**Vietnamese National University HCMC**

**International University**



**School of Computer Science and Engineering**



**PROJECT REPORT**

**MINESWEEPER GAME**

**Course: Data Structures & Algorithms**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Group members:**

Trần Đình An ITDSIU20113

Đinh Quang Hiển ITITIU20201

Nguyễn Vạn Huy ITITIU20215

Lê Thế Duyệt ITITIU19112

**Table of Contents**

**Chapter 1: Introduction**

1. About the game
2. Objectives
3. Techniques and tools
4. Contribution table

**Chapter 2: Project Analysis**

1. Class diagrams
2. control
3. CoreGame
4. view
5. GFrame
6. GPanel
7. PanelNotification
8. LabelNumber
9. PanelPlayer
10. MineButton
11. ButtonSmile
12. model
13. LoadData

**Chapter 3: Conclusion**

1. Limitations

**References**

**Chapter 1: Introduction**

The intention of this report is to give detailed documentation of the project that was done as part of the Data Structures and Algorithms course requirement. The project involved creating and implementing the popular game Minesweeper to obtain a better knowledge of the key concepts and principles covered in the course, as well as enhancing our abilities in constructing a software project utilizing an object-oriented approach.

1. **About the game**

Minesweeper is a well-known single-player puzzle game created by Microsoft in the 1990s. It has subsequently become a classic and has been an integral part of PCs since Windows XP. The aim is to clear the board without detonating any mines, with hints about the number of mines in each area to help the player’s decision making. The game requires the player to exercise careful judgment and logic while making quick decisions to avoid mines. It is a fun way to pass the while also challenging yourself to develop your logical thinking abilities.

The game is played by choosing a cell on the board and figuring out future cells that are safe to open based on the information provided by the opened cells. The player acquires more information to solve the board repeatedly by using the minesweeper hints. These clues indicate the number of mines that are next to each cell. This data helps the player determine which cells are safe to open and which contain mines.

1. **Objectives**

* Improve our knowledge of data structures and algorithms covered in the course.
* Enhance our problem-solving and critical thinking skills through designing a game.
* Improve collaboration skill by working together on a programming project.
* Gain experience in implementing game logic, user interface, and game mechanics in Java.
* Develop debugging and testing skills to ensure the game's correctness and robustness.

1. **Techniques and tools**

* Java: a high-level, object-oriented programming language commonly utilized in the development of desktop, mobile, and online applications.
* Various libraries, such as Java AWT, Swing, etc. are used to create graphical user interfaces, as well as to perform file input/output operations.
* Data structures such as arrays and stacks are used in the game implementation.

1. **Contribution table**

|  |  |  |
| --- | --- | --- |
| **Name** | **Responsibility** | **Contribution** |
| Trần Đình An | Build the code for the base game, implement the undo feature, prepare for the presentation and the report. | 25% |
| Đinh Quang Hiển | Implement the timer/bomb counter function, build class diagrams, participate in writing the report. | 25% |
| Nguyễn Vạn Huy | Implement the flag and double click function, build class diagrams. | 25% |
| Lê Thế Duyệt | Implement the smile button and difficulty option function, participate in writing the report. | 25% |

**Chapter 2: Project Analysis**

* 1. **Class diagrams**

**A picture containing text, screenshot

Description automatically generated**

<https://drive.google.com/file/d/1wv2g-dvHvPhhRLvDCFDg24P-HkHMozDb/view?usp=sharing>

* 1. **control**

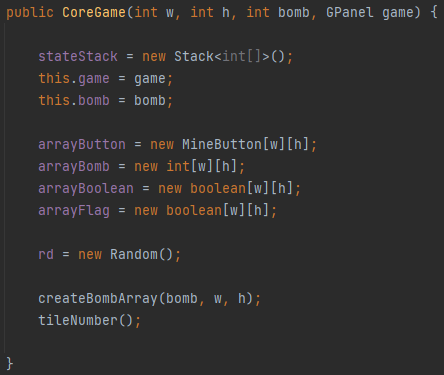
This package contains the CoreGame class, which implements the game's core logic and features.

1. **CoreGame**

This class maintains the game state, which includes the game board, bomb placement, flagging cells, opening cells, checking for win or loss conditions, and handling various game interactions. The class also keeps track of the essential data structures, such as arrays to represent the game board, boolean arrays to track cell states, and stacks to provide undo funtion. It also includes methods for updating the game's user interface components, including buttons, labels, and panels, to reflect the current game state.

**Constructor:** The object creation for this class takes 4 parameters: width of the cell array (w), height of the cell array (h), number of bombs (bomb), and the game panel (game).

The constructor first creates a new empty stack (stateStack) to store game state for the undo function, then assigns the provided game panel (game), bombs (bomb) to the local variables of this class. Various arrays (arrayButton, arrayBomb, arrayBoolean, and arrayFlag) are initialized with the specified width and height (w and h). It also creates a new instance of the Random class (rd) to generate random numbers. Finally, 2 methods: createBombArray() (passing the number of bombs (bomb), width (w), and height (h)) and tileNumber() was called to randomly place the specified number of bombs on the game board, calculate and assign the numbers indicating the adjacent bomb count for each non-bomb cell on the game board.



**Double click:** This method handles the revealing of neighboring cells when a user performs a double-click on a cell. It ensures that cells are revealed according to the Minesweeper rules, considering the presence of bombs and the numbers indicating the neighboring bomb counts. The method also checks if any cell surrounding the cell double clicked are flagged so that the flagged cells are not revealed when double clicked.

The method initializes ‘isBomb’ as false to track neighboring cells with bombs. It iterates over a 3x3 grid centered around the double-clicked cell. Valid neighboring cells are checked for boundaries and flagged status. If not flagged, the method handles bomb and non-bomb cases. For a bomb, the corresponding button is updated and marked as revealed. If not a bomb, it recursively reveals connected cells or updates the button with the number of neighboring bombs. After iterating, if ‘isBomb’ is true, buttons for bomb cells are updated and false is returned. If ‘isBomb’ is false, true is returned, indicating no neighboring bombs.

A picture containing text, screenshot, font

Description automatically generated

**Flag a tile:** The method allows the player to toggle the flag status of a cell, keeping track of the flag count and updating the game interface accordingly. It is mainly used for the player to keep track of any predicted cells containing bombs.

The method first takes two parameters ‘i’ and ‘j’ to determine the cell the player wanted to flag. If the cell is not revealed, it handles the flag status. If the cell is already flagged, the flag is removed, and the display is updated. If the cell is not flagged and there are available flags, the flag is placed, and the display is updated.

A screen shot of a computer code

Description automatically generated with medium confidence

**Open a cell:** The method is used to reveal a cell at the specified indices (i, j) in the game.

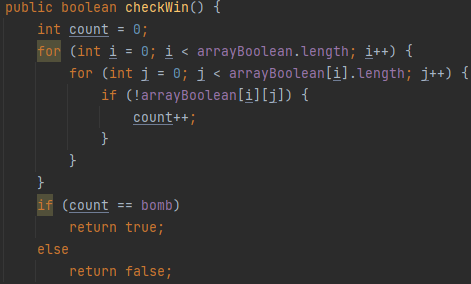
The method first checks if the game is not lost or won. If the cell is unrevealed, it proceeds with opening it. If arrayBomb[i][j] is 0, the cell is further opened by marking it as revealed, updating its button, and repainting the display. The method then checks if this move leads to a win. If so, ‘isWin’ is set to true and returns false. Next, the method recursively opens neighboring cells within the game boundaries and has not yet been revealed. If arrayBomb[i][j] is not 0, it checks if it's not -1 (indicating a bomb). In this case, the cell is revealed, updates its button, and checks for a win. If the player has won, the method sets ‘isWin’ to true and returns false. If arrayBomb[i][j] is -1, indicating a bomb, the button is updated to show a bomb symbol, ‘isLost’ is set to true, and buttons are updated for all bomb cells. It then returns false. If none of the above conditions apply, the method checks for a win. If the player has won, it sets ‘isWin’ to true and returns false. Otherwise, it returns true. If the game is already lost or won, the method returns false.

A picture containing text, screenshot, font

Description automatically generated

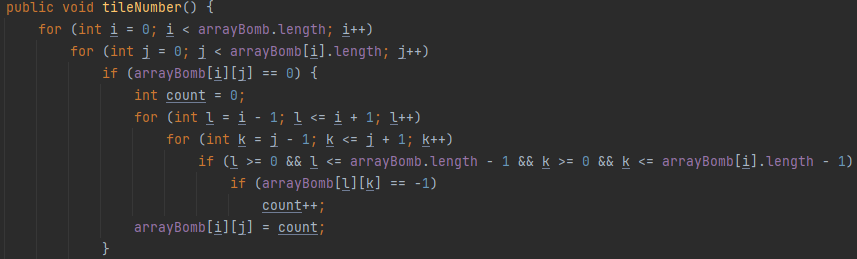
**Check if the player wins:** This method counts the number of unrevealed cells and checks if it matches the total number of bombs, indicating that the player has successfully revealed all non-bomb cells and won the game.

It first initializes a variable count to 0, which will keep track of the number of unrevealed cells. The method then iterates over the ‘arrayBoolean’ 2D array, which represents the status of each cell (whether it has been revealed or not). For each cell, it checks if arrayBoolean[i][j] is false, indicating that the cell is unrevealed. If it is unrevealed, ‘count’ is incremented. After iterating through all cells, the method compares the value of ‘count’ with the total number of bombs (bomb). If ‘count’ is equal to ‘bomb’, it means that all non-bomb cells have been revealed, and the player has won the game. In this case, the method returns true. Otherwise, it turns out false.



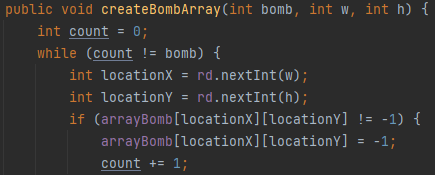
**Assign number to each cell:** This method scans each empty cell in the ‘arrayBomb’ array, counts the number of neighboring bombs, and updates the cell with the corresponding count. This process populates the ‘arrayBomb’ array with numbers indicating the number of neighboring bombs for each cell.

The method first iterates over each cell in the ‘arrayBomb’ array using nested for loops. For each cell, it checks if the value of arrayBomb[i][j] is 0, indicating an empty cell without a bomb. If the cell is empty, the method initializes a variable ‘count’ to 0, which will store the number of neighboring bombs. It then iterates over the neighboring cells of the current cell, forming a 3x3 grid centered around it. The method checks if each neighboring cell is within the boundaries of the game board by verifying that the row index l is between 0 and the length of the ‘arrayBomb’ array minus 1, and the column index k is between 0 and the length of the arrayBomb[i] array minus 1. If the neighboring cell contains a bomb (the value is -1), ‘count’ is incremented. After counting the number of neighboring bombs, the method assigns the value of count to arrayBomb[i][j], indicating the number of neighboring bombs for the current cell.



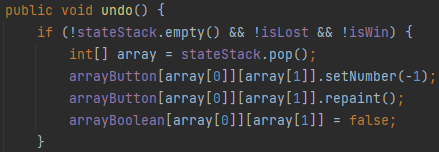
**Scatter the bombs around the game’s map:** The method is responsible for randomly placing bombs in the arrayBomb array.

Firstly, a ‘count’ variable is initialized to 0. The method then enters a loop that continues until the ‘count’ reaches the desired number of bombs (bomb). In each iteration, it generates random ‘locationX’ and ‘locationY’ coordinates within the boundaries of the array using the nextInt() method of the Random class. The method checks if the cell at the generated coordinates in the ‘arrayBomb’ array does not already contain a bomb (value -1). If the condition is satisfied, it assigns the value -1 to that cell, indicating the presence of a bomb, and increments ‘count’ by 1. The loop continues until the desired number of bombs has been placed in the array. This ensures that exactly ‘bomb’ number of bombs are randomly distributed in the ‘arrayBomb’ array.



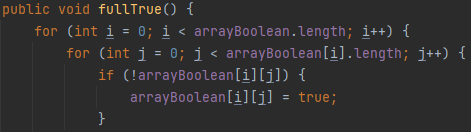
**Undo a move:** This method effectively reverts the game state to the previous move, hiding the information and status changes associated with that move.

First, the method checks if the ‘stateStack’ is not empty (i.e., there are moves to undo) and if the game is not already lost or won. If these conditions are met, it proceeds to undo the previous move. The method retrieves the coordinates of the previous move from the top of the ‘stateStack’ by calling the pop() method, which removes and returns the top element of the stack. The coordinates are stored in an array called ‘array’. Next, the method updates the button at the corresponding coordinates in the ‘arrayButton’ array. It sets the number of the button to -1, which represents an unrevealed cell, using the setNumber(-1) method. Then, it calls the repaint() method to update the button's display. Additionally, the method sets the corresponding boolean value in the ‘arrayBoolean’ array to false, indicating that the cell is no longer revealed.



**Set all cell statuses to ‘revealed’:** This method ensures that all cells are marked as revealed, regardless of their previous state.

The method uses nested for loops to iterate over each cell in the ‘arrayBoolean’ array. It starts with the outer loop iterating over the rows, and the inner loop iterating over the columns. For each cell, the method checks if the boolean value stored in arrayBoolean[i][j] is false, indicating that the cell has not been revealed. If the cell is not revealed, the method sets its corresponding boolean value to true using the assignment statement arrayBoolean[i][j] = true.



* 1. **view**

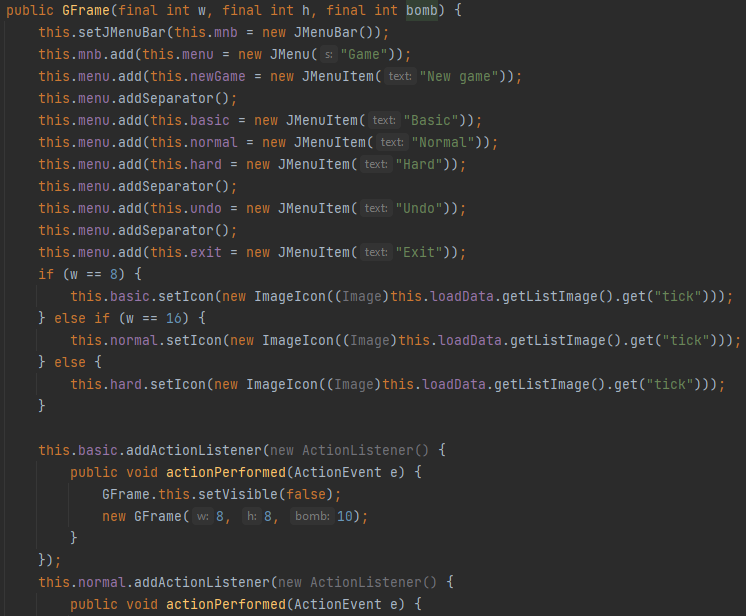
This package contains classes related to the graphical user interface (GUI) of the Minesweeper game.

1. **GFrame**

This class serves as the primary user interface for the game application. It creates a window where the game is displayed and provides essential functionalities such as setting the game dimensions, defining the number of mines, and allowing the user to start a new game or exit the application. The class also incorporates a menu bar with different menu items for game customization. This class also contains the main method which is used to run the game.

**Constructor:** This constructor is responsible for initializing the GFrame object, setting up the Minesweeper game frame, and displaying it to the user. It takes 3 parameters to create a new object for this class: width of the cell array (w), height of the cell array (h) and the number of bombs (bomb).

It first creates menu items such as ‘New game’, ‘Basic’, ‘Normal’, ‘Hard’, ‘Undo’, and ‘Exit’, and adds them to the menu. Depending on the value of the ‘w’ parameter, the constructor sets the tick icon for the corresponding difficulty menu item (‘basic’, ‘normal’, or ‘hard’). It also defines action listeners for these menu items to handle user interactions. When a difficulty menu item is selected, the current frame is set to invisible, and a new ‘GFrame’ object is created with the appropriate grid size and number of bombs. Additionally, an action listener is defined for the ‘Undo’ menu item to handle undo functionality. Finally, the constructor adds a ‘GPanel’ to the frame, sets the frame's icon image, adjusts the frame size, makes it non-resizable, centers it on the screen, sets the default close operation, and makes the frame visible to the user.



**The main method:** This is the entry point of the program. The method serves as the starting point for the execution of the Minesweeper game. It creates an instance of the ‘GFrame’ class, which represents the game window with an 8x8 grid and 10 bombs. By calling new GFrame(8, 8, 10), the constructor of the GFrame class is invoked, initializing the game frame, setting up the menu bar, and displaying the Minesweeper game to the user. The program execution continues from there, allowing the user to interact with the game and play Minesweeper.



1. **GPanel**

This class represents the main panel of the Minesweeper game. It extends the JPanel class and implements the MouseListener interface to handle mouse events. The purpose of the GPanel class is to provide the graphical user interface for the Minesweeper game and handle user interactions. It contains various components and methods related to the game panel.

**Constructor:** This constructor initializes an instance of the GPanel class with the specified width (w), height (h), number of bombs (bomb), and a reference to the parent GFrame object (gameFrame).

It first assigns ‘gameFrame’, ‘bomb’, ‘w’ and ‘h’ to instance variables which have the same names in the class. The constructor then creates a new instance of the ‘CoreGame’ class, passing the width, height, number of bombs, and a reference to the current ‘GPanel’ object (this) as arguments. The layout manager for the ‘GPanel’ object is then set to a ‘BorderLayou’t with horizontal and vertical gaps of 20 pixels. The constructor creates and adds a ‘PanelNotification’ object to the north position of the ‘GPanel’ object. This panel displays game-related information and is assigned to the ‘p1’ instance variable. Additionally, the constructor creates and adds a ‘PanelPlayer’ object to the center position of the ‘GPanel’ object. This panel contains the game grid and mine buttons and is assigned to the ‘p2’ instance variable.

A screen shot of a computer

Description automatically generated with medium confidence

**Handle the mouse pressed event:** This method is an event handler that gets triggered when a mouse button is pressed on the ‘GPanel’ object.

First, it sets the stage of the smile button in the ‘PanelNotification’ object to 3 (which is the ‘wow’ state) and repaints it. Then, it retrieves ‘arrayButton’ from the ‘PanelPlayer’ object. Next, it iterates over each cell and checks if the left mouse button was pressed on an unflagged cell. If so, it starts the game timer if it's not already running and attempts to reveal the cell. If the revealing operation fails, it checks if the game is lost and displays a confirmation dialog to replay or reset the game accordingly. If the game is won, it displays a similar confirmation dialog. The method also handles the right mouse button click to toggle the flag status of a cell. Additionally, it detects double-click events on revealed cells and prompts the player to replay or reset the game.

A screen shot of a computer program

Description automatically generated with low confidence

1. **PanelNotification**

This class represents a panel that displays game-related information and controls. It extends the ‘JPanel’ class and contains various components and functionality. The purpose of this class is to provide a graphical user interface for showing the game time, the number of remaining bombs, and a smiley face button that serves as a game reset/restart button.

**Constructor:** This constructor sets up the notification panel by creating and arranging its sub-components, initializing the time tracking functionality, and adding event handlers to the button for managing game restarts and updates. It only take 1 parameter: the game panel (game).

First, the provided ‘GPanel’ object is assigned to the game instance variable. Then the ‘lbTime’, ‘lbBomb’, and ‘bt’ instance variables are assigned with corresponding objects retrieved from the ‘CoreGame’ object of the game panel. Next, the layout of the ‘PanelNotification’ object is set to ‘BorderLayout’, and a lowered bevel border is created and set as the panel's border. Three sub-panels (p11, p12, p13) are created and added to the ‘PanelNotification’ object. Within ‘p11’, a ‘LabelNumber’ object (lbBomb) is created with an initial value of ‘000’, and the ‘updateLbBomb’ method (which will be mentioned later) is called. Within ‘p12’, another ‘LabelNumber’ object (lbTime) is created with an initial value of ‘000’. Then a ‘Timer’ object (time) is created with a 1-second interval, and an anonymous inner class implements the ‘ActionListener’ to update the time display. Within ‘p13’, a ‘ButtonSmile’ object (bt) is created. Lastly, a ‘MouseListener’ is added to the ‘bt’ object to handle mouse events, including game restart and button state changes.



**Update the timer:** This method updates the time display on the ‘PanelNotification’ object.

It first checks if the current time (nowTime) is greater than 999. If it is, it sets the time display to ‘inf’ (indicating infinity). Otherwise, it converts the current time to a string (cTime) using String.valueOf(). Next, it checks the length of the ‘cTime’ string. If it has a length of 1, it sets the time display to ‘00’ followed by ‘cTime’. If ‘cTime’ has a length of 2, it sets the time display to ‘0’ followed by ‘cTime’. If ‘cTime’ has a length greater than 2, it sets the time display to ‘cTime’ directly. Finally, the repaint() method is called on the ‘lbTime’ object to update the visual representation of the time display.

A picture containing text, screenshot, font

Description automatically generated

**Update the bomb counter:** This method updates the bomb count display on the ‘PanelNotification’ object.

It first calculates the remaining number of bombs by subtracting the number of flags from the total number of bombs, and converts it to a string using String.valueOf() and assigns it to the ‘bomb’ variable. Next, it checks the length of the ‘bomb’ string. If it has a length of 1, it sets the bomb count display to ‘00’ followed by ‘bomb’. If ‘bomb’ has a length of 2, it sets the bomb count display to ‘0’ followed by ‘bomb’. If ‘bomb’ has a length greater than 2, it sets the bomb count display to ‘bomb’ directly. Finally, the repaint() method is called on the ‘lbBomb’ object to update the visual representation of the bomb count display.

A picture containing text, screenshot, font

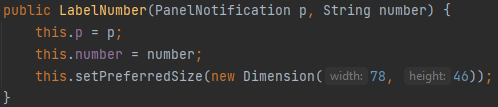
Description automatically generated

1. **LabelNumber**

This class provides a custom label component that can display numeric values as image representations, enhancing the visual appearance of the ‘PanelNotification’ panel. Its purpose is to display a numeric value as an image representation.

**Constructor:** This constructor initializes a ‘LabelNumber’ object with the given parameters: a ‘PanelNotification’ object (p) and a String value (number).

Inside the constructor, the ‘p’ variable is assigned the value of the ‘PanelNotification’ object passed as a parameter, and the ‘number’ variable is assigned the value of the String passed as a parameter. Additionally, the setPreferredSize() method is called to set the preferred size of the ‘LabelNumber’ component to a ‘Dimension’ object with a width of 78 pixels and a height of 46 pixels. This preferred size provides a specific size hint for the layout manager, which is used to determine the appropriate size for the ‘LabelNumber’ component during layout calculations.



**Rendering the label:** This method draws the appropriate images on the ‘Graphics’ object based on the value of the ‘number’ variable, resulting in the visual representation of the label.

In the method, it checks the value of the ‘number’ variable. If it equals ‘inf’, it means the label should display the image associated with infinity (which is the symbol ‘-‘). The method then uses the ‘Graphics’ object (g) to draw three images of ‘inf’ at specific coordinates and dimensions using the drawImage() method. The images are retrieved from the ‘ListImage’ object (which will be mentioned later) obtained from the ‘p’ (PanelNotification) object's ‘Game’ object, which in turn accesses the ‘GFrame’ object and its ‘LoadData’ object (which will also be mentioned later). If the number is not ‘inf’, it means the label should display a numeric value. The method retrieves the individual characters of the number string using the charAt() method and converts them to strings using valueOf(). It then uses these strings to retrieve the corresponding images from the ‘ListImage’ object of ‘LoadData’ class and draws them at different coordinates and dimensions on the ‘Graphics’ object.

A picture containing text, screenshot, font

Description automatically generated

1. **PanelPlayer**

This class represents a panel in the game interface that contains the grid of mine buttons. Its purpose is to create and manage the mine buttons for the game.

**Constructor:** This constructor is the main part of the class. It initializes the ‘PanelPlayer’ object by setting the layout, creating and adding ‘MineButton’ objects to the panel, and associating the ‘GPanel’ object as a ‘MouseListener’ for each cell. The constructor for this class only takes the game panel (game) as a parameter.

Within the constructor, the ‘game’ instance variable is assigned the reference to the ‘GPanel’ object passed as an argument. The layout of the ‘PanelPlayer’ is set to a ‘GridLayout’ with dimensions based on the width (game.getW()) and height (game.getH()) obtained from the ‘GPanel’ object. The ‘arrayButton’ instance variable is then assigned the 2D array of ‘MineButton’ objects retrieved from the getArrayButton() method of the ‘CoreGame’ instance of the ‘GPanel’ object. A lowered bevel border is set for the panel using the setBorder() method of the ‘BorderFactory’ class. Next, a nested loop is used to iterate over the ‘arrayButton’ 2D array. In each iteration, a new ‘MineButton’ object is created and added to the panel using the add() method. Additionally, the newly created ‘MineButton’ object is assigned to the corresponding position in the ‘arrayButton’ array (this.arrayButton[i][j] = new MineButton(this)). Finally, the ‘GPanel’ object is registered as a ‘MouseListener’ for each ‘MineButton’ object using the addMouseListener() method.

A picture containing text, screenshot, font

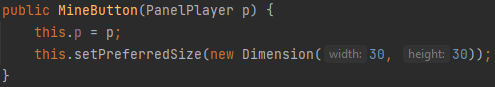
Description automatically generated\

1. **MineButton**

This class encapsulates the functionality and appearance of the mine buttons before and after being revealed in the game, with its rendering being determined by the assigned number value.

**Constructor:** This constructor is responsible for initializing a ‘MineButton’ object, which represents a cell. It takes a ‘PanelPlayer’ parameter ‘p’ which represents each cell in the game panel.

Inside the constructor, ‘this.p = p’ assigns the ‘p’ parameter to the ‘p’ member variable, allowing the ‘MineButton’ object to access the associated ‘PanelPlayer’ object. The line this.setPreferredSize(new Dimension(30, 30)) sets the preferred size of the cell to be a square with dimensions 30x30 pixels. This size ensures that the cell is visually consistent and fits well within the layout of the game interface.



**Rendering each cell:** This method overrides the paint method of the ‘JButton’ class to customize the painting behavior of the button. It is responsible for painting the appropriate image on the cell based on its number value.

Within the method, a switch statement is used based on the value of the ‘number’ variable. Depending on the value, a corresponding image is retrieved from the ‘LoadData’ object, which is obtained through a series of method calls starting from p.getGame().getGFrame().getLoadData(). The images are then drawn on the button using the ‘drawImage’ method of the ‘Graphics’ object ‘g’. Here's a breakdown of the different cases and the corresponding actions:

Case -1: Draws the image with the key "noUse", which represents unrevealed cells.

Case 0: Draws the image with the key "b0", which represents ‘0’ cells.

Case 1: Draws the image with the key "b1", which represents ‘1’ cells.

Case 2: Draws the image with the key "b2", which represents ‘2’ cells.

Case 3: Draws the image with the key "b3", which represents ‘3’ cells.

Case 4: Draws the image with the key "b4", which represents ‘4’ cells.

Case 5: Draws the image with the key "b5, which represents ‘5’ cells ".

Case 6: Draws the image with the key "b6", which represents ‘6’ cells.

Case 7: Draws the image with the key "b7", which represents ‘7’ cells.

Case 8: Draws the image with the key "b8", which represents ‘8’ cells.

Case 9: Draws the image with the key "flag", which represents ‘flagged’ cells.

Case 10: Draws the image with the key "bomb", which represents ‘bomb’ cell that is clicked on.

Case 11: Draws the image with the key "bombRed", which represents ‘bomb’ cells that are not clicked on.

Case 12: Draws the image with the key "bombX", which represents ‘bomb’ cells that are revealed when double clicked.

The images are drawn at coordinates (0, 0) with the width and height specified by getPreferredSize().width and getPreferredSize().height respectively, ensuring that the images fit within the dimensions of the button.

A picture containing text, screenshot, font

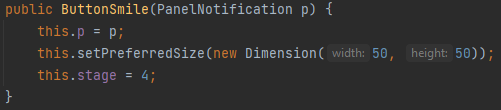
Description automatically generated

1. **ButtonSmile**

This class extends the ‘JButton’ class and represents a custom button component with different stages for displaying a smiley face. The stage determines the image that is painted on the button, and the class provides methods to retrieve and update the current stage.

**Constructor:** This constructor initializes a ‘ButtonSmile’ object by assigning the provided ‘PanelNotification’ object, setting the preferred size of the button, and initializing the stage to display a default image.

Inside the constructor, the provided ‘PanelNotification’ object ‘p’ is assigned to the instance variable ‘this.p’, establishing a reference to the panel for future use. The preferred size of the button is then set to a width and height of 50 pixels using the setPreferredSize method. This ensures that the button will be displayed with the specified size. Additionally, the ‘stage’ variable is initialized with a value of 4, indicating that the button should initially display a regular smiley face image.



**Rendering the smile button:** This method dynamically draws the appropriate image on the button based on its current stage, resulting in a visual representation that corresponds to the state of the button.

First, a switch statement is used to determine the appropriate image to be displayed based on the stage value. Each case corresponds to a specific ‘stage’ value, and for each case, the corresponding image is retrieved from the ‘LoadData’ object associated with the ‘GameFrame’ object accessed through the ‘PanelNotification’ object ‘p’. The drawImage() method is then called with the retrieved image, specifying its position and size using the getPreferredSize() method to ensure the image fits within the button. The (ImageObserver)null parameter indicates that no image observer is specified. Depending on the ‘stage’ value, a different image will be drawn on the button. The images can represent various states such as a smiling face for regular display, a winning face, a losing face, a pressed face, or a pressed play face.

A picture containing text, screenshot, software

Description automatically generated

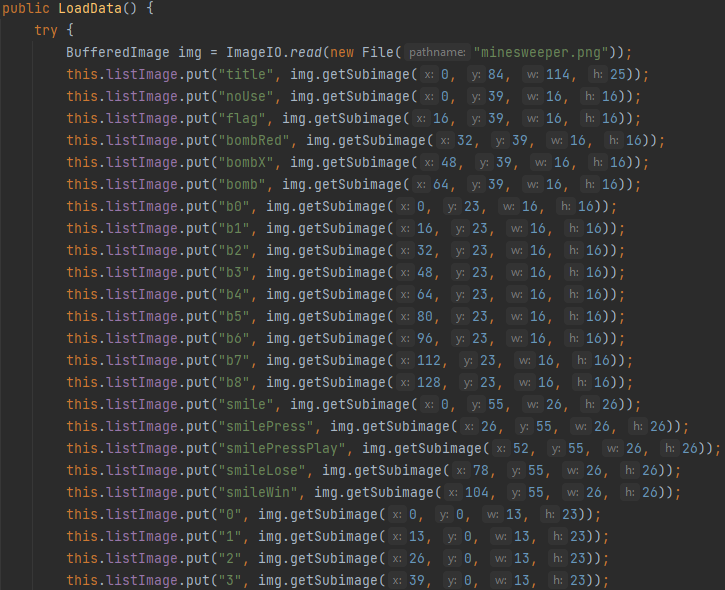
* 1. **view**

This package contains the ‘LoadData’ class, which is mainly used for adding sub-image to the ‘listImage’ object. This object is used in other classes to get the image required for displaying different components of the game.

1. **LoadData**

This class is responsible for loading and storing a collection of images used in the Minesweeper game. It uses the Java’s ‘ImageIO’ library to read the images from a file named ‘minesweeper.png’. The loaded images are then stored in a 'HashMap' named 'listImage', where each image is associated with a unique key.

The constructor in the ‘LoadData’ class reads and extracts specific ‘subimages’ from the ‘minesweeper.png’ file, representing various elements of the Minesweeper game, such as tiles, numbers, bombs, and game states. These ‘subimages’ are then stored in a HashMap called ‘listImage’ with corresponding keys. If any ‘IOException’ occurs during the image loading process, the constructor handles it by printing the stack trace.



**Chapter 3: Conclusion**

Through this project, we gained a deeper understanding of fundamental data structures and algorithms and their practical application in game development. The project allowed us to enhance our problem-solving skills and grasp important concepts such as time and space complexity analysis. Furthermore, this project reinforced the importance of code organization, modular design, and encapsulation. We divided the code into logical components, ensuring the separation of concerns and making the implementation more maintainable and extensible.

* 1. **Limitations**

While working on the project, we encountered several limitations that impacted our implementation:

Firstly, one major limitation was the lack of a comprehensive plan from the beginning. This led to challenges in structuring the codebase and resulted in some inefficiencies and code refactoring during the development process.

Secondly, another limitation we faced was the undo function's inability to undo multiple cells at once when the player clicked on a zero cell. The undo functionality was primarily designed to revert the previous move, but it did not account for the scenario where multiple cells were revealed simultaneously. As a result, players had to manually undo each cell individually, which affected the user experience and caused some inconvenience.

**Reference**

[1] Vien Huynh 2020, Minesweeper (Java), accessed in February 2023, <https://youtube.com/playlist?list=PLlD_ilI9JDbBVYP5uQUII81Q3w_pxAKhB>