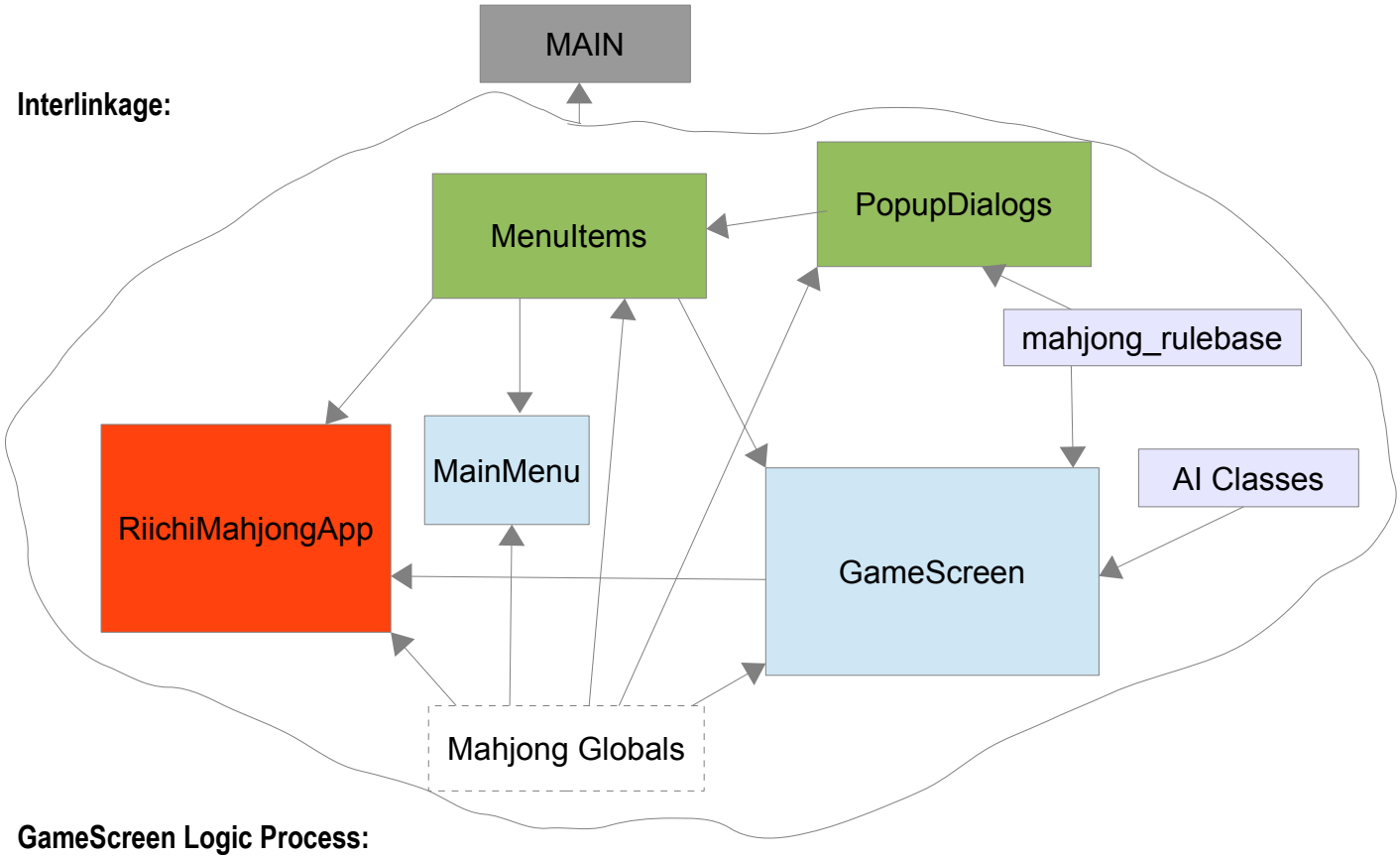
Class IOHelper:

A basic helper class that allows for the easy and swift retrieval of information from specified text files. These files can have commented lines, and are delimited by a specified character.

Class GameRunningException:

Simple exception class for logic errors when handling tiles.

Interlinkage:



GameScreen Logic Process:

Note that this process is mostly streamlined, and does not take into account waiting for animations or for the player's responses and menu choices.

1. Firstly, load and define all standard sprite locations and variables that will be reused over and over.
2. If the game is being loaded from a file, go to 3; otherwise, go to 4.
3. Use the saved game file to define and set all game variables to their stored state. Go to 11.
4. Set all game variables, such as the player's hands or the wall, to their round starting state.
5. Start the round; display the round name, roll the dice twice to get the dealer and the location of the dead wall, and then allocate seating to each player. Draw tiles from the wall until every player had thirteen in their hand. Finally, determine who goes first, then go to 8.
6. Check to see if any of the players can ron/kan/pon/chi on the current tile, and then see if they wish to do so. If they ron, they win: go to 13. If they kan/pon/chi, then the current player switches to them, go to 11.
7. First check to see if the game should end in a draw. If so, go to 13. Otherwise, the current player should draw a tile from the wall.
8. Check to see if the player can call tsumo/kan/riichi on that tile. If so, then do so, heading to 13 if declaring tsumo, to 9 if declaring riichi or back to here if declaring kan. If not, or the user/AI declines, then go to 10.
9. If the user declared riichi, then highlight the possible tiles to discard and get their response. If it was the AI, then query their choice of discard. After discarding a tile, go to 12.
10. Check to see if the player is currently in riichi. If not, continue. If so, and the tile completes their hand, then they win: go to []. If it does not, then discard the tile and go to 12.
11. Ask the player or AI which tile in their hand they want to discard, and then discard it.
12. Add one to the turn counter and rotate the players around to the next player, getting ready for the next turn. Go to 6.
13. Get the current status of the game. If it was a draw, display what kind of draw; if it was a ron or tsumo, show the winning hand and its worth. Tally up the scores and get the game ready to move on. If the dealer won, then go to the next bonus round and go back to 5. If not, and it is not the last round, go to 5. Otherwise, go to 14.
14. The game is now over. Display a list of the scores over each round, and then let the player quit back to the main menu.

Support Modules:

The specific requirements for this program have been supplied in the readme file in the root directory of the application. Briefly, the external libraries used by classes in this program are:

- pygame, especially the Surface and mixer libraries, as well as display, transform and locals. These libraries form the essential basis of this program.
- PIL, which is only needed to convert some image formats for use in Tkinter canvas drawing.

As well as this, the official python libraries used were sys, random, os, webbrowser, Tkinter, tkMessageBox and time.