COMP3211

Software Engineering

Group 1 – Design Document

Monopoly

Hong Kong Special Edition

Group 1	
Student ID	Student Name
	Kent Max CHANDRA
	Cheuk Tung George CHAR
	Tommaso Fabrizio COSTANTINI
	Xin DAI

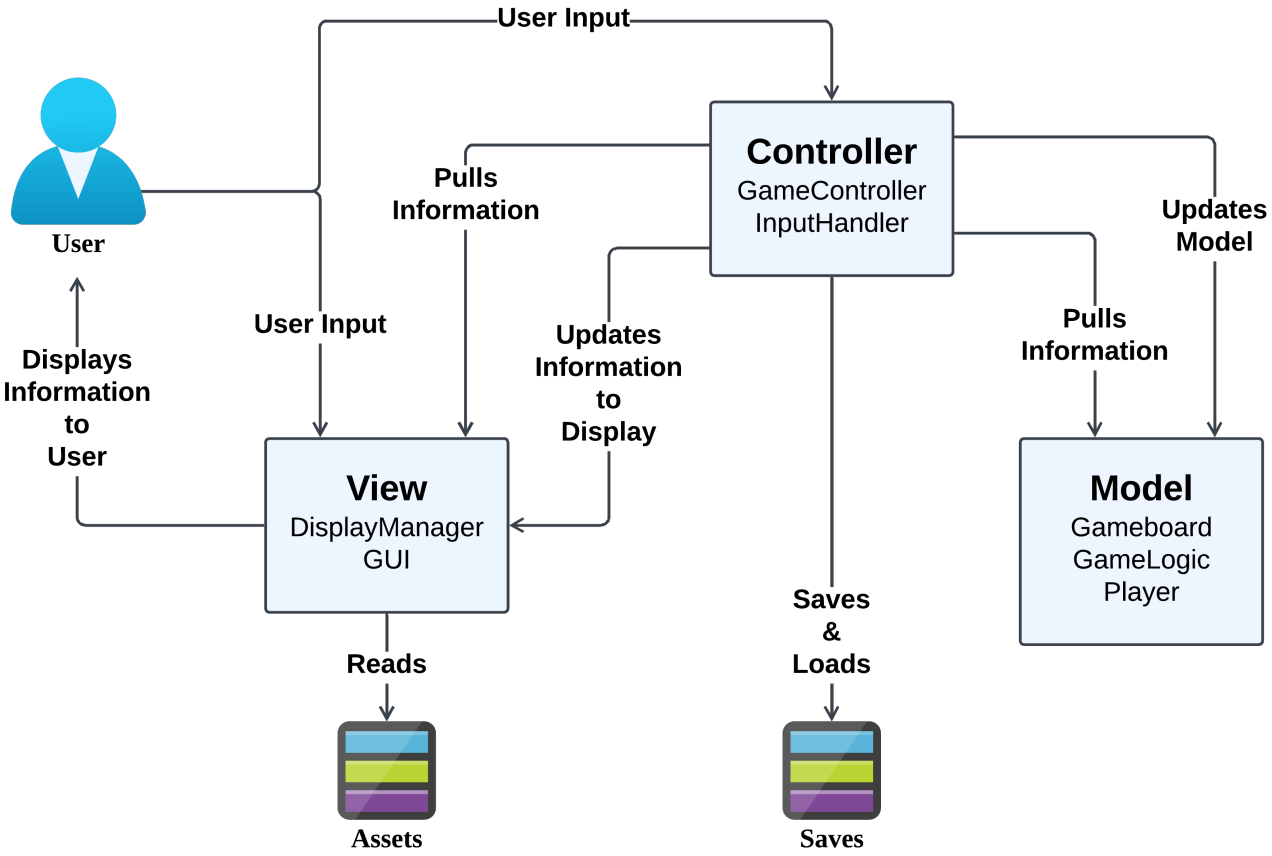
CONTENTS

I. Architecture

Software Architecture

The project follows the Model-View-Controller (MVC) architectural pattern to ensure a clean separation of concerns, improve code maintainability, and allow future extensions. The Model manages the core game logic, the View is responsible for rendering the user interface, and the Controller manages interactions between the Model and View.

Architecture Design



I. Architecture

Default game information is hardcoded into the Model. On startup, the View references all assets in the Assets folder but starts without any information on what to display. It gets populated by information that the Controller pulls from the Model and loads into the View itself.

Except for two instances where the Controller must halt execution until it detects a user click, in all other instances, input is received by the View that updates its internal fields or calls the appropriate methods.

Subsequently, Controller pulls information form the View. If the user is editing information, then controller updates the Model accordingly. Otherwise, the Controller will call methods from the Model and determine the next action.

Upon pulling new or updated information from the Model, the Controller updates the View fields, and the UI is updated consequently.

In this Architectural design the link between all components is the Controller. The View and the Model do not have access to the Controller, nor they *know* it exists. Only the Controller *knows* the existence and has access to all components of the design.

The Controller also has access to the Saves folder where it can save and from where it can load game instances and board designs.

II. Code Structure and Relationships

Code Structure

As shown in the Architectural Design diagram printed above, each component has different modules that carry out different functionalities.

The following is the comprehensive table of all modules, and the classes contained within. All classes on a higher indentation level are children, on a lower indentation level they are its parent classes.

Component	**Module**	**Classes (*Abstract)***
Controller	GameController	• GameController • SavedGameboard • SavedGame
	InputHandler	• InputHandler

Model	Gameboard	• *(Tile)*
		  ○ Property
		  ○ Jail
		  ○ Go
		  ○ GoToJail
		  ○ Chance
		  ○ IncomeTax
		  ○ FreeParking
		• Gameboard

	GameLogic	• GameLogic
	Player	• Player

II. Code Structure and Relationships

View	DisplayManager	• *(DisplayManager)*
		  ○ GameplayFrame
		  ▪ EditBoardFrame
		  ○ NewGameFrame
		  ○ MainMenuFrame
		  ○ *(LoadFrame)*
		  ▪ LoadGameFrame
		  ▪ LoadBoardFrame
		  ○ *(SaveFrame)*
		  ▪ SaveGameFrame
		  ▪ EnterNameFrame
		  ▪ SaveBoardFrame
		  ○ InfoPageFrame

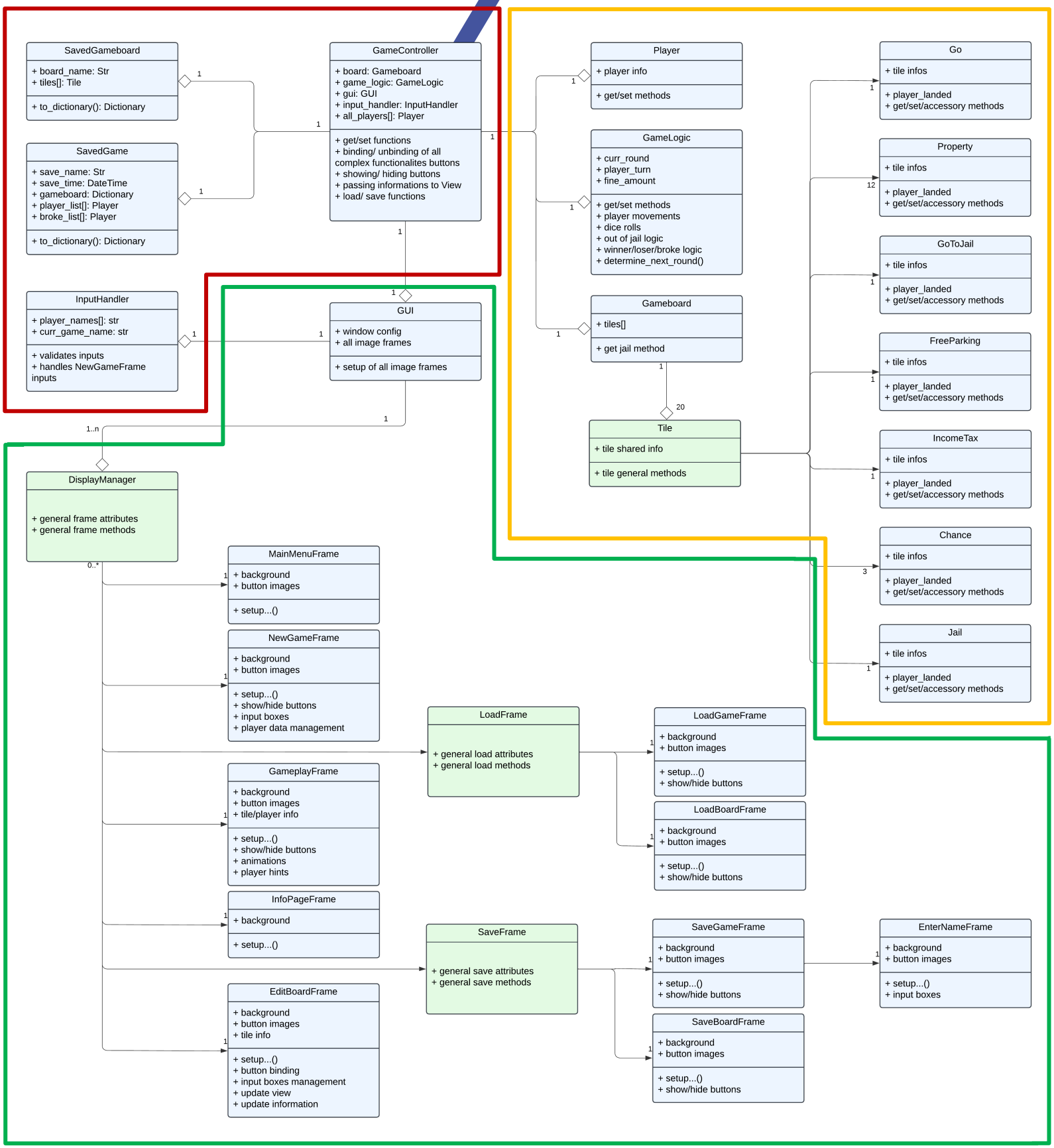
	GUI	• GUI

Class Diagram

The flowchart, found on the next page, presents a description about the construction of a Monopoly game including the game tiles and their types, game saving and loading frames and boards, and game and GUI logic classes. The GameController class acts as the main interface for the components of the game, whereas the GameLogic class is responsible for movement of players, dice, and stating the winner. The role of the GUI class is to initialize the various frames of the game, whereas the Tile class consists of attributes and functions for each kind of tile located on the gameboard.

In the following diagram, classes in the green are Abstract. Every other class in light blue is initialized and serves a purpose other than inheritance.

II. Code Structure and Relationships



II. Code Structure and Relationships

High level explanation of Classes and Methods

Starting from the leaves of the diagram, the explanation will take the reader to the brain of the operations. GameController is the heart of the operations, all classes belonging to Controller are circled in dark red. To the left, highlighted in yellow, all classes in the Model can be found. At the bottom all the classrooms in green are part of the View.

With our focus shifting on the Model, to the far right it is possible to see all Tile classes. Each tile contains all the information related to its functionality and self contains all methods regarding its working logic. More into the specifics each tile has getter and setter methods and a *player_landed* function that contains the logic the tiles apply on the player that lands on it.

All tiles have an abstract parent called *Tile* that has no functionality but contains all information and methods that all tiles share. All tiles converge into the Gameboard a class whose sole purpose is to contain all tiles and has methods for finding tiles locations in case a player decides to move them around while editing the board *(moving tiles not implemented US8-Old).* On the same level of the Gameboard class two other classes are found.

Player class contains all getter and setter methods for the Players objects and has variables initialized to set a default player. Finally, to conclude the Model explanation, GameLogic can be found. GameLogic is the class the contains the actual playing logic of the game. Inside this class the reader can find all methods to manage all the specific states a player can find himself into (i.e. Player in Jail, lands on a property, rolls the dices, won, lost…)

Moving onto the View component of the architecture, highlighted by the green lines, the reader can find all children classes to the DisplayManager. Each of the children classes manages a specific Frame of the game. While for spatial purposes Frame classes are arranged on multiple levels on the Class Diagram,

II. Code Structure and Relationships

they all can be considered on the same as they directly manage one specific and distinct instance of the game. Load and Save Frame classes are abstract and are parent classes for the sole purpose of reusability as the Game and Board loading and saving share many of the same functionalities.

All Frame classes have the same functionality of managing the specific view that is presented to the user and share the same working logic. For this reason, a general overview will be sufficient to understand how all the Frame classes work. In the following explanation I will use GameplayFrame class as a reference as it is the most complex.

On startup the frame initializes as follows. Images for the background, buttons, tile colours are pulled from the assets and initialized. Coordinates for placing these items on the canvas are sometimes hardcoded, other times referenced off other items positions. The class will find player and gameboard information loaded into its arrays. The loading is taken care by the Controller at an earlier date.
After this step, the GameplayFrame has all information necessary to initialize itself. The function *setup_new_gameplay_frame* is responsible for this action. Within this function accessory functions are called.

Accessory functions to Frames can be grouped into functions that create the buttons, functions that hide and display the buttons, functions that populate the Gameboard with information and functions that display player information, functions that manage the player movements.

On calling of the *setup_new_gameplay_frame* the expected execution is as follows: gameboard is populated, player information is displayed, players are moved to the correct starting position based on the game instance that is loaded. All buttons are created and displayed on the canvas. Canvas is returned along with an array containing all button IDs.

Buttons that only affect displayed information are bound in this class. All other buttons that call methods from the Model are bound within the GameController class that will be explained next.

II. Code Structure and Relationships

The Controller module is the heart of the operation. In the Class Diagram it is shown in red. There the SaveGame and SaveGameboard objects are built on request of the GameController. The classes contain all the necessary information to be saved in order to successfully restore a playing game instance and a gameboard layout. Save will pull information from the model and builds a JSON file while Load will read the specified JSON file and load the information into the model.

Before tying everything together in the GameController the last class that needs addressing is the InputHandler class that simply contains methods for handling player inputs on creating and starting a new game.

The GameController class, often referenced as the brains of the application, is the one managing every aspect of the game. On startup it is responsible for pulling information from the model and loading it into the View. This class has knowledge of which View is presented to the user and binds and unbinds all buttons that are necessary. Also has methods that couple Model functionalities and View functionalities for the view to update according to changes in the state of the game.
Finally, when the user asks for loading and saving of the game it displays the appropriate Views and calls for methods that carry through the user request.

Player Turn

Every player turn starts with the UI showing the GameplayFrame where the user has the entire overview of the board, containing all information regarding the tiles. The player also can see all other players and their amount of money, properties owned and their current square.
The game execution is halted. The user always has always two choices, *Save & Quit* or *Roll Dice*. If the user decides to click the latter, the turn execution will begin as shown in the flow diagram in following page.

II. Code Structure and Relationships