COEN 275: Object-Oriented Analysis & Design
Professor Leyna Cotran
Spring 2018

Santa Clara University

School of Engineering

Final Report
A Jigsaw Puzzle

Team 2

Saurabh Somani

Shruthi Nagalapura Raghava

Shengtao Lin

Eric Castronovo

Rajas Purohit

Introduction

Games have proved to be a great idea of fun since time immemorial and it continues to be one even today. It has the ability to capture the attention of people belonging to all age groups. Also, in the current scenario game development is one of the most sought-after things. With this regard, we decided to build a game that revolves around the above objectives. An idea of a jigsaw puzzle is what made us stop looking for other ideas. It can be tricky and could take some time to get the final form no matter how well versed we are with puzzles and we exploited this very feature to make our project interesting.

Glossary of Terms

We decided to use some meaningful nomenclature for the various aspects of our game. These terms are very much relative to that of the real-world gaming examples, and therefore easy to understand.

1. Puzzle - An image that is divided into equal sized blocks and these blocks are re-positioned randomly. The number of divided blocks could be different based on the selection.
2. User - An entity that interacts with the application to play the game.
3. Login - It acts like the entry point for the entire game and provides an interface that takes in the name of the user before starting the game.
4. MainMenu - It provides an interface to the user to make a selection based on the level of complexity she wants to play. It accepts the parameters selected and redirects to a specific screen depending on them.
5. ScoreMenu - Displays the computed score for a particular user. It also provides an interface to navigate to the scoreboard and the main menu.
6. ScoreBoard - Displays the users with the top ten scores, in order to keep track of the high scores that has been received.
7. File I/O - This provides a way to update, read or write user related data into the locally stored text file.

UML Diagrams

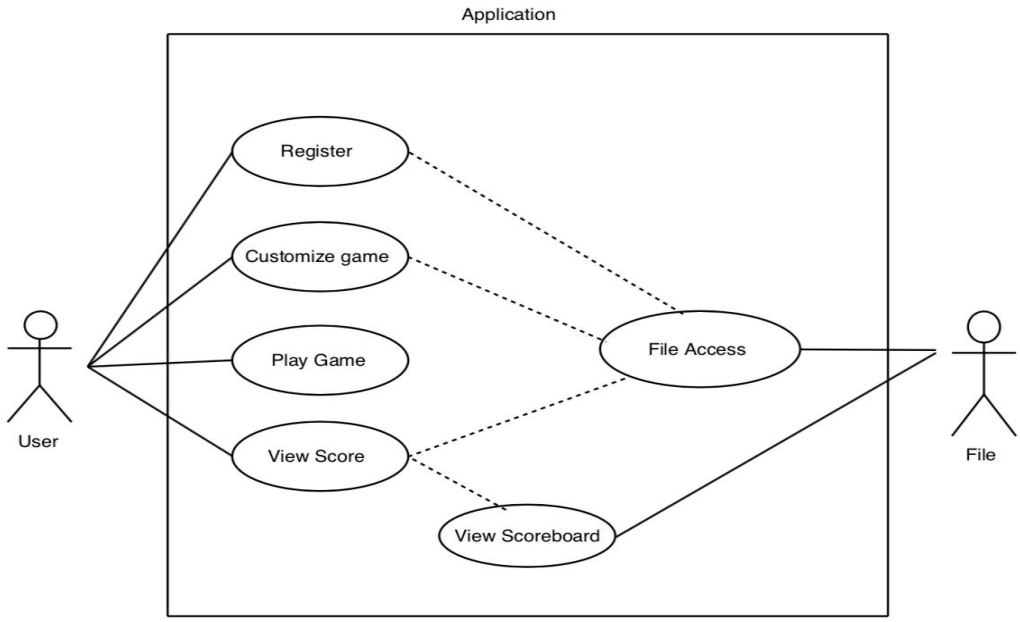

Figure 1: Use Case Diagram

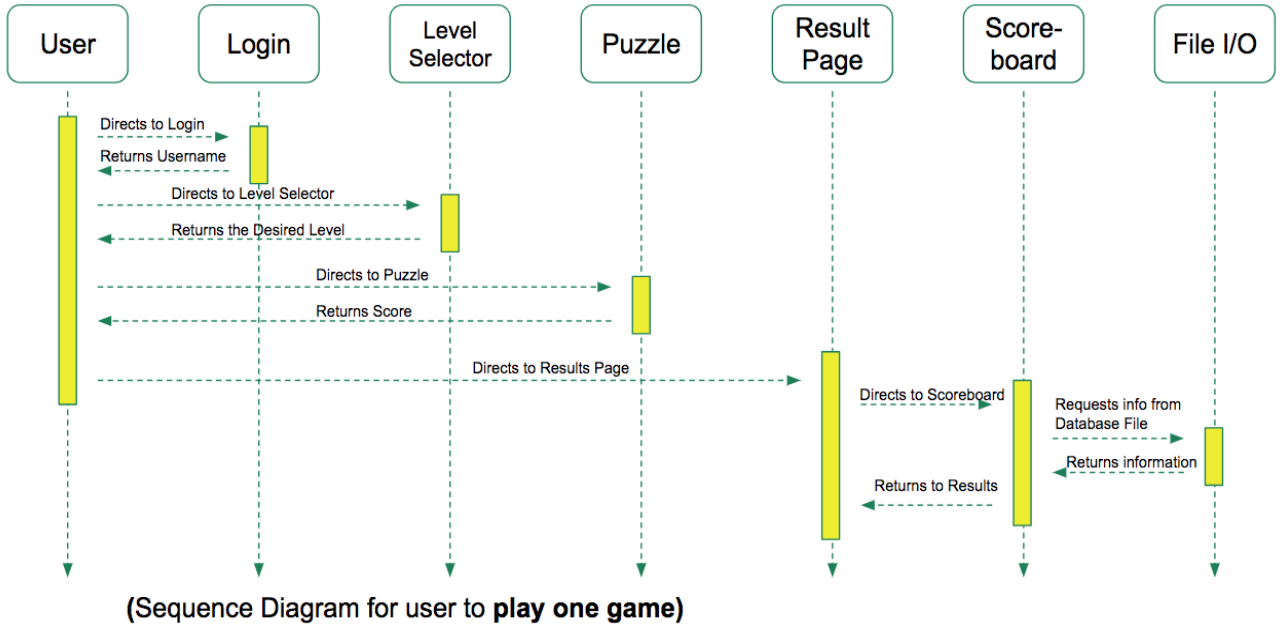

Figure 2: Sequence Diagram

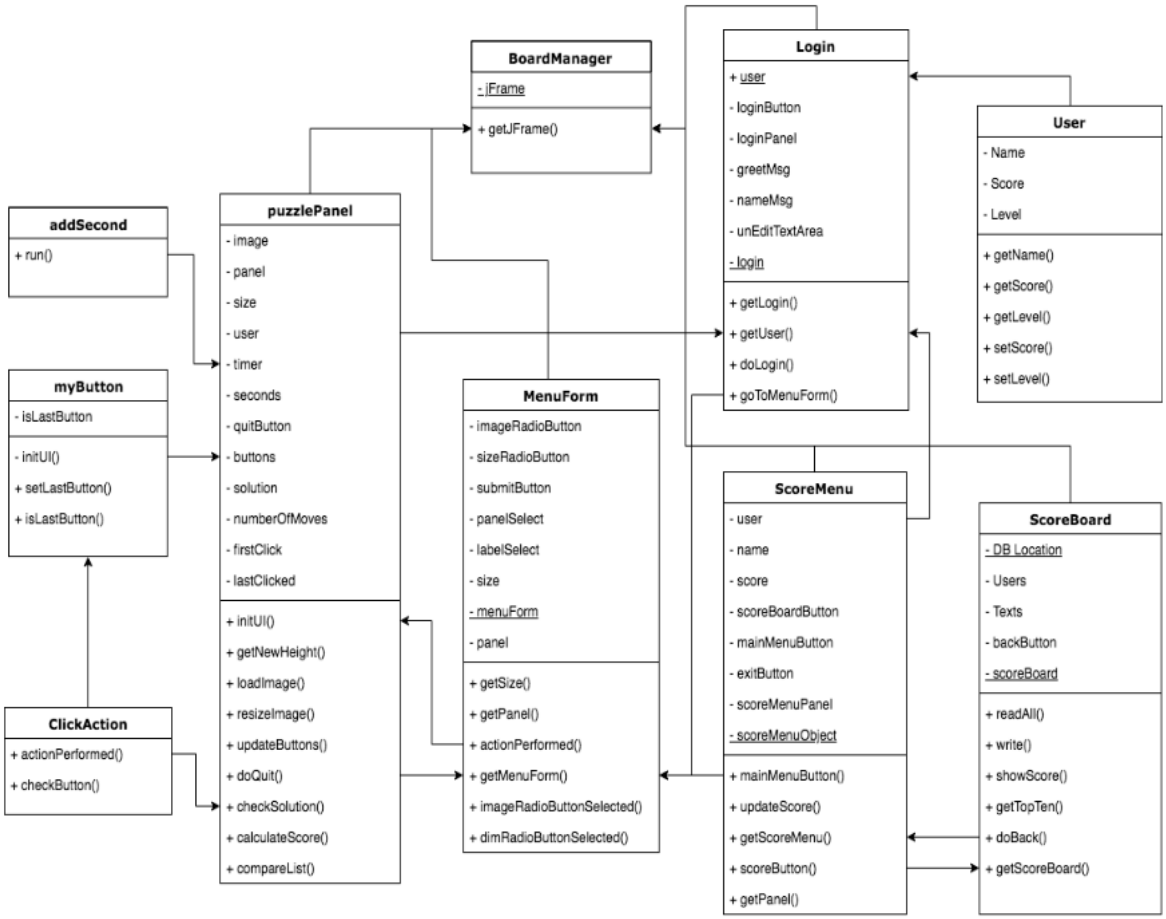

Figure 3: Class Diagram

Object Oriented Principles

Inculcating the object-oriented principles in our program made our understanding of the code and its behavior easier.

1. Encapsulation - We have achieved encapsulation in our game by making all the data members private and defining the getter() and setter() methods to access them. Also, when the data members are made private, the methods would be having the public access modifier. Some of these data members would be the GUI components such as JButton, JLabel, etc.

2. Modularity - Each class defined is targeted to perform a particular task. In the beginning the Main class calls the ScoreBoard class which creates a JFrame. Throughout the execution of the game, we only replace the JPanels that we create on top of the JFrame. Therefore, each JPanel is laid out with the specific functionality respective to that particular screen. And these functionalities have been separated out into manageable pieces, thus we have achieved Modularity in our program.

3. Hierarchy - Throughout the class design, hierarchy is very evident in the form of HAS-A relationships. The BoardManager class (i.e. that creates a JFrame) HAS-A login screen that takes in the user name, HAS-A Menu Form for the selection of a customized puzzle, also HAS-A Puzzle Panel for displaying the puzzle to solve, HAS-A ScoreBoard where the user can get the statistics about the top scorers for the game, and also HAS-A ScoreMenu that would display the score obtained by the user.

4. Abstraction - When the user solves the puzzle, at the background the timer starts running the moment the puzzle screen is rendered. And when the user completes the game successfully, she would be provided with the details such as time taken, and the score obtained. Although, a lot of processing goes on behind the scenes, the user would only see and interact with the simple UI of the game by hiding all the ugly complexities below it.

Booch's Principles for handling Software Complexities

In order to tackle the software related complexities encountered during the development of a system, we made use of the below concepts for a better flexibility and representation of our game.

1. Stable Intermediate Forms - We have developed the entire system in such a way that each class or each functionality could be implemented individually without affecting the overall system. If we re-used one of the component from our code, for instance the login feature, the score calculation feature, the menu form for selecting a particular game, etc. it would still work in other systems without any problem. Therefore, a component by itself functions independently, and also the system after integration of many such standalone components.

2. Hierarchical Structure - The hierarchy between the board manager and the other classes has already been explained above in the Object-Oriented principles section. In addition to this, the puzzle panel would have the action listeners implemented in order to capture the event and take a necessary action towards it. Each shuffled piece in the panel would be modified as buttons, on click of which an action would be triggered.

3. Separation of Concerns - One of the main necessities for a complex system is to separate out the overall functionality into manageable pieces, so that they are destined to perform a specific task. Here, in our game we have separated out the functionality in each of the panels. Once the login is performed, a user class would be created to record the various attributes and these values are altered throughout the execution of the system. The instance of this class keeps track of the user related details such as the score obtained, etc.

4. Finding Common Patterns - As we discussed in the previous section, the JFrame created for the entire system would be just one, and the same window is used for all the screens by only replacing the various JPanels. Some of the GUI components can be reused across the various screens.

5 C's applicable for Class design

The 5 C's that are defined for designing the class are applicable here.

1. Cohesion
2. Clarity
3. Completeness
4. Convenience
5. Consistency

Cohesion refers to the concept of Abstraction in the implementation. This means that there would be coupling in the classes in terms of making a functionality visible to the user. When we try to access an instance of one class in another, the program does so in a seamless manner without the user having to worry about the detailed implementation.

Each and every class that has been defined has specific functions to perform that are clearly defined. Hence, we see Clarity in the class definition.

Completeness refers to the ability of the classes to be complete by covering all the necessary data members and member functions. Thus, making the class definition complete.

Convenience has been achieved by ensuring there is modularity preserved in the code. Also, the naming conventions should be relevant to the scenario so that the code is readable. We have taken care of this aspect right from the inception of our project.

One of the most important aspects that forms a part of class design would be to check whether the class defined is consistent with that of the requirements analyzed for the system as a whole, and we handled it by mapping the functional requirements defined across the code implementation.

Design Patterns used and How we avoided Anti-Patterns

We have made use of the Strategy Design Pattern and the Singleton Design Pattern.
Some of the aspects like the size of the Board and the image and the number of buttons the puzzle is divided into, is decided during the runtime. Depending on the selection of these parameters, the object instantiation takes place. Therefore, application of the Strategy design pattern is most suited for this scenario.

In addition to this, we can also apply the Singleton Design pattern. In our game application, the BoardManager class creates the JFrame and it calls multiple JPanels throughout the execution of the code depending on the functionality. Therefore, there is just one instance of the BoardManager class.

As soon as the user enters the name in the login screen, this information instantiates an object of the user class, and this is done only once and updated throughout the execution of the game. The user class is updated with the score obtained once the user completes the game successfully. Thus, only a single instance of the user class is maintained throughout the system.

Anti-Pattern - We have avoided the God Object in our system. Initially, we had just one Score class that had the functionality of the ScoreBoard (To display the top ten score holders) and also the ScoreMenu (Displays the score obtained by the user along with the time taken). As when moved forward with implementation, the functionality of this class expanded and hence we had to separate this into two different classes, namely ScoreBoard and the ScoreMenu. Thus, we ensured the “Separation of Concerns” in our system.

Multi-threading Functionality

We have made use of the concept of Multi-threading in our program by implementing the Timer API. The timer in our program keeps track of the time taken by the user to complete the puzzle game, also simultaneously it also records the number of moves that were involved.

This feature can be treated as multithreading since we have the user interacting with the puzzle by clicking pieces of puzzle which are in the form of buttons, and at the background we have the timer running for the purpose of score calculation.

These two operations are in sync with one another and both the tasks are making progress at the same time.

Law of Demeter

All the classes we have defined exhibit the law of demeter. We have enabled interaction between any two classes only when required. For instance, we have the Login class interact with the User class to create an instance by passing on the user name to it. Similarly, we see an interaction between the MenuForm and the PuzzlePanel class where the parameters from the form, selected by the user is passed on to render the relevant puzzle at runtime. Also, once the puzzle is completed successfully the user gets her score along with the time taken, this is possible with the interaction between the PuzzlePanel and the ScoreMenu class.

Requirements

Functional Requirements

ID	Title	Description	Dependency	Priority
FR-001	Register username	The user should be able to register a name for the game session		High
FR-002	Display Game Menu	A game menu showing the difficulty levels along with the image to be played with, should be displayed	FR-001, FR-004	High
FR-003	Display Puzzle Board	The image selected should be shown, distorted into the difficulty level chosen by the user	FR-001, FR-002, FR-004	High
FR-004	Respond to user input	The game should accept and react to the user input		High
FR-005	Changing Difficulty	The difficulty level should change based on user input	FR-002, FR-003, FR-004	Medium
FR-006	Timer	A timer for the game session should be maintained, for further use while calculating user score	FR-001, FR-003, FR-004	High
FR-007	Moves Tracker	The number of moves user takes to complete the image	FR-003, FR-004	High

FR-008	Game Status	Display status after game completion	FR-004, FR-006, FR-007	Low
FR-009	Show scoreboard	Displays the top 10 scores of the players	FR-008	High
FR-010	Main Menu	Redirect user to the main menu	FR-003, FR-004	Medium

Non-Functional Requirements

ID	Title	Description	Priority
NR-001	Local data storage for fast access	During login, the name is stored in the file and subsequently the other related information such as the score, time taken would be saved in a local file for faster access. This information would be retrieved onto scoreboard quickly if the user is one among the top ten score holders.	High
NR-002	Intuitive UI	Starting from the welcome screen upto the exit screen, all of them have been designed keeping in mind the ease of use for the users and also made intuitive so that interaction becomes seamless.	Medium
NR-003	Performance	This JIGSAW puzzle game takes minimal time in processing the user requests and data delivery. This is very evident when the user makes a request to view the scoreboard.	High

Traceability

We took several measures to ensure that our game application was consistent with the requirements that we decided on before the implementation. For this purpose, we came up with the UML diagrams like Class diagram, Sequence diagram, Use case diagram. With the help of these models we could understand the data flow from one entity to another, and what states changes when a particular behavior is applied onto an entity, what are the states that needs to be maintained for the system to be preserved, etc. Once we designed the various screens of our game and how the data flows from one screen to another we were able to come up with an initial crude version of our game.

As and when each of us went on developing different parts of this game application, we realized that "Separation of Concerns" was really helpful and also let us define the various modules of the system separately without dumping a great deal of functionality under one entity. When we designed our classes, we made sure the 5 C's for class design were applied so that our system would be reliable and maintainable.

Functional and the non-functional requirements were defined and each of the requirement was assigned a priority, this enabled our team to work productively on the end goal we had set for ourselves. We made sure that these requirements were communicated clearly to all the team members through various brainstorming sessions. With all of the above logistic support we were able to plan our work and execute it efficiently.

Testing Approaches

As mentioned in the previous sections, our game application consists of several components that have been developed, unit tested and integrated into the entire system. It is possible to only take a particular component in order to re-use it in another system and it would work fine after doing so. Our code is very readable and hence easy to understand with clarity, consistency and convenience taken care of.

The different components of our system would be, the user interface (i.e. GUI), the locally stored file with user information, the puzzle logic, the scoreboard and the use of API's whenever necessary. We have individually tested each of these in order to ensure there are no prevalent errors or bugs in the code, as and when we tested these individual components which were error free, we went on integrating it into the system one after the other and each time we recorded the cumulative results. This approach helped us troubleshoot the problem in a convenient manner.