Team: Camilla Lambrocco, Edwin Coy III, William Dell'Anno

Title: Strategy Game Arcade

Project Summary: Our program will have several strategy style arcade games including chess, checkers, tic-tac-toe, and connect four. One and two player gameplay will be possible and an AI built to defend against a loss (vice work for a win) will be implemented. A GUI will be built which the user can interact with using a mouse.

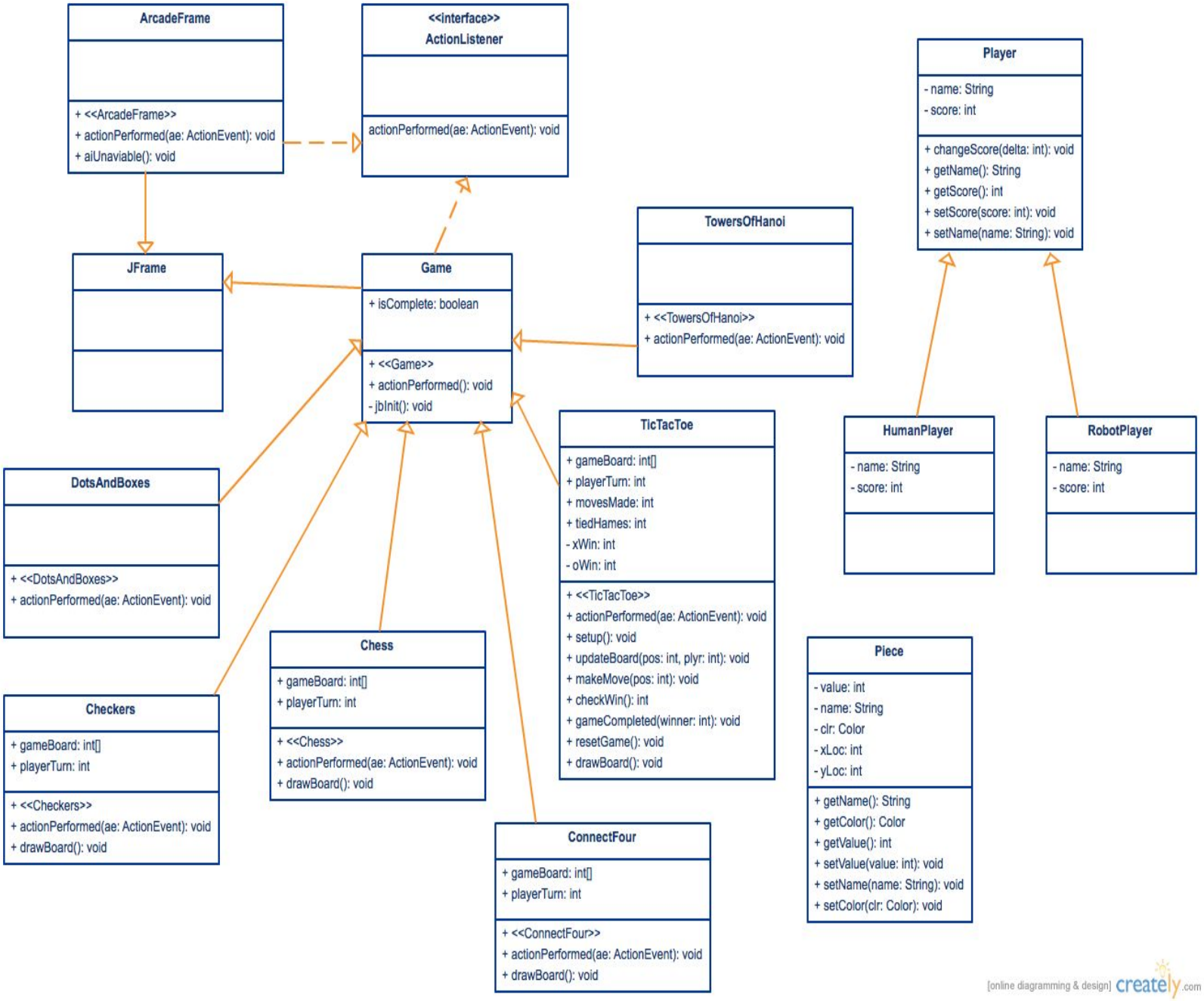
Implemented Features: The following requirements tables are taken from the Project Part 2 document. Requirements highlighted in GREEN were successfully implemented in our software. Requirements highlighted in YELLOW were partially implemented in our software. Requirements highlighted in RED were not implemented in our software.

User Requirements		
ID	**Description**	**Priority**
UR-01	User can select a game to play.	Critical
UR-02	User can use a mouse to make selections.	Critical
UR-03	User can view the score.	High
UR-04	User can select a game mode.	High
UR-05	User can access the arcade menu from any game.	High
UR-06	User can quit a game.	High
UR-07	User can select a one or two player mode.	High
UR-08	User can reset the score for a set of games.	Medium
UR-09	User can restart a game.	Medium
UR-10	User can view high scores.	Low
UR-11	User can enter their name.	Low
UR-12	User can select the difficult level of the AI.	Low

Functional Requirements		
ID	**Description**	**Priority**
FR-01	Application AI will compete against single player.	High
FR-02	AI will select game moves defensively.	High
FR-03	Game pieces are distinguishable between players.	High
FR-04	Game will alternate between player turns.	High
FR-05	AI will select game moves offensively.	Medium
FR-06	Game moves and actions will trigger audio effects.	Low
FR-07	High difficulty AI will make offensive and defensive moves.	Low
FR-08	Low difficulty AI will make defensive moves only.	Low

Non-Functional Requirements		
ID	**Description**	**Priority**
NFR-01	Invalid player selections will not crash the application.	Critical
NFR-02	Games have real time response.	High

Final Class Diagram:



Benefits of Designing Before Coding:

We found that the process of thinking through code design ahead of time lead to cleaner code solutions. Spending the extra time in the beginning allowed us to catch structural issues before writing a single line of code. When it came to implementing the program that we had designed on paper, we found it rather satisfying to see our well thought out structures fall into place as planned. For the most part this was true, however, in a few scenarios there were things that popped up that we were unable to predict during our design phase. These were not big issues and we were able to get past these without too much hassle or change in our design.

The benefits of designing before coding became clear when we started developing the software. By designing a good class diagram we were able to successfully implement a working software that accomplished the goal originally set. While designing the diagrams we realized, indeed, that we could have made use of the concept of encapsulation to create a cleaner code. We can see that our program has the classes `RobotPlayer`, `HumanPlayer` and `Piece` which perform a specific task for the game. We created instances of those classes in the various games implemented, resulting in benefits like data binding and protection from the outside interference. Moreover, thanks to the designing discussed, the concrete classes developed for the games of the arcade were easily implemented because of the reusable code created. Finally, we understood that a good design of the class diagram is necessary to be able to build a dynamic and efficient software.

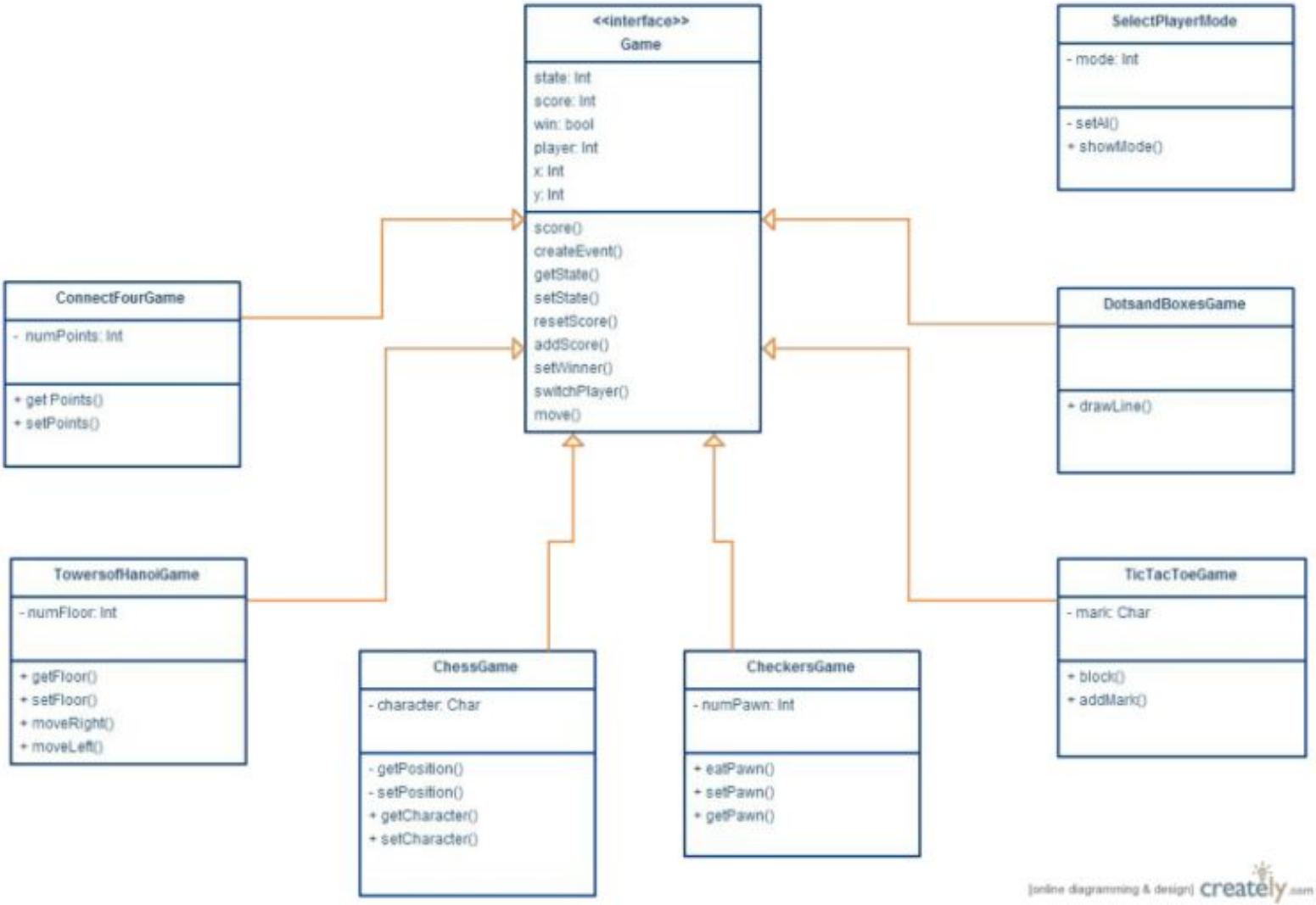
Design Patterns:

Our prototype software did not specifically make use of any design patterns. Upon further examination and investigation, we believe that our software lends itself to the Abstract Factory design pattern. The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes.

The `Game` class is the Abstract Factory with the individual games (`TicTacToe`, `Chess`, `Checkers`, etc.) being the Concrete Factories. The `ArcadeFrame` class is the application client. The `Player` and `Piece` classes are Abstract Products, whereas the `HumanPlayer` and `RobotPlayer` classes are Concrete Products.

Rather than trying to design to a design pattern specifically, we could improve our software by utilizing some of the benefits of the Abstract Factory design pattern. For example, all games require two players, option buttons (Menu, Quit, etc.), player score displays, and a gameboard. The `Game` class can contain all of these items so that they are available to the Concrete Factories for each game to make use of as applicable. Additionally, the `Player` class could either be used for a `HumanPlayer` or a `RobotPlayer`. Each is a player in one or more games, however each is controlled differently and may make use of different properties of a player (e.g., name, difficulty, etc.).

Initial Class Diagram:



Comparison of Initial and Final Class Diagrams:

We changed the class diagram to better implement the various concrete classes of the games in the arcade and to give a more complete idea of the final game. We added a class `Player` which features are inherited by the concrete classes `HumanPlayer` and `RobotPlayer`. This allows us to satisfy the user requirement UR-08 (“User can select a one or two player mode”) and functional requirements FR01 and FR04 (respectively, “Application AI will compete against single player” and “Game will alternate between players”).

We added the class `Piece` that gives us the ability of instantiate the Piece object for any different game of the arcade, fulfilling the functional requirement FR-03 (“Game pieces are distinguishable between players”). This feature lowers the amount of coding labor and give to the code flexibility.

The `Game` class was changed from being an Interface to be a concrete class that implements the Interface ActionListener and extends the class JFrame. The former was implemented to improve the event calling when a move is performed. By implementing ActionListener, indeed, any class that is interested in processing an action can do it by invoking the method `actionPerformed()`. The latter was extended to create the layout of the game: the frame. This allows us to constructs a new frame that is then inherited by the different games of the arcade. We notice that by redesigning our class `Game` we can add any new game (subclass) to the arcade without changing its superclass (`Game`, indeed).

Finally, we updated the game `TicTacToe` to provide a fully functional demo of at least one of the arcade games. `TicTacToe` uses all the new features described above and fulfills the high priority requirements established in our project requirements sheet (see project requirement part in StrategyGameArcade_Part2).

By better specifying the methods and the attributes of `TicTacToe` we were able to better implement data hiding and encapsulation. As an example the score of the player is stored in the `Player` class. This allows to have controlled access to “score” and to wrap the information so that it can be hidden by binding it to the respective name of the player.

As a final comment, thanks to the features implemented (i.e. new classes `Piece`, `Player` and modification of class `Game` in addition to the private attributes) we are now able to control better how data is accessed, the code is more readable and maintainable and we can reuse it for future purposes like adding a new game in the arcade.

Reflection on Analysis and Design:

Designing a system from customer requirements may lead to an incomplete system if the requirements are incomplete. Additionally, if the requirements are too specific, the developer may not be able to implement a software design as effectively as he or she otherwise would. Analysis and design requires a fine balance between determining what the customer's needs are and balancing them with a software solution that could make use of existing design patterns for improved efficiency.

In the process of designing, implementing, and delivering our software product, there were a number of design changes that needed to be made to accommodate required features. More comprehensive planning and a more complete vision at the beginning phases of the project likely would have yielded a more cohesive design and more efficient implementation and delivery phase.

Link to Software Demo Video

Link to Full Presentation Video