YAHTZEE

SOFTWARE DESIGN DOCUMENT

Team 17
Khang Phan
Duc Phan
David Recic
Eric Le

INTRODUCTION

PURPOSE

In this software design document, we will be chronicling the design we intend to implement for the Yahtzee Game we are developing for Procrastination Pastimes. Detailed in this document is the System Architecture and the subsystems needed for this project's functionality. The overall goal of these subsystems that we have created is to correctly model the game Yahtzee in a clear and concise way such that any technical debt can be minimized.

DESIGN GOALS

The design goals of this project are in part laid out by the client and then added onto by the developers. The client has specified their functional requirements, so the design goals should reflect the needs of those requirements. In order to do so in such a way as to provide maintenance readily and easily. These include basic design decisions that are common in Object-Oriented Programming such as encapsulation, inheritance, and abstraction. These concepts allow for greater understanding and utility while also providing structure to the design of the system. In addition to these core concepts, the readability and understanding of the code from the outside is crucial to the maintainability of the project going forward. Thus, the developers have also decided to use standard naming conventions such as: classes being named with singular nouns, methods named using verbs and nouns that the describe, errors thrown during runtime should provide context to the user such that they can find out the issue. With these design goals in mind, the end product will have code that is: clean, readable, understandable, and maintainable.

DESIGN TRADE-OFFS

When designing the system, there were several decisions that had to be made that would affect the end product. One of those was the design of the User Interface subsystem. As of now the project is slated to be programmed in C# using Unity to provide a framework. While the the developers could agree on the necessity of a package to handle the UI, their lack of working knowledge of Unity did not allow for them to correctly create classes to properly model the UI. Another decision that was made was whether or not to include the option for

multiplayer over the internet. However, since Yahtzee is a turn based game, the need for internet connectivity is unnecessary as they could simply take turns at the keyboard.

GUIDELINES AND CONVENTIONS (1)

- Use of encapsulation
- Use of abstraction
- Use of inheritance
- Classes will be named in singular nouns with first letter uppercase
- Errors will be put through an exception
- Method will be named with verb phrases
    - Verb will be lower case
    - Nouns will be upper case
- Code convention:
    - C# code convention (see reference)
    - Tab: 4, replace tab with spaces
    - 120 characters per line.

DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

- RAD: Requirements Analysis Document
- SDD: Software Design Document

REFERENCES

- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions#layout-conventions

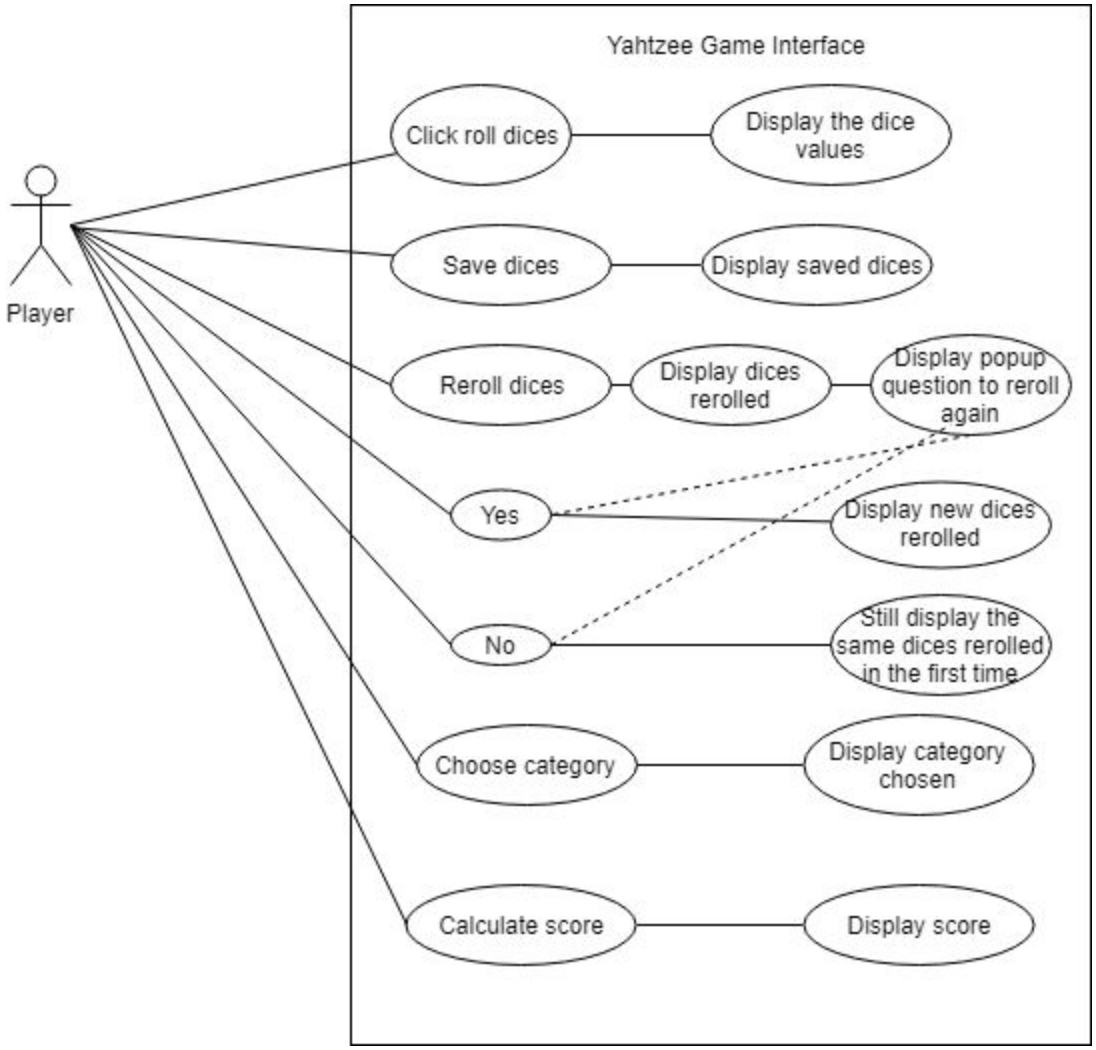
OVERVIEW

The project was creating the software architecture. We based on the client's wishes and see what trades off had to be made in order to create the current version of the software architecture. Guidelines and conventions have been created to create consistency in programming so that all developers and clients understand what functions of different parts of the system.

CURRENT SOFTWARE ARCHITECTURE

OVERVIEW

The architecture of this game will be built in the following manner. There will be five different subsystem. These subsystems are the Drive subsystem, the Interface subsystem, the Player subsystem, the Score subsystem, and the Settings subsystem.



Subsystem decomposition

As mentioned in the overview, the software architecture is composed of the Drive subsystem, the Interface subsystem, the Player subsystem, the Score subsystem, and the Settings subsystem. The Drive subsystem will be an object with the functionality of staring, loading, and exiting a game. The Interface subsystem is where the six different graphical interfaces will be created. These six graphical interfaces are the start screen where players will have the option of starting, loading, or exiting the game. The next screen after pressing the start button will be the character selection screen. After characters are created the next screen will the play screen. In the play screen there will also be the score screen, which will pop up when the user press the expand button. Then there are two screen that will be an option on every other screen, the Menu and the Settings. The Menu will gain a function to go the Settings screen and the ability to do feedback on the game, look at the policy agreement of the game and also functionalities such as saving, restarting and existing the game. The Player subsystem will be comprised of two classes the Player class and the Dice class. The Score subsystem will be comprised of the scoreSheet class and the scoreCategory class. The Settings subsystem will be comprised of the audio and display class. All of the functionalities of these classes can be referred to in class interface section of the SDD.

There will be the interface portion. In three of the interfaces, the setting class will be available. The start . The game interface will have access to the player class and scoreSheet class.

Hardware/software mapping

No specific requirement for hardware and software mapping.

Persistent data management

The only type of data management that will occur in this architecture is in the save feature. Users will have the option of saving a game and then be able to load that game in a different session. This will be supported by putting the information from each player in a game session into a text file. The text file will contain information on the character names and the scores when a player saved a game. There will also be data persistence for languages in order to select a preferred language.

Access control and security

This will be non-applicable in our system architecture.

Global software control

This will be non-applicable in our system architecture.

Boundary conditions

When the users turn on the game, a game object will be created that will allow for the Design subsystem functionality. If the user want to shutdown the game, the user will press the exit button, which will then run the terminate function of the game object. There will be several error handlers. One will be for the load function if the format of the file does not match the necessary requirement or they contents of the file are empty. Another error if the user doesn't create a character and try to start a game.

Proposed Software Architecture

Removed the Move class in Current Software Architecture.

Subsystem Services

The following part describes the services of the above subsystems in term of operation.

UserInterface:

- update the canvas to display one of four main menus
- exit the program
- change values for language, music based on user's selection

Controller:

- Function to load game.

- Function to save game.
- Function to start game.

Game:

- Function to throw dice.
- Calculate the score
- Function to save dice.
- Function to score and update Scoresheet.
- Function to end game when a player wins.

Move: A move of a particular player. This class's functions help calculate score of a player based on the dice rolled by that player and the category the player chooses.

- Update player move.
- Provide the player's dice value.
- Update the score categories that are compatible to the dice values.

ScoreCategory: Provides score categories with corresponding value of each one.

- Update types of categories.

Setting:

- Modify volume level.
- Change song.
- Change language.

Player subsystem: containing two class: Player and Scoresheet. Its main function is to store information of player, including name and scores.

- Update player's information (language)
- Update player's score sheet.

PACKAGES

This section describe each package in the implementation and its function.

Interface

This package include files required for the interfaces. The interface will have at least three files for three screen: first screen for different creating game options (load from old file or create new game), one screen for adding players, and one screen for the game.

Player

This package will have three file corresponding to two class in the Player subsystem: Player, and Scoresheet.

Main

This package will contain the Game and Controller class.

Score

This package will contain ScoreCategory class and different implementations on how to obtain a score.

Setting

This package will contain Setting class.

Class Interfaces

UserInterface:

- changeView(value: int) -> update the canvas to display one of four main menus
- gameExit() -> exit the program
- updatePreference() -> change values for language, music based on user's selection

Controller:

- loadGame(fileName: string) -> void, throw LoadingException
- saveGame(fileName: string) -> void, throw SavingException
- startGame()

Game:

- throwDice(player: Player) -> [5 dices, scoreSheet with supposed score]
- saveDice(player: Player, dices: Set of Dice) -> void
- score(player: Player, category: ScoreCategory) -> new Updated scoreSheet
- endGame(player: Player) -> winner player (if applicable)

Scoresheet: following functions provide information about the current score of the player.

- getLowerScore():
- getUpperScore():
- getUpperBonus():
- getTotalScore():
- getNumberBonusYahtzee():
- getDetailScore()
- saveScore(): save score
- checkYahtzeeFilled():
- isYahtzeeFilled(): true/false whether yahtzee is filled or not.

Player:

- saveScore(category: ScoreCategory, score: int) -> new Updated scoresheet.
- isYahtzeeFilled() -> true/false whether yahtzee is filled or not.

- setLanguage(): change the language of current player

Move:

- getPlayer();
- getDiceValue();
- getScoreCategory();
- calculateScore();

ScoreCategory: Provides score categories with corresponding value of each one.

- getName();
- getValue();

Setting:

- setVolume(volume: integer) -> void
- setSong(songName: string) -> void
- setLanguage(language: Language) -> void

Detailed Design

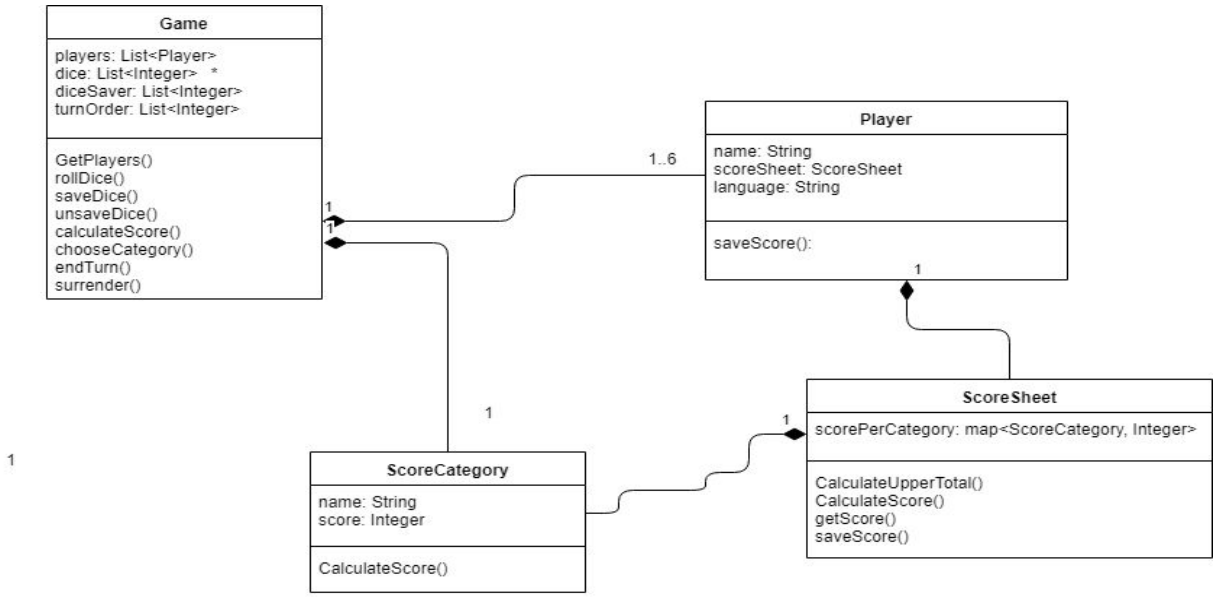


Figure 1: Design-level Class Diagram

We remove the Move class from the previous version because we no longer need to save the actions from previous player's turn to undo moves. We remove the Setting class and have it inside Unity. We also implement more functions in Game class and modify the ScoreSheet functions.

GLOSSARY