Bilkent University

Department of Computer Engineering

CS319
Object Oriented Software Engineering
Project Design Report

Katamino

Group 2-F

Sena Er - 21502112

Zeynep Sonkaya - 21501981

Hüseyin Orkun Elmas - 21501364

Utku Görkem Ertürk - 21502497

Bartu Atabek - 21602229

Table of Contents

1. Introduction

1.1 Purpose of the system

Katamino v.2.0 is a 2D brain training, puzzle game. It is designed in a way that lets the player work on increasing their problem solving abilities by continuously simulating it with puzzles and also give the player satisfaction to encourage the player to keep playing. Katamino v2.0 is based on the original Katamino game version, but we also add more functionalities to the game. Katamino v2.0 is a 2D desktop game with 3 different modes: multiplayer, single player classical and single player custom. Single player classical mode includes a level-based system, similar to the original Katamino. Katamino v2.0 also includes a custom mode to enable different board shapes for players. We also add multiplayer functionality to make users compete and make the game more interesting. Also, to encourage players, Katamino v2.0 includes score system in single player games, which players can also compete in the high score table.

Our design and our architecture will consist of subsystems to meet all functional requirements in our analysis and repeated above again and according to our constraints.

1.2 Design goals

We prefer designing the system before implementation, therefore we first establish our criteria and trade-offs for the design. These design goals will also guide our choices in the implementation process. Below, each of our design goals is detailed.

1.2.1 User Criteria

Ease of Learning

Our game will include a tutorial for any player who want to learn about the game. Since the game and controllers will be intuitive, the game will be easy to learn.

User - Friendliness

Katamino will have user friendly interface design which will enable players to easily understand control mechanisms such are need to drag and turn a pentomino. Our game design is similar to any game that is common in our daily life thus players can adapt the game usage without difficulty. The player will mostly use the mouse for any interaction with the game.

Performance

Performance in games have crucial value, in our design of the Katamino v2.0, players will not wait for a long time when they want to perform a specific action such as saving, loading user or moving pentominoes. Player's actions will result in less than 1 seconds on average.

1.2.2 Maintenance Criteria

Maintainability

Game design will be based on object-oriented programming concepts and design patterns which will allow us to implement the game in an extensible manner. Game's subsystems will have hierarchical structure so that new features will be added easily without any maintenance problem.

Robustness

Players should play our game without bugs that locks down the system that disrupt the flow of the game. To achieve this goal, we will review the user's interactions with the game through user interface that are designed for determining actions even though player does not perform the fully action like dragging the pentominoes to the exact location of the game board however our user interface will understand the location itself.

Additionally, our design is up to handle exceptions in a safe manner without reflecting to the player.

Modifiability
Change is frequent in software engineering, therefore we prioritized modifiability. Katamino v.2.0 designed to allow modification of the source code without having side effects. We make use of loose coupling principle, to make connected classes more independent and therefore prevent them get affected by a change in the other one.

Reuse of Components
According to our design we will reuse components for not include a subsystem that perform the same features as another subsystem does. Our components can be used in different segments.

1.2.2 Trade-offs

Functionality vs. Usability
One of the main target audience of Katamino v.2.0 is children, considering the preferences of target audience, we focus on ease of usability in the cost of functionality. We prefer simpler interfaces and interactions between the user and the system.

Cost vs. Robustness
Katamino is a software that cost only developers time and robustness of the game is highly important for players. We prefer robustness over cost of our time to give the user better experience with the product.

Efficiency vs. Portability
Katamino v.2.0 is only supported in desktop so portability will not be one of our main design goals but this preference will increase our efficiency. However, we will still include portability because of our decision to use Java which is supported in many computer types.

Cost vs. Reusability
We do not expect the game or much of the components of the system to be used in other systems. Therefore, we prefer to minimize costs such as developer time, instead of increasing reusability of the system.

1.3 Definitions

Java Virtual Machine (JVM): An abstract computing machine for running a Java program

Model View Controller (MVC): Separation of model, view and controller. Design pattern of Katamino. Model and controllers are explicitly defined in our project. Views are handled by .fxml files so that in this report there is no view classes.

Graphic User Interface (GUI): Form of Katamino user interface that allows users to interact with electronic devices through graphical icons and visual indicators

Java Development Kit (JDK): The full-featured SDK for Java.

Java Runtime Environment (JRE): A package of everything necessary to run a compiled Java program, including the Java Virtual Machine (JVM), the Java Class Library, and other infrastructures.

JavaFX: A software platform for creating and delivering desktop applications, as well as rich Internet applications that can run across a wide variety of devices. JavaFX make us to use .fxml files to manage view.

2. High-level software architecture

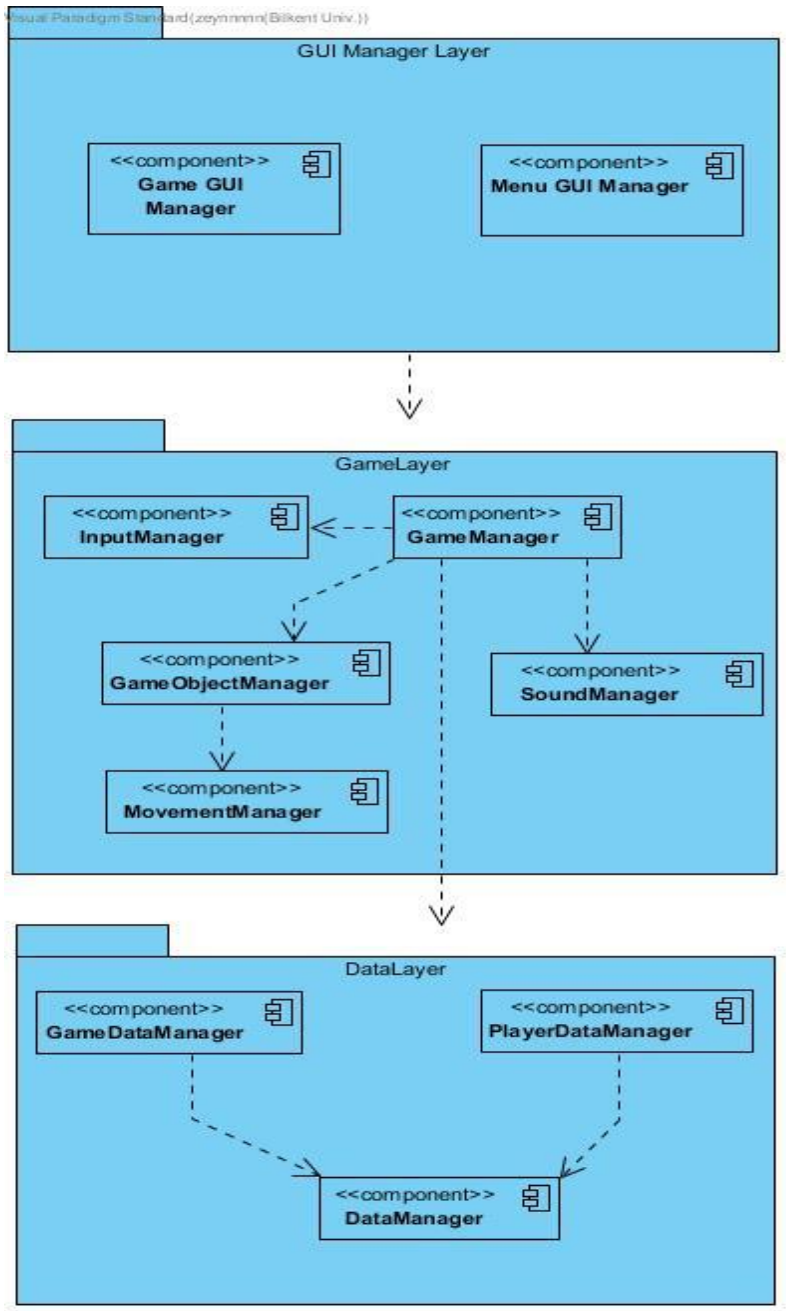

Visual Paradigm Standard(zeynnnnn(Bilkent Univ.))
GUI Manager Layer
<<component>> Game GUI Manager
<<component>> Menu GUI Manager
GameLayer
<<component>> InputManager
<<component>> GameManager
<<component>> GameObjectManager
<<component>> SoundManager
<<component>> MovementManager
DataLayer
<<component>> GameDataManager
<<component>> PlayerDataManager
<<component>> DataManager

Figure 1: High level software architecture showing GUI Manager, Game and Data layers

2.1 Subsystem decomposition

In this section, we will decompose Katamino system into subsystems, and group subsystems into layers to increase flexibility and maintainability of our design. We also plan to increase the extendibility of our system by using layers of subsystems for later modifications. Our design consists of 3 layers, namely; GUI Manager layer, Game Layer and Data Layer. We use a closed architecture model, in which each layer can only access the layer immediately below it.

GUI Manager Layer is responsible for all GUI tasks, such as displaying the game and taking inputs from the user. GUI Manager Layer has two subsystems: Game GUI Manager and Menu GUI Manager. Game GUI Manager is responsible for managing graphical user interface in the game such as displaying board, pentominoes and movements and Menu GUI Manager is responsible for menu interfaces such as main menu, single player classical menu. In some cases, the GUI manager will propagate these inputs to the Game Layer, specifically to the input manager subsystem. GUI manager is the top layer of our architecture, GUI manager layer is not used by any other layer.

Game Layer is mainly about game logic and movement. The Game Manager subsystem in the game layer is responsible for managing the game's state and communication of the other subsystems in the layer. The game objects management is responsible for managing board's data and pentominoes' data. This subsystem uses Movement Manager subsystem to calculate and update board and pentominoes' data with respect to the input from player. The input manager subsystem will take inputs from the GUI Manager Layer, process them and send information about the type of input and input to the Game Manager. Game manager will use game objects and movement manager subsystems to update the objects according to the specified input. Game Manager subsystem then will update Data Layer based on the input.

Data Layer focuses on the data of the game that we want to save. This can be Player data such as name, score, current level, available levels or game data such as places of pentominoes on the board, sound volume and current theme. Player Data Layer and Game Data Manager subsystems are responsible for managing mentioned data respectively.

Data Layer subsystem's purpose is to manage all of the game and player data to be saved and read from save files or board files to accomplish load functionality and write data to accomplish save functionality. Data Layer is our bottom layer, it does not depend on any other layer.

2.2 Hardware/software mapping

Katamino v.2.0 game will be implemented in Java 8, and we will use JDK 1.8+. We will use JavaFX libraries to implement GUI. JRE 1.8 is required to run the game. By using JRM, our game will be able run in Linux, Mac and Windows environments. Additionally, a mouse and keyboard will be enough to interact with the game. GUI Manager subsystem needs a mouse to get input for Game and Menu GUI Managers. Also, a keyboard hardware is necessary for Menu GUI Manager to get input about the Player identity. Data Manager subsystem is working with operating system for file management that occur during saving and loading game.

2.3 Persistent data management

In Katamino v2.0, we will save user data in a text file with JSON format in local storage. We will make use of the serialization and deserialization functionalities of Java standard libraries to load and save game. The save will include the current level, played levels, themes, boards, score, score board, completed levels and boards.

2.4 Access control and security

Katamino v2.0 does not handle any valuable information about the user, and the multiplayer side of the game is local. Therefore, we do not prioritize security. We only store user names to serve multiple of people on one computer.

2.5 Boundary conditions

2.5.1 Initializing Katamino v2.0

Players will use an executable file to start Katamino v2.0. Main menu will be shown without any requirements. After single player mode selection, if the folder that contains player information does not exist or corrupted, then the game creates that file, then game initialization continues.

If any of the player save files is removed then the player won't be shown the players list. Additionally, if the selected player save file is corrupted, then the game will show an error message, loading will fail and initialization will continue from main menu also corrupted file will be removed.

Other than those cases where a single game mode selected, the system will read selected player's data, and initialize game.

When multiplayer game mode selected, system will initialize the game.

When high score displaying selected in main menu, the file contains high score data in local is removed or corrupted, loading will fail and an error message will be displayed. After displaying the error, initialization will continue from main menu also corrupted file will be removed.

Another initialization failure occurs when a current Katamino.exe running and another Katamino.exe is tried to be started. In this case initialization of second Katamino will not be allowed and cause an error message displaying. After message, system will shut down the second Katamino window.

2.5.2 Shutdown

Players will be able to exit from "Exit game" button on the main menu. They can also exit game by using the "X" mark on the window. In both cases the game will automatically save before exiting.

2.5.3 Error Behavior

If the game cannot load data of board game from Board Game files because of the corruption of the data on the file during the game continuing states, we will display a run time error message.

Another error can occur when the player data is changed during the current player game. If the folder of the current player is deleted, a new folder will be created and new data of the player from game manager layer saved as a new player added.

3. Subsystem services

3.1 GUI Manager Layer

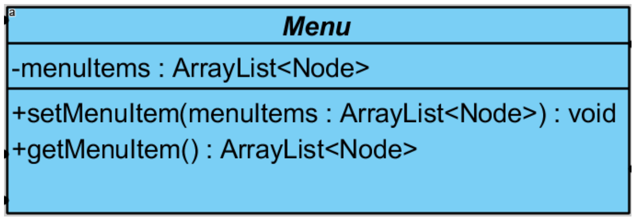
3.1.1 Game GUI Manager

GUI of the game handled by .fxml files and internal libraries .fxml files are connected to controller classes to bind user input to game data.

3.1.2 Menu GUI Manager

This Layer consist of Menus, setting and scoreboard. Menus have direct interaction with users. They are the basic classes that manage the directions for multiple menus and game sessions. Setting class used for sound operations and scoreboard shows the top three score of the game it is updated when a single game is over using updateScoreBoardMethod().

3.1.2.1 Menu


Menu
-menuItems : ArrayList<Node>
+setMenuItem(menuItems : ArrayList<Node>) : void
+getMenuItem() : ArrayList<Node>

Attributes:

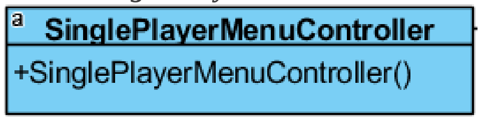
private ArrayList<Node> menuItems: This is the MenuItems**.**

Methods:

public void setMenuItem(ArrayList<Node> menuItems): This function sets menu items such as labels and buttons. This is necessary to customize the GUI objects.

public ArrayLıst<Node> getMenuItem():

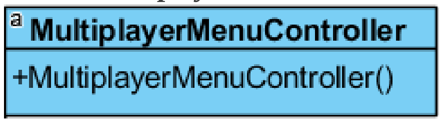
3.1.2.2 Single Player Menu Controller



Constructor:

public SinglePlayerMenuController: This is the constructor that construct the object in terms of configuration of the fxml files. In this case it changes labels.

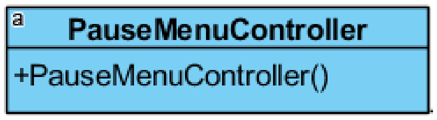
3.1.2.3 Multiplayer Menu Controller



Constructor:

public MultiPlayerMenuController: This is the constructor that construct the object in terms of configuration of the fxml files. In this case imageViews.

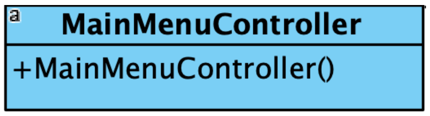
3.1.2.4 Pause Menu Controller



Constructor:

public PauseMenuController: This is the constructor that construct the object in terms of configuration of the fxml files. In this case it changes labels.

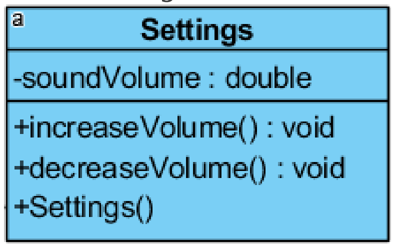
3.1.2.5 Main Menu Controller



Constructor:

public MainMenuController: This is the constructor that construct the object in terms of configuration of the fxml files. In this case it changes labels.

3.1.2.6 Settings



Attributes:

private double soundVolume: This attribute keeps current volume of the sound.

Constructor:

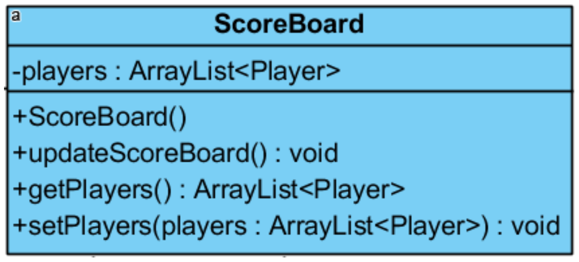
public Settings(): The default constructor of the Settings class.

Methods:

public void increaseVolume(): This function increases the volume 1% each call and will be called when user adjust volume using slider.

public void decreaseVolume(): This function decreases the volume 1% each call and will be called when user adjust volume using slider.

3.1.2.7 ScoreBoard


a
ScoreBoard
-players : ArrayList<Player>
+ScoreBoard()
+updateScoreBoard() : void
+getPlayers() : ArrayList<Player>
+setPlayers(players : ArrayList<Player>) : void

Attributes:

private ArrayList<Player>: This attribute stores the players of the game as arraylist of players.

Constructors:

public ScoreBoard(): The default constructor of the ScoreBoard class.

Methods:

public void updateScoreBoard(): This function is called when the high score of a player is updated through the Player class so that it refreshes the scoreboard view.

public ArrayList<Player> getPlayers(): This getter function returns the players attribute of the class in an arraylist of players.

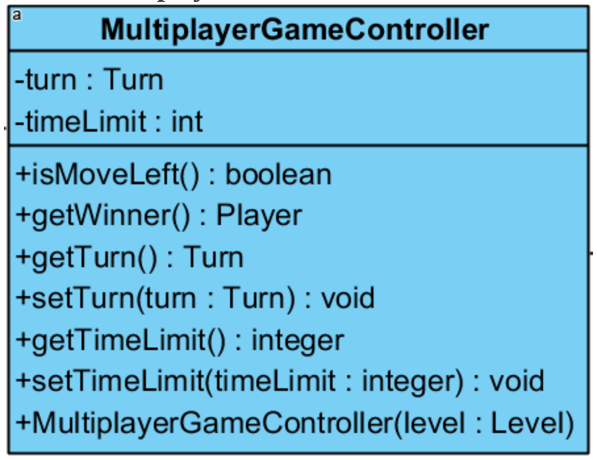
public void setPlayers(ArrayList<Player> players): This setter function sets the players of the game.

3.2 Game Layer

This is the main subsystem of the game. GameBoardController takes responsibility of manage the cells and low logic of the controls such as movePentamino(). This class connected to all acts on the game except menu actions. MultiplayerGameController is an extension for GameBoardController. By the agency of this class Katamino can be played multiplayer. Without this class, the game won't be broken just do not have multiplayer function. Turn is an enum, identifies the player turns. This enum is only used in multiplayer games. GameController is the highest level of abstraction of the game. GameModel which is in Data layer has 2 controllers one of them is GameBoardControler which handles low level work and other one is GameController that high level commands such as start and pause actions.

3.2.1 Game Manager

3.2.1.2 Multiplayer Game Controller


MultiplayerGameController
-turn : Turn
-timeLimit : int
+isMoveLeft() : boolean
+getWinner() : Player
+getTurn() : Turn
+setTurn(turn : Turn) : void
+getTimeLimit() : integer
+setTimeLimit(timeLimit : integer) : void
+MultiplayerGameController(level : Level)

Attributes:

private Turn turn: This attribute takes Turn enum and shows whose turn it is.

private int timeLimit: This attribute keeps maximum time limit for each player’s turn.

Constructor:

public MultiplayerGameController(Level level): This constructor takes Level class instance as a parameter so that MultiplayerGameController class can build game map.

Methods:

public boolean isMoveLeft(): This function is called to show that is there any move left for a given pentomino pool.

public Player getWinner(): This function calls isMoveLeft() function if isMoveLeft() returns true this fucntion will return winner based on opposite of the current player. Otherwise, this fuction will return null.

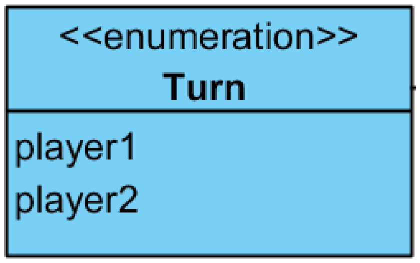
public Turn getTurn(): This is basic getter function which returns current Turn.

public void setTurn(turn: Turn): This is basic setter function that set current turn.

public integer getTimeLimit(): This is basic getter function which returns current Turn.

public void setTimeLimit(timeLimit:integer): This function sets time limit ,that is used throughout the game, of the each player.

3.2.1.3 Turn

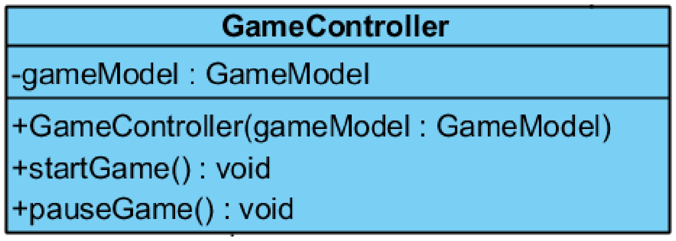


Enums:

player1: This enum stands for turn of first player.

player2 : This enum stands for turn of seconds player.

3.2.1.5 Game Controller



Attributes:

private GameModel gameModel: This attribute holds an instance of the game model that is being played.

Constructors:

public GameController(gameModel: GameModel): The constructor of GameController class which takes a GameModel instance as the parameter and initializes the attribute accordingly.

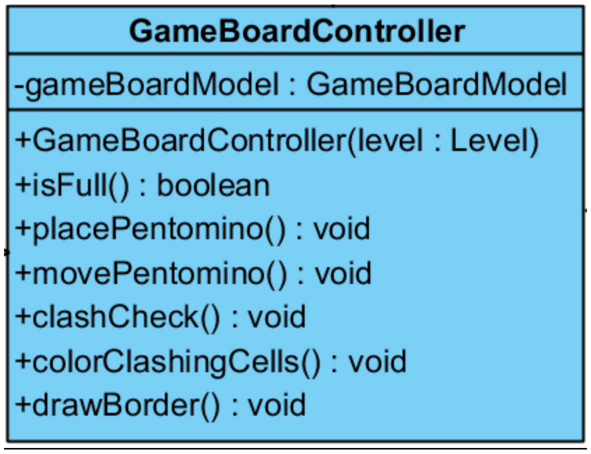
Methods:

public startGame(): This function is used for starting the game from the menu.

public pauseGame(): This function is used for pausing the game from the menu.

3.2.2 Game Objects Manager

3.2.1.1 Game Board Controller



Attribute:

private GameBoardModel gameBoardModel: This attribute keeps gameBoardModel which created in the constructor of GameBoardController constructor.

Constructors:

public GameBoardController(Level level): This constructor takes level to initialize level and create GameBoardControler instance.

Methods:

public boolean isFull(): This function check all Cells that if onBoard attribute equals true and cellNumber not equal to 0 this function will return true, otherwise false.

public void placePentomino(): This function places pentominoes when user release the mouse. If this is multiplayer game time count will be stop for the player.

public void movePentomino(): This function used for synchronization and adjusting cell attributes while user dragging the pentomino. Figure 2 is an example of both moving and placed pentomino depending of the perspective.

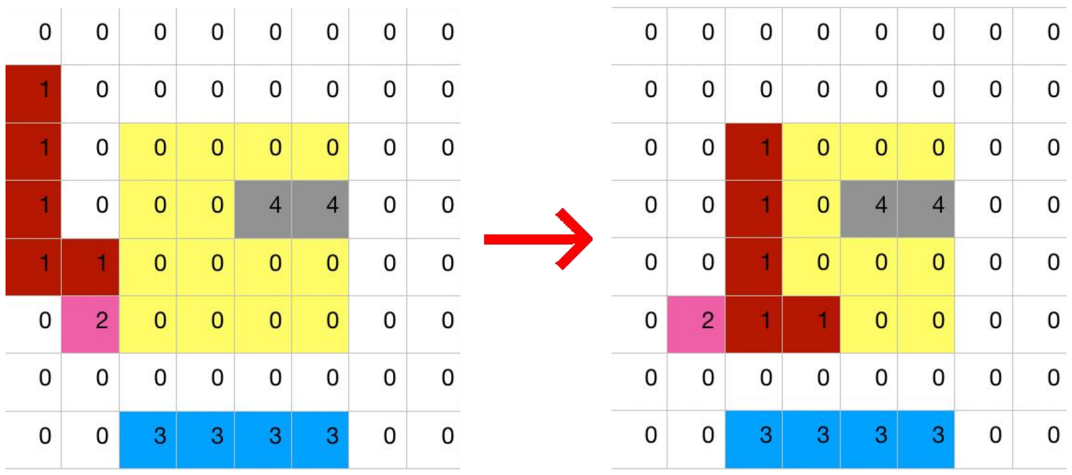


Figure 2

public void clashCheck(): This function checks if there is any clashes. Since Cell class implementsUndoableEdit interface this can be done without using extra variable or extra layer to show multi pentominoes.

public void colorClashingCells(): This function changes the color of pentominoes when the pentominoes clash.

public void drawBorder(): This function draws border if cellNumber of Cell attribute is 0.

3.2.3 Movement Manager

This subsystem is a logical component. It represents classes which is responsible for movement of pentominoes during the game. However, it has no explicit run-time equivalent.

3.2.4 Input Manager

This subsystem is a logical component. It represents classes which is responsible for inputs of users during the game. However, it has no explicit run-time equivalent.

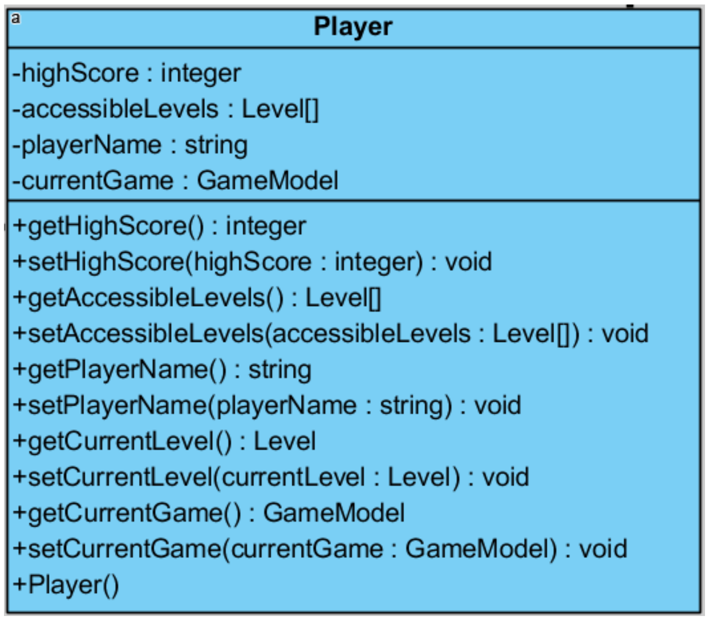
3.2.5 Sound Manager

This subsystem is a logical component. It represents classes which is responsible for sounds played during the game. However it has no explicit run-time equivalent.

3.3 Data Layer

3.3.1 Player Data Manager

3.3.1.1 Player


Player
-highScore : integer
-accessibleLevels : Level[]
-playerName : string
-currentGame : GameModel
+getHighScore() : integer
+setHighScore(highScore : integer) : void
+getAccessibleLevels() : Level[]
+setAccessibleLevels(accessibleLevels : Level[]) : void
+getPlayerName() : string
+setPlayerName(playerName : string) : void
+getCurrentLevel() : Level
+setCurrentLevel(currentLevel : Level) : void
+getCurrentGame() : GameModel
+setCurrentGame(currentGame : GameModel) : void
+Player()

Attributes:

private int highScore: This attribute keeps the high score of the player in integer type.

private Level[] accessibleLevels: This attribute stores all the levels accessible by the player, and they are stored as an array of levels.

private String playerName: This attribute represents the name of the player.

private GameModel currentGame: This attribute represents an instance of the game model.

Constructors:

public Player(): This is the default constructor of the Player class.

Methods:

public int getHighScore: This function returns the high score of the given player stored in the highScore attribute as an integer.

public void setHighScore(int highScore): This function updates the player's high score as they progress through the game by completing the levels. The updated high score will also be reflected to the ScoreBoard of the players and rearrange the board accordingly.

public Level[] getAccessibleLevels(): This function returns all the accessible levels in an array of Levels so that the player will be able to play them.

public void setAccessibleLevels(Level[] accessibleLevels): This function enables to set the accessible levels attribute of the class by setting it to the levels accessible to the player in the form of an array of Levels.

public String getPlayerName(): This getter function returns the name of the player.

public void SetPlayerName(String playerName): This setter function sets the player's name attribute to the given string.

public Level getCurrentLevel(): This getter function returns the current level that the player is playing as an instance of the Level class.

public void setCurrentLevel(Level currentLevel): This setter function updates the current level of the player as they keep playing different levels. The functions take the new current level as an instance of the Level class.

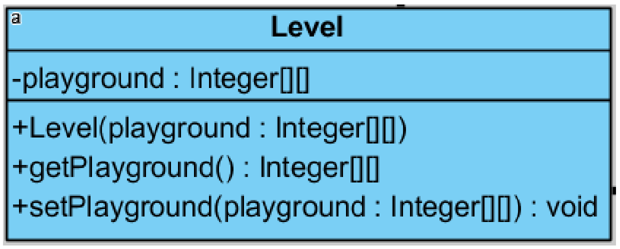
public void setCurrentGame(GameModel gameModel): This setter function updates the current game attribute of the class with the given value.

public GameModel getCurrentGame(): This getter function returns the current gameModal of the player.

3.3.2 Game Data Manager

There are four game objects in Katamino: Cell, Game Model, Game Board Model and Player. Cell is the main component of the game. Whole game board consists of cells as grid base. So that pentominoes and board will be combination of cells. Player is the model for each user. Whenever user asks for a player, this leads to creating of Player instance. Game Model is the model for the basic game which has GameBoardModel.

3.3.2.1 Level


Level
-playground : Integer[][]
+Level(playground : Integer[][])
+getPlayground() : Integer[][]
+setPlayground(playground : Integer[][]) : void

Attributes:

private Integer[][] playground: This attribute keeps the game board for the specified level indicating the cell mapping of the tiles (pentominoes).

Constructors:

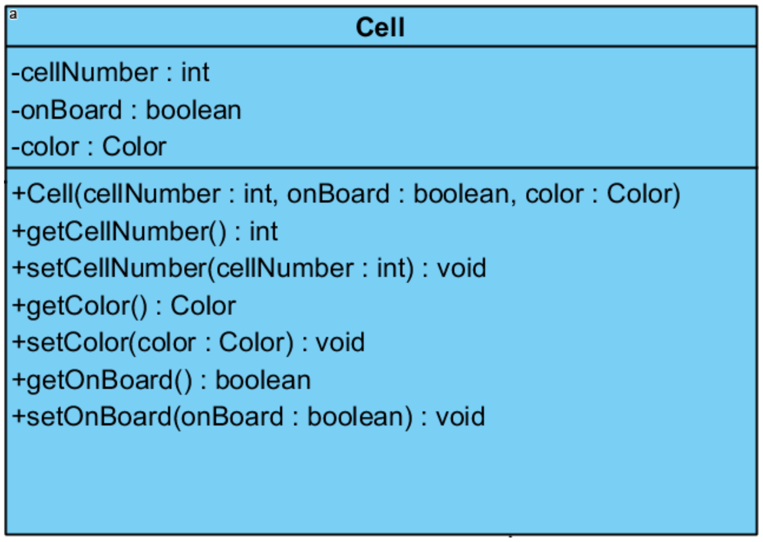
public Level(Integer[][] playground): This is the constructor that takes integer mapping of the playground such as Figure 3.

Methods:

public Integer[][] getPlayground(): This function getter of the playground attribute returns the cell mapping of the game board as an 2D integer array.

public void setPlayground(Integer[][] playground): This setter function changes the playground property to the given cell mapping inputted as 2D integer array.

3.3.2.2 Cell


Cell
-cellNumber : int
-onBoard : boolean
-color : Color
+Cell(cellNumber : int, onBoard : boolean, color : Color)
+getCellNumber() : int
+setCellNumber(cellNumber : int) : void
+getColor() : Color
+setColor(color : Color) : void
+getOnBoard() : boolean
+setOnBoard(onBoard : boolean) : void

Attributes:

private int cellNumber: This attribute keeps number of the cell to track of interaction of the cells. This is the number shown on *Figure 3*.

private boolean onBoard: These attributes show the cell is on board or outside of the board.

private Color color: This attribute keeps the color of the cell using Java Color.class.

Constructor:

public Cell(int cellNumber, boolean onBoard, Color color): This constructor takes initial condition of the cell in terms of cell number, onBoard or not and color of the cell.

Methods:

public String getColor(): This function getter of the color attribute GameBoardController is the one of the caller of this function.

public void setColor(Color color): This function is used for future change of the cell color.

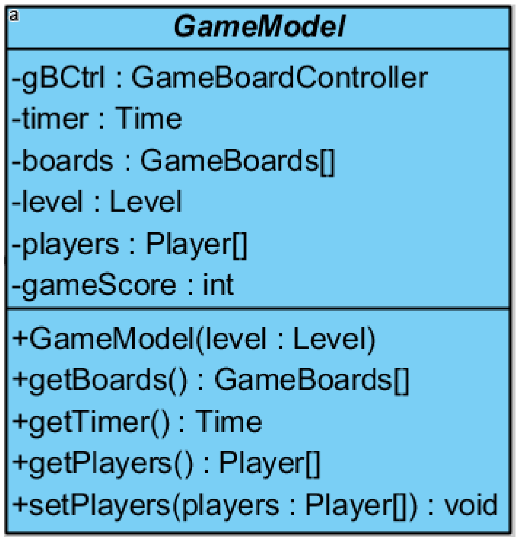
public boolean getCellNumber(): This function getter for cell number.

public void setCellNumber(int cellNumber): This is the setter for cellNumber attribute this function will be dynamically changed during movement of the pentomino by movePentomino() function of the GameBoardController().

public boolean getOnBoard(): This is the getter function for OnBoard attribute that gives position of the cell relative to board.

public void setOnBoard(boolean board): This is the setter function for OnBoard attribute.

3.2.1.4 Game Model



Attributes:

private GameBoardController gBCtrl: This attribute represents the instance of the game board controller class.

private Time timer: This attribute shows the time passed while playing the game.

private GameBoards[] boards: This attribute represents the collection of all available game boards as an array of game boards.

private Level level: This attribute represents the current level in the game.

private Player[] players: This attribute represents the players of the game stored as an array of players.

Constructors:

public GameModel(Level level): This is the constructor which takes a level instance as the parameter and sets the level attribute accordingly.

Methods:

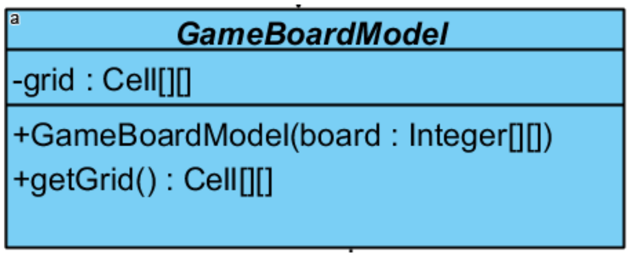
public GameBoards[] getBoards(): This getter function of the boards attribute returns the game boards and an array of GameBoards.

public Time getTimer(): This getter function of the timer attribute returns the value in the attribute in time formal.

public Player[] getPlayers(): Get the current players of the game as an array.

public void setPlayers(Player[] player): Set the current players of the game as an array. If game is multiplayer there will be two players in the array. If game is single player there will be one player in this array.

3.2.2.1 GameBoardModel

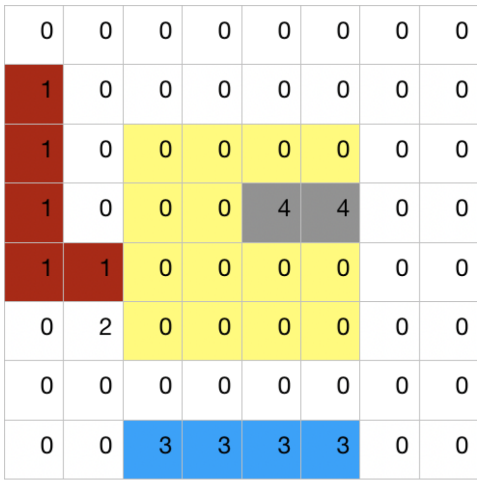


Attributes:

private Cell[][] grid: This attribute keeps all grids on the whole view.

Constructor:

public GameBoardModel(Integer[][] playground): This is the constructor that takes integer mapping of the playground such as Figure 3.



0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	4	4	0	0
1	1	0	0	0	0	0	0
0	2	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	3	3	3	3	0	0

Figure 3

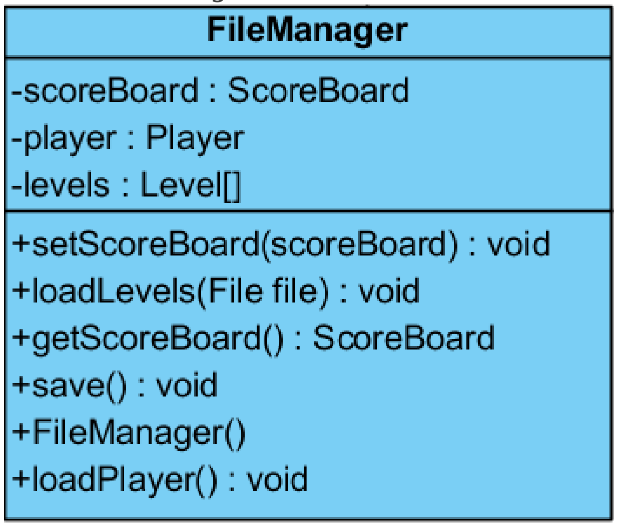
Methods:

public Cell[][] getGrid(): This is the standard getter used by GameBoardController. By this method GameBoardControler can access the Grid and determine the positioning of the items as a whole.

3.3.3 Data Manager

The Data Manager subsystem layer consists of the FileManager class and it is responsible for the handling of data manipulations and basic CRUD functionalities with the help of serializable interface. In this subsystem the FileManager holds instances of ScoreBoard, Player and Level classes and is responsible for loading game map files as levels, updating the score board and recording the data changes to the system.

3.3.3.1 FileManager



Attributes:

private ScoreBoard scoreBoard: This attribute keeps the scores of the players as an instance of the ScoreBoard class.

private Player player: This attribute represents the current player.

private Level[] levels: This attribute represents the available levels of the game represented as an array of the Level class.

Methods:

public ScoreBoard getScoreBoard(): This getter function returns the attribute scoreBoard representing the scoreboard of the players.

public void setScoreBoard(ScoreBoard scoreBoard): This setter function updates the scoreboard with the new one taken as a parameter.

public void loadLevels(File file): This function loads the game levels from the txt file and sets the levels attribute accordingly to the uploaded levels.

4. Low-level design

4.1 Object design trade-offs

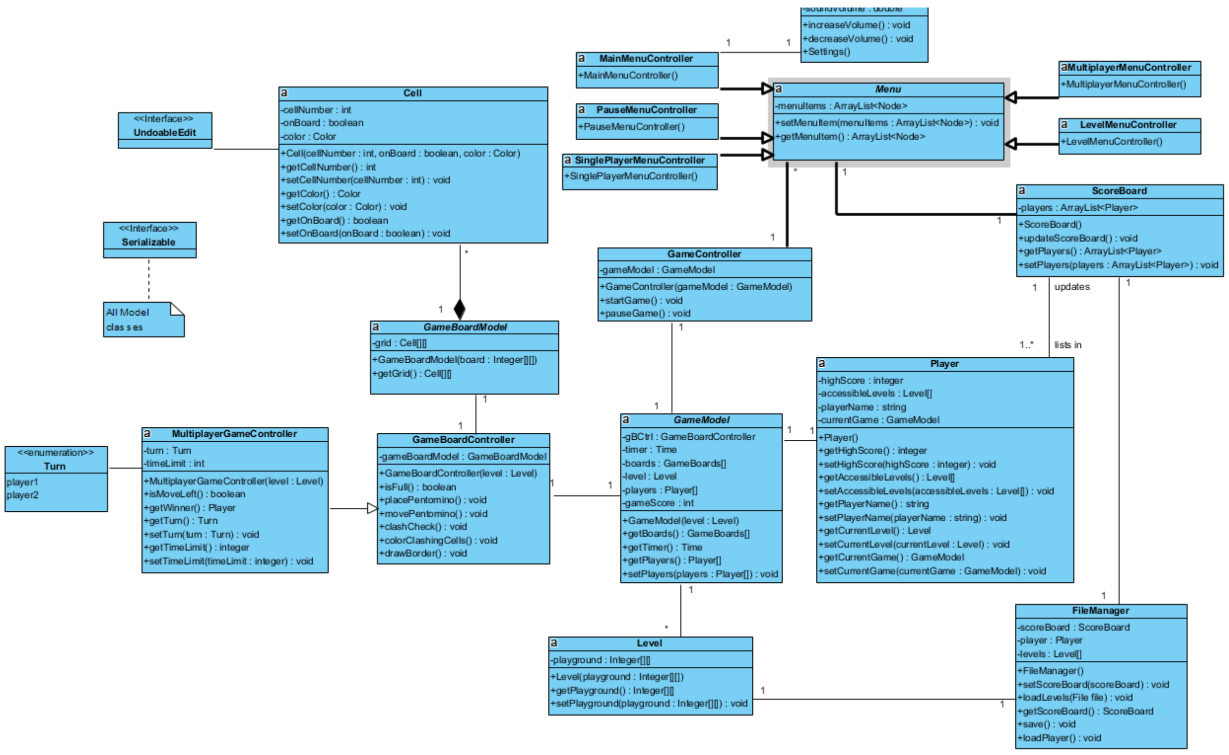
Inheritance vs Composition

In the design of Katamino v2.0, we value composition over inheritance to increase flexibility of design. Additionally, with the use of composition we plan to increase the maintainability and modifiability of our code.

Loose coupling vs tight coupling

In designing Katamino v2.0, we prefer loose coupling over tight coupling to decrease the dependencies between classes and ease the testing of our code. Moreover, with the use of loose coupling we plan to increase maintainability, modifiability and reusability of our code.

4.2 Final object design



+increaseVolume() : void
+decreaseVolume() : void
+Settings()
MainMenuController
+MainMenuController()
PauseMenuController
+PauseMenuController()
SinglePlayerMenuController
+SinglePlayerMenuController()
Menu
-menuItems : ArrayList<Node>
+setMenuItem(menuItems : ArrayList<Node>) : void
+getMenuItem() : ArrayList<Node>
MultiplayerMenuController
+MultiplayerMenuController()
LevelMenuController
+LevelMenuController()
<<Interface>>
UndoableEdit
Cell
-cellNumber : int
-onBoard : boolean
-color : Color
+Cell(cellNumber : int, onBoard : boolean, color : Color)
+getCellNumber() : int
+setCellNumber(cellNumber : int) : void
+getColor() : Color
+setColor(color : Color) : void
+getOnBoard() : boolean
+setOnBoard(onBoard : boolean) : void
<<Interface>>
Serializable
All Model classes
ScoreBoard
-players : ArrayList<Player>
+ScoreBoard()
+updateScoreBoard() : void
+getPlayers() : ArrayList<Player>
+setPlayers(players : ArrayList<Player>) : void
GameController
-gameModel : GameModel
+GameController(gameModel : GameModel)
+startGame() : void
+pauseGame() : void
GameBoardModel
-grid : Cell[][]
+GameBoardModel(board : Integer[][])
+getGrid() : Cell[][]
updates
lists in
Player
-highScore : integer
-accessibleLevels : Level[]
-playerName : string
-currentGame : GameModel
+Player()
+getHighScore() : integer
+setHighScore(highScore : integer) : void
+getAccessibleLevels() : Level[]
+setAccessibleLevels(accessibleLevels : Level[]) : void
+getPlayerName() : string
+setPlayerName(playerName : string) : void
+getCurrentLevel() : Level
+setCurrentLevel(currentLevel : Level) : void
+getCurrentGame() : GameModel
+setCurrentGame(currentGame : GameModel) : void
<<enumeration>>
Turn
player1
player2
MultiplayerGameController
-turn : Turn
-timeLimit : int
+MultiplayerGameController(level : Level)
+isMoveLeft() : boolean
+getWinner() : Player
+getTurn() : Turn
+setTurn(turn : Turn) : void
+getTimeLimit() : integer
+setTimeLimit(timeLimit : integer) : void
GameBoardController
-gameBoardModel : GameBoardModel
+GameBoardController(level : Level)
+isFull() : boolean
+placePentomino() : void
+movePentomino() : void
+clashCheck() : void
+colorClashingCells() : void
+drawBorder() : void
GameModel
-gBCtrl : GameBoardController
-timer : Time
-boards : GameBoards[]
-level : Level
-players : Player[]
-gameScore : int
+GameModel(level : Level)
+getBoards() : GameBoards[]
+getTimer() : Time
+getPlayers() : Player[]
+setPlayers(players : Player[]) : void
Level
-playground : Integer[][]
+Level(playground : Integer[][])
+getPlayground() : Integer[][]
+setPlayground(playground : Integer[][]) : void
FileManager
-scoreBoard : ScoreBoard
-player : Player
-levels : Level[]
+FileManager()
+setScoreBoard(scoreBoard) : void
+loadLevels(File file) : void
+getScoreBoard() : ScoreBoard
+save() : void
+loadPlayer() : void

Figure 4

4.3 Packages

Java.util

This package will be used for collections for data management on memory. Such as the current gameboard's shape and pentominoes locations on the board. From this package, we will also use IO subpackage for loading from the and saving data to computer, such as player data, highscore data and custom board data.

Javafx.scene

This package will be used for support to scene graph. From this package, we will also use sub-packages such as input, layout and image. We will use input subpackage to use handlers for mouse and keyboard events, layout to fit GUI objects, image to support images in GUI.

Javafx.event

We will use this package to handle javaFX events, such as mouse clicks and key presses.

Javafx.application

This package will be used for the JavaFX application life cycle.

4.4 Class Interfaces

ActionListener

This interface is responsible for catching any action with the corresponding action events. This interface will be working with Menu class. This interface will be used to enable buttons in the menus.

MouseListener

This interface will be used to catch mouse click actions. MouseListener will be implemented by inputManager class to use the input in game. These inputs will be used to calculate movements and new positions of pentominoes.

KeyListener

KeyListener interface is responsible for catching keyboard events. KeyListener will be implemented by input manager. These input

java.io.Serializable

The classes which will be converted their states to byte streams (serializing) will implement Serializable interface. These byte streams are used to obtain the original object afterwards (deserializing). The subclass also implements Serializable interface if its parent also implements Serializable.

Serialization and deserialization will be used to save a player's data such as high score or progression in the game. Data Manager subsystem will be responsible for managing this process.

javax.swing.undo (UndoableEdit)

This interface is used for getting support for undo/redo functions of pentomino movements about accessing older values of locations and color of pentominoes.

5. References

[1] "Overview (Java Platform SE 8)", *Docs.oracle.com*, 2018. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/overview-summary.html. [Accessed: 08- Nov- 2018].

[2] "Overview (JavaFX 8)", *Docs.oracle.com*, 2018. [Online]. Available: https://docs.oracle.com/javase/8/javafx/api/overview-summary.html. [Accessed: 08- Nov- 2018].