LP24 Report

Subject : Rummikub

I) Presentation of the subject

The project consists in the development of a Rummikub game. In order to develop it, I chose to use the Swing library for graphical interface. The game works that way :

Once the game starts, a menu is displayed (called Menu Scene), enabling the user to make a few things such as consulting the game's rules for instance. This is also where the player will decide to start the game, the application will then ask for the number of players to play, and that's where the game start.

This particular phase of the program will be seen only once in the game, because the user chooses the number of players only at the beginning. The players must be real persons, there isn't any AI implemented in this project.

The players will then play against each other turn by turn, until one of them win. After that, a new menu is launched to display the players score and to allow the players to play again or to quit the application. (called End Game Scene)

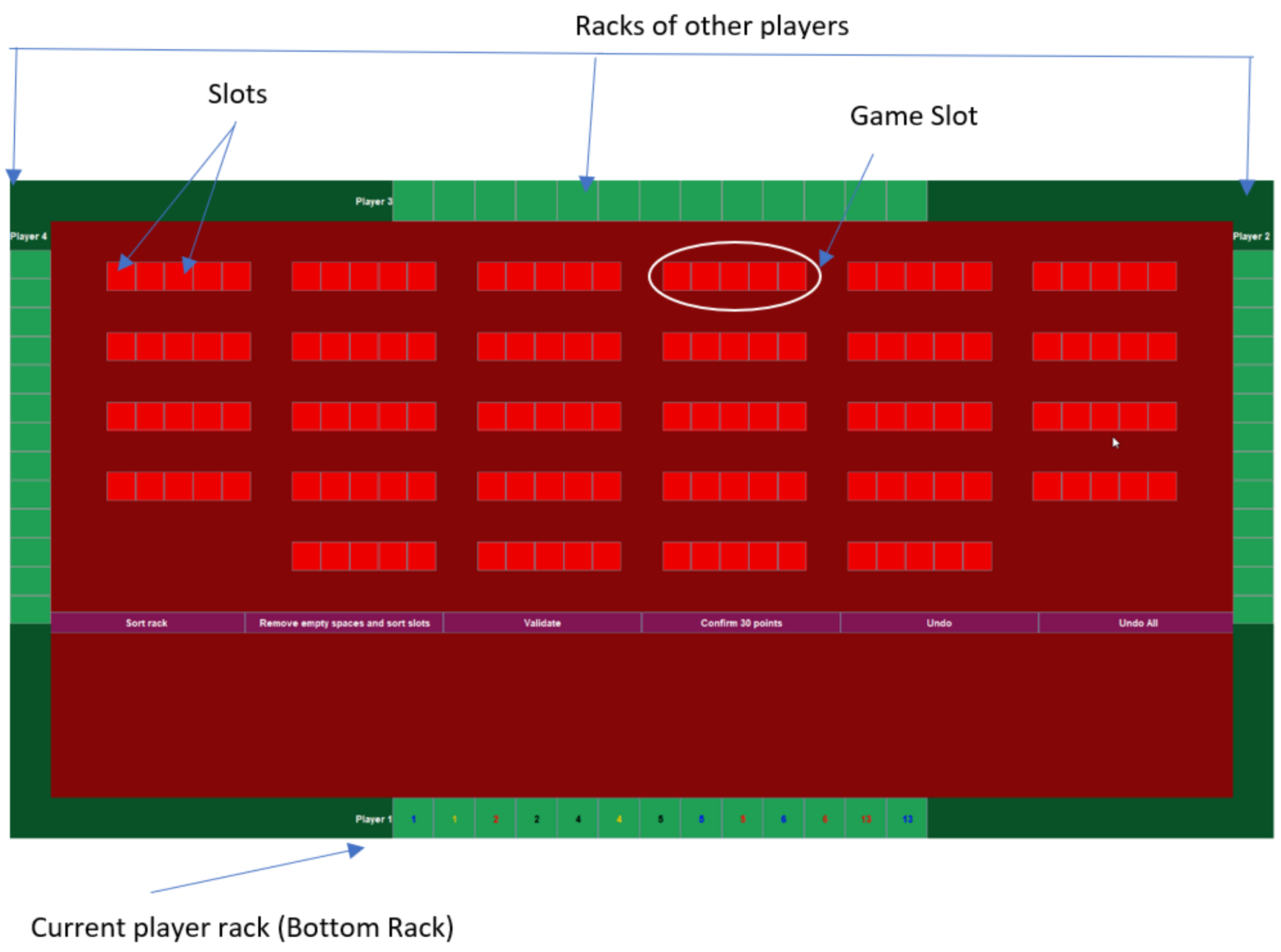
Therefore, the application will always follow this kind of way :

Start menu -> Round -> Restart Menu -> Round -> Restart Menu -> Quit

(MenuScene -> GameScene -> EndGameScene -> GameScene -> EndGameScene -> Quit)

The usage of the start menu and restart menu being easy to understand and to explain, I will now talk about the round part.

Here is a picture of the game when players play against each others :



This Panel actually is split into three distinct parts :

- The center (called Board) : it corresponds to the area where the player places their tiles (in red)
- The bottom (called Bottom Rack) : it corresponds to the tiles of the player whose the turn is (called current player).
- The sides (called Rack (excluding the bottom)) : it corresponds to the other players tiles (which are deactivated). It also shows the order of the player's turn. Here, Player 1 goes first, then Player 2, then Player 3... At each end of turn, the players are rotating clockwise.

Therefore, all the actions occur between the bottom side of the panel, and the center of the panel.

The center is composed of 2 distinct things : the features part, and the slots part.

The slot part : it contains an amount of slots where the current player is able to place his tiles in order to form some sets. These slots are put in group of 5, forming GameSlots. Each GameSlot will then have the capacity to contain a set.

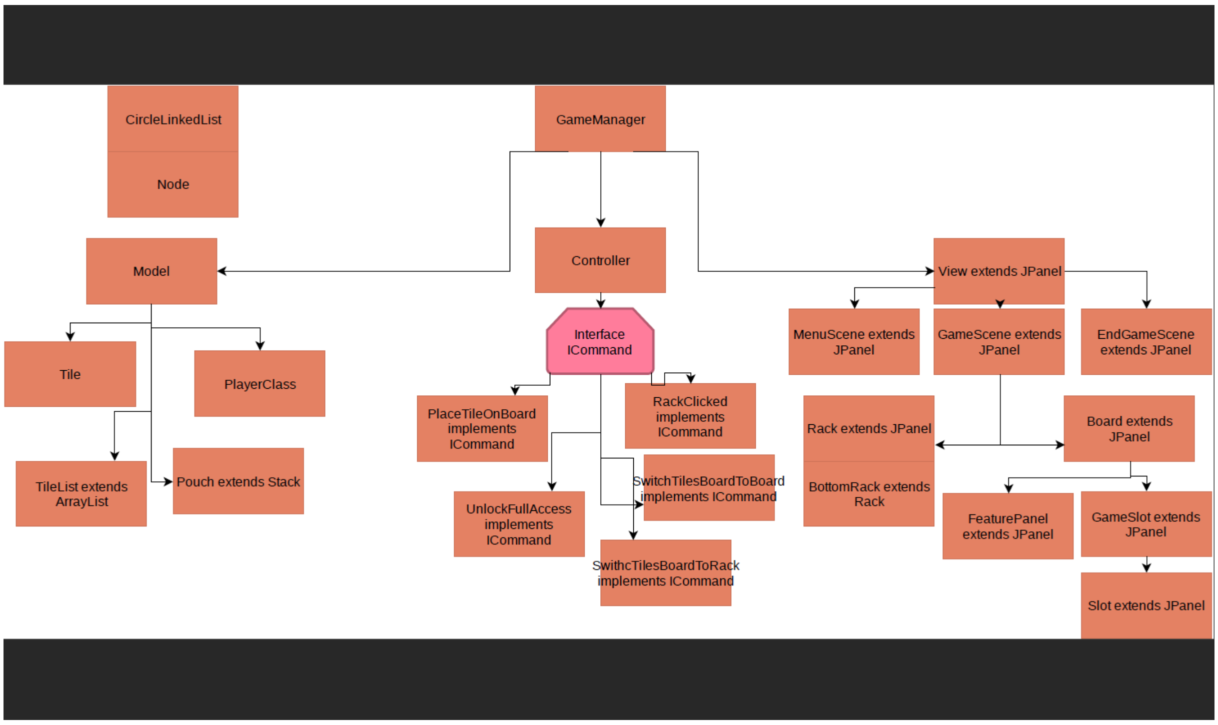
The number 5 was chosen because any set bigger than 5 tiles can be split into 2 or more sets of tiles.

The feature part is simply a set of buttons which allows the player to make multiple things which can make his life easier. He can :

- Sort his tiles
- Remove empty spaces between the tiles placed on every game slots of the board (while, in the main time, sort the tiles of the game slots).
- Confirm that he has enough points on the board in order to play with the sets from other players. In that case, this button will be unavailable for that player, and he will always have access to the other player’s sets until the end of the round. (We will say that the player is “out”. When a player is not “out”, the sets from others players are locked)
- Cancel his last action/cancel all his actions of the turn
- Get to the next turn

To sum up, the current player places his tiles as he wants on the different gameSlots, forming his sets himself. When he is done, he can whether tell the game that he has enough points to be “out” in order to continue playing with the other players sets or end his turn directly. (the management of the user’s error (when placing his tiles) will be explained in detail later (when he hits the confirm button although he doesn’t have enough points to be out for instance).

II) Architecture of the project :



The project is organized that way (every rectangle represents a Class). The GameManager is responsible for the instantiation of the model, the controller, and the view. These 3 classes are the implementation of the MVC pattern. The model handles all the data (which means the players, the pouch and the tiles for instance), the view handles the UI elements. It corresponds to everything the player will see and will interact with. Finally, the controller processes the user's actions on the view. In other words, it handles the events of the project. An event occurs on an element from the view, which will be catch by the Controller, the controller will then do a set of actions to update the model relatively to the event that occurred, while also updating the view.

The model, view and controller are implementing the singleton pattern in order for them to only have one instance created only, and to have an access to them everywhere in our program without passing them as parameter. Therefore they are first instantiated at the beginning of the program, and re-initialized at every beginning of round (but not re instantiated).

This implementation helps keeping the project clean and modular.

Let's have a more detailed explanations of how the model, controller and view work.

The Model

The model has several purposes. It instantiates the persistent data of a round (which are the players, tiles and pouch). The players are stored in a Circle linked list. This is a type created in order for the turn management to be easier : at the end of a turn, we simply set the head of the list to be the next node and state that the head of the list is the new current player. The model also updates the data when necessary (at the end of a player’s turn for instance, or when a player wins).

The last thing the model is used for is to track the tiles played by the current player during his turn, and to track whether a tile is focused by a player or not. (A tile becomes focused by the player whenever there was no focused tile when the player clicks on it). We will see why later.

The View

The view’s purpose is to display the UI element on the screen. Thus its only usage is to build the game window, and to switch between the different scenes of the game : the MenuScene (the first to be displayed), the GameScene and the EndGameScene(the one displayed at the end of a round).

To do that, a card layout is used to choose which panel we want to display.

The Controller

The controller’s purpose will be to update the view and the model based on the events he receives.

III) Detailed operation of a round

1) Placement of the tiles

The player places the tiles thanks to mouse clicks. When he clicks on a new tile, this tile becomes focused (if no other tiles were previously focused) and is highlighted. From then, the player is able to click whenever he wants to perform an action. He can place the tile he focuses on the board, put it back in his rack if he played it during his turn, switch the position of tiles… Whenever the player clicks on a button, an event is fired, this event will trigger a method with an if else chain to determine which action the user wants to perform, thanks to the information of the button the player clicked on, and the information of the focused tile. (for instance if the focused tile variable is null, and the player clicks on a tile, then this tile becomes focused, if he then clicks on an empty slot, then the program will know the user wants to place a tile on a board because the focused tile variable is not null and the event was fired by in a slot button. The program will then place the focused tile on the board and set the focused tile variable to null).

2) Management of the players' tiles

To each player corresponds a list of tiles which comes from the original pool of tiles, where all tiles are. This pool of tile first is shuffled, before being distributed among players at every beginning or rounds.

At the beginning of a player's turn, to each rack will be attributed the list of tiles of its player (it will be stored in a variable linked to the rack, called "buttonsToAttach"). During the turn, we will manipulate this variable and update it thanks to the UI. Every time a player will take back a tile from a board to his rack, or place one of his tiles on the board, the buttonsToAttach list will be refreshed to match with the UI of the Rack. To be more specific, when we are, for instance, placing a tile on the board, the buttonsToAttach variable stays the same, however there is one less component attached to the Rack(The UI is updated, but our variable is not). Therefore there's a function which will refill the buttonsToAttach variable (by "reading" the components attached to the Rack) to match tiles attached to the Rack.

We're using that function to keep our variable updated every time there is a change in the Rack

That way :

- the tiles rendered in the rack will always match the tiles the players did not place on the board (or, equivalently, the buttonsToAttach variable)
- the placement of the tiles within the rack will remain the same over the turns

There will be two variables which will contains all the player's tiles during the round. These variables are the "buttonsToAttach" which corresponds to the tiles in the rack, and the "tilesPlayedDuringTurn", which corresponds to the tiles that the player placed on the board (and only the ones on the board, if a player take a tile back in his rack, it will no longer belongs to the tilesPlayedDuringTurn variables). The tilesPlayedDuringTurn variable is used to keep track of the tiles which are on board, but still belongs to the current player. That way, the player will be able to take them back into his rack. When the player validates his turn or when he confirms that he has 30 points to be "out", then these tiles will no longer be considered as his tiles, therefore, he won't be able to place them back in his rack (in other words, the tilesPlayedDuringTurn variable is set to null) unless he reverses the action with the undo button (however he wouldn't be "out" anymore).

3) Undo button and command pattern

The player is able to rewind his actions as long as he doesn't end his turn. The actions he can rewind corresponds to the classes which implements the ICommand interface (see the project's architecture picture). In the Controller class, there's a list of ICommand to deal with this rewind mechanic. Each time the player does an action which is rewindable, an object of the class corresponding to this action will be instantiated and added to the list of ICommand. The methods the interface implements are execute() and undo(). The code in execute will contain the code needed for an action to be performed, while the undo code will contain the code which will do the inverse of the execute's code. When the player hits the Undo button, the head of the list is removed, and his undo method is called.

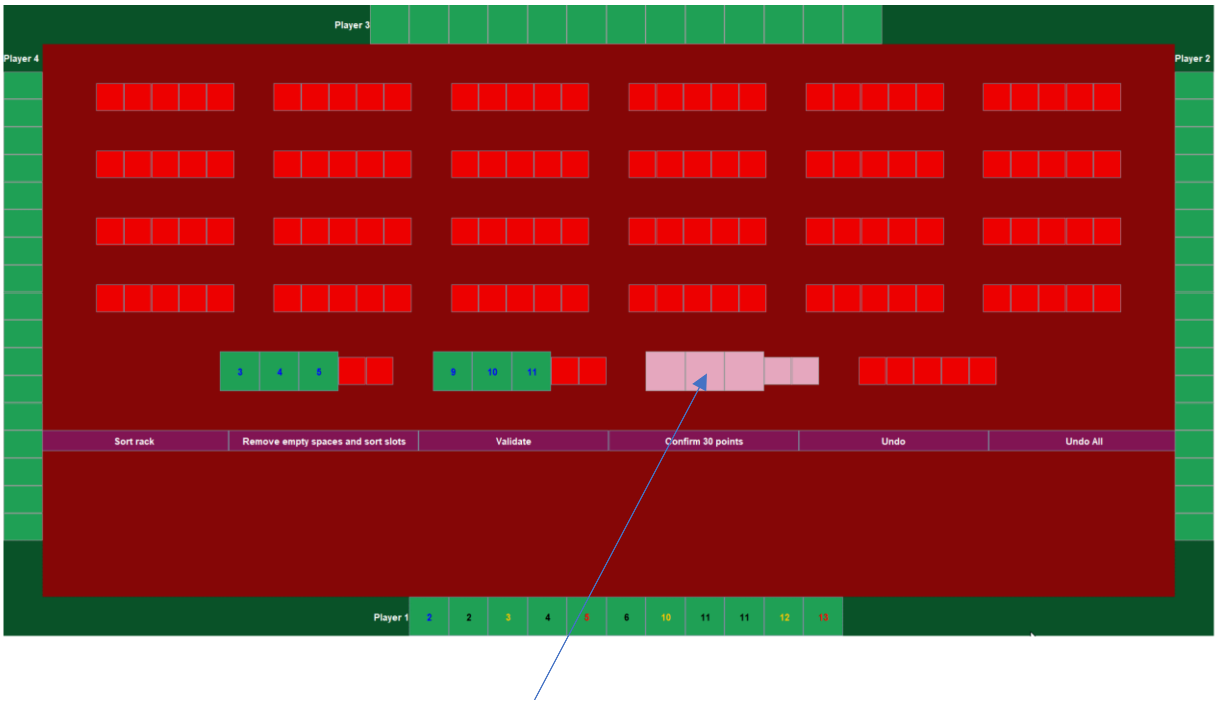
4) Management of the validate and confirm 30 points buttons

Once the player is done placing his tiles, he can click on validate to get to the next turn or tell the game that he has enough points in order to unlock the sets from the other players (with the button "confirm 30 points". In the case of the confirmation of the 30 points, the game will first check that every sets that the player formed is valid, if so, it will count the points formed by these sets. If these points exceed 30, then the player will be considered "out", the button will be disabled, and the sets from other players will be available. If these points don't exceed 30, a message will be displayed telling the player that he hasn't enough points to be out. The player has the possibility to undo this action with the undo button, if he does so, he won't be out anymore, he will be able to put back the tiles he put on the board into his rack and so on.

When he clicks on validate, the program will check for two things before getting to the next turn :

- It will scan every game slot to see if there is an error in the combination of sets. To do that, every game slot containing tiles will be associated with a TileList variable. The TileList class is a class extending ArrayList<Tile>, which is used to determine whether the list of tiles is a group, a run, or not a set. If one of the TileList from the different gameSlot is not a set, then a warning will occur, and the player will have to change the placement of his tiles.

- If there is no problem in the gameSlots (in other words, if the first check is valid), but there still are some sets the player formed on the board even though the player is not "out", there will be 3 possibilities to happen depending on the case :
    - If there is a big enough amount of points in the player's sets, and there are sets from other players on the board, we consider that the player forgot to hit the "confirm 30 points" button, and we simulate the action of this button in order for the player to be able to play with the sets of the other players (the player will then be considered as out). If it was intended from the player not to hit the confirm 30 points button because he knew he had no interest in the sets of other players, then he will have to click on validate two times in a raw to get to the next turn.

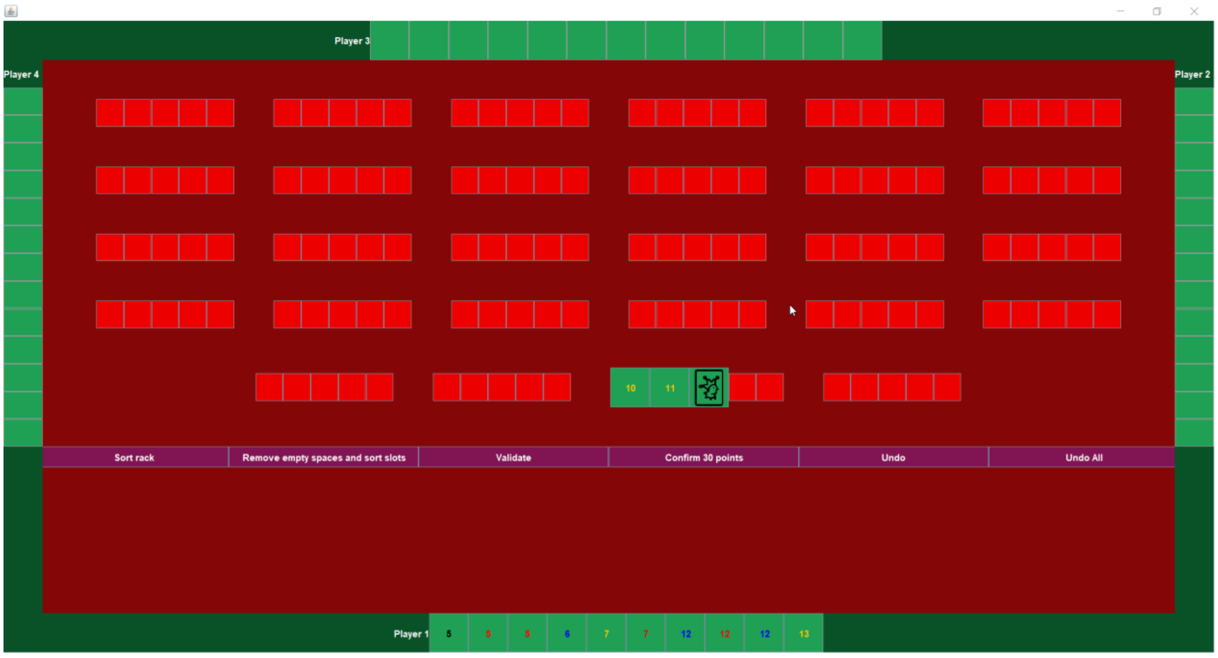


Set from an other player, not available (locked) because the current player is not out

The situation I talk about is represented in this picture. Here the player is not out yet, the set formed on the board are valid, and they are exceeding a total amount of points of 30. In that case, when the player hits validate, we set him as out in order for him to play with the other set on the board which was not available for him.

- If there is a big enough amounts of points in the player's sets, but not any set from other players, we directly set the player "out" and get to the next turn. We consider that it was in the intention of the player to directly get to the next turn without hitting the confirm 30 points button, since there were no sets from other players on the board anyway.
- If the total points formed by the sets is below 30, it will display an error telling the player that he has not enough points to be out.

I decided to deal with these two particular cases I've just talked about that way in order not to force the player to click on the confirm 30 points button every time before clicking on the validate button even though when he doesn't place any tile on the board. And I think it is the most comfortable possibility for the player.



In this picture, if the player hits validate, he will be set to out automatically while getting to the next turn. (visual representation of the explanation above)

If there are no problem on the board when validating, the game can get to the next player's turn. The tail of the circle linked list hence becomes the head, and the associated player becomes the current player.

This stops when the pouch is empty, or when a player has no more tiles. If the pouch is empty, the player with the less tiles wins.

(when a player doesn't place any tile on the board during his turn, he automatically draws a card from the pouch).

5) Possible actions the player can do

During his turn, the current player can do the following actions :

- Place a tile on a board (reversible with the undo button)
- Switch tiles within the board (reversible with the undo button)
- Switch tiles between bottom rack and board ((reversible with the undo button)
- Switch tiles within bottom rack
- Undo the last action (using Command pattern)
- Undo all the actions (using Command pattern)
- Sort his tiles

- Sort the tiles placed on the game slot (when a list of tiles is sorted and it contains a joker, the joker’s place will remain the same) and remove the empty spaces between tiles within a same game slot
- Confirm that he has enough points to be out (reversible with the undo button)
- Validate to change player’s turn

IV) Management of the variables’ instantiation for memory optimization

During the whole working of the program, there are variables which will be persistent over the rounds. In other words, there are variables which we will only need to be instantiated once in order for the program to work. However we have, in our case, 3 problems to handle :

- Determine which variables are persistent over the rounds
- Determine when to instantiate them
- Determine when and how to reinitialize them without re instantiating them

For the first point, we can easily see that most of our attributes will be persistent over the rounds. The pool of tiles, the players, the board, the window.. basically every attributes of the model and the view (considering the attributes of the controller only exists to use the caching technique on model and view’s attributes for faster accessibility to them throughout the controller’s methods).

However, some of these attributes can’t be instantiated at the very beginning on the program. Indeed, we need the number of players to determine how much players we instantiate as well as how much racks we instantiate for instance.

For the last point, it is needed to reinitialize the players with new tiles, to clear the racks from their tiles, to clear the slots from their tiles, and to start the process of shuffling tiles and distributing them to players without restarting the instantiation process.

Let’s start from the GameManager class, which purpose is to instantiate our Controller. I implemented the Singleton pattern on the Model and the View without lazy instantiation because their constructor do not rely on other classes. Therefore, their instances are created when the program compiles, in early binding.

In the view’s constructor, the persistent data which doesn’t require the players is instantiated (therefore, everything but the Racks of the players), then it displays the MenuScene so that the user can choose how many players will play the game.

The model’s constructor is used to instantiate the 106 tiles in the pool of tiles array. The players will be instantiated once the number of players will be known by the program.

The Controller’s instance is however created using lazy instantiation, because its constructor relies on the Model and View’s class. Indeed, the Controller’s constructor purpose is to cache the attributes of the model and of the view (the one which are already instantiated) into its own attributes for later usage. It also adds the listeners to all the elements which has already been created (the tiles, the slots, the menu buttons).

We now have all variables which do not need to be re instantiated at every round instantiated and ready to use. When the player will choose the number of players, the Controller will catch the number of players selected by the player and trigger the rest of the object instantiation. Therefore, it will start, on the one hand, the instantiation of the players, and on the other hand, the instantiation of the Racks. It will also cache and add the listeners to the rest of the variables which couldn’t be instantiated at the very beginning of the program.

We are now good to go, the function of the controller which start the instantiation of the Racks and the Player also tells the View to load the game Scene, and that’s where the round start.

When the round end, the End GameScene appear, and when the player clicks restart, it will launch all the reinitialization functions, shuffle the tiles and distribute them back to players before starting a new round. No new objects will be instantiated during this process.

I would also like to add something which is a bit off topic from this project but that I think might be quite interesting :

If one could find a feature which would make the amount of one of the variables we have modifiable over the rounds (let’s say the tiles), then implementing the object pooling pattern would be quite interesting in order instantiate the right amount of tiles we would need at a particular time, and setting them deactivated or activated depending of the amount of tiles we need, and eventually re instantiate some tiles in case we lack of tiles in the pool.

V) The 4 pillars of OOP

Polymorphism : I mainly used it for the implementation of the Command Pattern. Since the command pattern relies on the polymorphism (because we are storing in a list different objects which implements the ICommand interface in order to call their methods). I also used it for the setEnabled() method of my GameSlot and Slot class, and for some little methods of my program.

Inheritance : I actually used a lot of composition when needing a specific object inside a class I was implementing. For instance my Model class contains an array of Tile, a CircleLinkedList, a Pouch… All these types are actual classes I defined myself. However I didn't use a lot of Inheritance. My pouch class derives from the Stack class in order to extends the Stack behaviors, such as the TileList class which extends the ArrayList<Tile> class. I also made my BottomRack class derives from the Rack class, it has no real usage in the program but it makes more sense to see the BottomRack as a special type of Rack instead of making no distinctions between the BottomRack and the other Racks.

Encapsulation : I made every attribute of my classes as well as all my methods which are used within the class only private. If an object of a class wants to access/modify an other object's attributes, then it will have to use the getters/setters methods of the object to modify. The only public attributes are constants variables.

Abstraction : I respected the principle when thinking of my object and the data it has to contain. Such as the Tile which can be reduced to a color and a number, or a Player which can be reduced to a name and a list of tiles…

VI) Bugs and improvements to be made

There shouldn't be any major bugs in the program. There are two I can think of :

- Sometimes, I have a little visual problem in my UI which is that a Tile can sometimes become invisible until I drag my mouse over it. It honestly isn't perturbing the game's experience in any way though; it is scarce and doesn't prevent the player from playing at all.

- If there are too much Tiles on the board, the place taken by the tiles becomes too big and the messages displayed by the game becomes hard to see/invisible because they eventually are out of screen.

About the improvements, I think there are some I could have made more or less easily.

- The actions a player can make on a tile depends a lot on the state of the tile (whether the tile is focused or not, whether the tile has been played during the turn or not…).. Therefore I think that implementing a finite state machine pattern would have made my code more modular and better when it'd have come to the management of the tiles/user's actions.
- I think that my command Pattern implementations isn't perfect since the actions I depicted in my classes might not need to be that "big". In other words, I think that my actions are too complex and that it might have been possible to find better way to implement this pattern, using simpler classes.

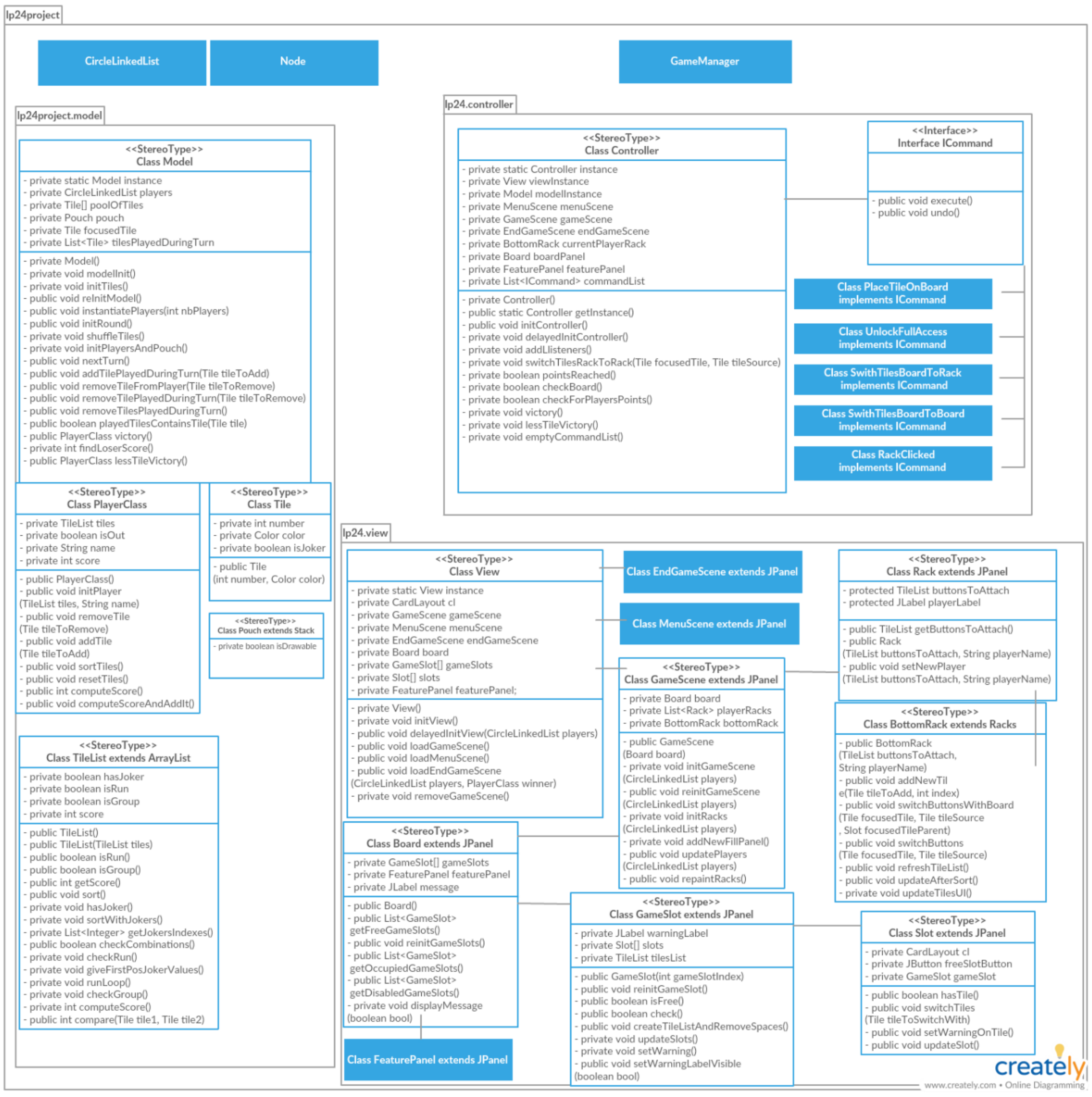
VII) Conclusion

This project is made to familiarize ourselves with OOP and to improve our skills in programming in Java at the same time. While developing that project, I noticed that a long work was necessary in order to start with a good architecture which will allow our project to be modular and optimized. However, once the architecture is set and that we know where we are going, it becomes very simple to code a lot without having the feeling of being lost or overwhelmed by the amount of lines of code when debugging for instance.

The hardest part of that project for me was to learn how to use the swing library. I struggled to determine how my View had to be set in order for it to be well organized.

Another hard part was, well, to code everything. I don't know what was really expected as output for this project however it really took me a long, long, long time to get this all done.

Annex


lp24project
CircleLinkedList
Node
GameManager
lp24project.model
<<StereoType>>
Class Model
- private static Model instance
- private CircleLinkedList players
- private Tile[] poolOfTiles
- private Pouch pouch
- private Tile focusedTile
- private List<Tile> tilesPlayedDuringTurn
- private Model()
- private void modelInit()
- private void initTiles()
- public void reInitModel()
- public void instantiatePlayers(int nbPlayers)
- public void initRound()
- private void shuffleTiles()
- private void initPlayersAndPouch()
- public void nextTurn()
- public void addTilePlayedDuringTurn(Tile tileToAdd)
- public void removeTileFromPlayer(Tile tileToRemove)
- public void removeTilePlayedDuringTurn(Tile tileToRemove)
- public void removeTilesPlayedDuringTurn()
- public boolean playedTilesContainsTile(Tile tile)
- public PlayerClass victory()
- private int findLoserScore()
- public PlayerClass lessTileVictory()
<<StereoType>>
Class PlayerClass
- private TileList tiles
- private boolean isOut
- private String name
- private int score
- public PlayerClass()
- public void initPlayer
(TileList tiles, String name)
- public void removeTile
(Tile tileToRemove)
- public void addTile
(Tile tileToAdd)
- public void sortTiles()
- public void resetTiles()
- public int computeScore()
- public void computeScoreAndAddIt()
<<StereoType>>
Class Tile
- private int number
- private Color color
- private boolean isJoker
- public Tile
(int number, Color color)
<<StereoType>>
Class Pouch extends Stack
- private boolean isDrawable
<<StereoType>>
Class TileList extends ArrayList
- private boolean hasJoker
- private boolean isRun
- private boolean isGroup
- private int score
- public TileList()
- public TileList(TileList tiles)
- public boolean isRun()
- public boolean isGroup()
- public int getScore()
- public void sort()
- private void hasJoker()
- private void sortWithJokers()
- private List<Integer> getJokersIndexes()
- public boolean checkCombinations()
- private void checkRun()
- private void giveFirstPosJokerValues()
- private void runLoop()
- private void checkGroup()
- private int computeScore()
- public int compare(Tile tile1, Tile tile2)
lp24.controller
<<StereoType>>
Class Controller
- private static Controller instance
- private View viewInstance
- private Model modelInstance
- private MenuScene menuScene
- private GameScene gameScene
- private EndGameScene endGameScene
- private BottomRack currentPlayerRack
- private Board boardPanel
- private FeaturePanel featurePanel
- private List<ICommand> commandList
- private Controller()
- public static Controller getInstance()
- public void initController()
- private void delayedInitController()
- private void addLlisteners()
- private void switchTilesRackToRack(Tile focusedTile, Tile tileSource)
- private boolean pointsReached()
- private boolean checkBoard()
- private boolean checkForPlayersPoints()
- private void victory()
- private void lessTileVictory()
- private void emptyCommandList()
<<Interface>>
Interface ICommand
- public void execute()
- public void undo()
Class PlaceTileOnBoard
implements ICommand
Class UnlockFullAccess
implements ICommand
Class SwithTilesBoardToRack
implements ICommand
Class SwithTilesBoardToBoard
implements ICommand
Class RackClicked
implements ICommand
lp24.view
<<StereoType>>
Class View
- private static View instance
- private CardLayout cl
- private GameScene gameScene
- private MenuScene menuScene
- private EndGameScene endGameScene
- private Board board
- private GameSlot[] gameSlots
- private Slot[] slots
- private FeaturePanel featurePanel;
- private View()
- private void initView()
- public void delayedInitView(CircleLinkedList players)
- public void loadGameScene()
- public void loadMenuScene()
- public void loadEndGameScene
(CircleLinkedList players, PlayerClass winner)
- private void removeGameScene()
Class EndGameScene extends JPanel
Class MenuScene extends JPanel
<<StereoType>>
Class GameScene extends JPanel
- private Board board
- private List<Rack> playerRacks
- private BottomRack bottomRack
- public GameScene
(Board board)
- private void initGameScene
(CircleLinkedList players)
- public void reinitGameScene
(CircleLinkedList players)
- private void initRacks
(CircleLinkedList players)
- private void addNewFillPanel()
- public void updatePlayers
(CircleLinkedList players)
- public void repaintRacks()
<<StereoType>>
Class Board extends JPanel
- private GameSlot[] gameSlots
- private FeaturePanel featurePanel
- private JLabel message
- public Board()
- public List<GameSlot>
getFreeGameSlots()
- public void reinitGameSlots()
- public List<GameSlot>
getOccupiedGameSlots()
- public List<GameSlot>
getDisabledGameSlots()
- private void displayMessage
(boolean bool)
Class FeaturePanel extends JPanel
<<StereoType>>
Class GameSlot extends JPanel
- private JLabel warningLabel
- private Slot[] slots
- private TileList tilesList
- public GameSlot(int gameSlotIndex)
- public void reinitGameSlot()
- public boolean isFree()
- public boolean check()
- public void createTileListAndRemoveSpaces()
- private void updateSlots()
- private void setWarning()
- public void setWarningLabelVisible
(boolean bool)
<<StereoType>>
Class Rack extends JPanel
- protected TileList buttonsToAttach
- protected JLabel playerLabel
- public TileList getButtonsToAttach()
- public Rack
(TileList buttonsToAttach, String playerName)
- public void setNewPlayer
(TileList buttonsToAttach, String playerName)
<<StereoType>>
Class BottomRack extends Racks
- public BottomRack
(TileList buttonsToAttach,
String playerName)
- public void addNewTil
e(Tile tileToAdd, int index)
- public void switchButtonsWithBoard
(Tile focusedTile, Tile tileSource
, Slot focusedTileParent)
- public void switchButtons
(Tile focusedTile, Tile tileSource)
- public void refreshTileList()
- public void updateAfterSort()
- private void updateTilesUI()
<<StereoType>>
Class Slot extends JPanel
- private CardLayout cl
- private JButton freeSlotButton
- private GameSlot gameSlot
- public boolean hasTile()
- public void switchTiles
(Tile tileToSwitchWith)
- public void setWarningOnTile()
- public void updateSlot()
creately
www.creately.com • Online Diagramming