Software Requirements Specification

Table of Contents

REVISION HISTORY

Revision Number	Revision Date	Nature of Revision	Approved By
1	07/06/2020	Table of Contents Amended to include Revision History	Derek

1. Glossary

API: Shorthand abbreviation of "Application programming interface" describing the public methods of a class that are externally accessible

Category: These are classifications for the types of questions that a user might be asked. Some examples include people, places, events, and dates.

Center Piece: The center piece is the final spot on the board. A user must first collect all wedges before they can move to the inside of the board. When this has happened, they must roll the exact number to land on the center piece. If the user answers a question correctly while on this space, they win the game.

Dream Team: Refers to the software development company started by JohnsHopkins University students Derek Randall, Stefan Doucette, Mauro Chavez, andJeff Pauls in June 2020.

GUI: A type of user interface through which users are able to interact with a system via visual indicator representations. Stands for graphical user interface.

Hasbro: Refers to Hasbro, Inc., an American company that develops toys, board games, and media. Hasbro owns the trademarks and products of, among others, Kenner, Parker Brothers, and Milton Bradley.

Question Bank: Set of flat files within a directory internal to the software where all questions, answers, and categories are stored as text in a well defined and consistent format.

Signal: The press of a button or the calling of an API that triggers an event within another subsystem

System: This refers to the particular entity that is performing the functions defined in use cases. The system in question cannot be an actor in its own use cases, but it can be an actor that triggers use cases in another system.

Terminal: What allows a user to accomplish and automate tasks on a computer without the use of a GUI. Aka command line or consoles.

Tkinter: Package that must be installed in Python3 in order to build GUIs.

Trivial Pursuit: Refers to the trivia-based board game sold by Hasbro. The idea for this product is based on a similar experience for users, but targeted towards the education sector.

US nighttime hours: This represents the time frame from 6:00 pm EST to 6:00 am EST.

User: A user is defined as a person who plays the game. Our main focus will be towards users that are students, however the game will be available to be enjoyed by non-students as well.

Vision Document: Previous planning document developed by the Dream Team.

Wedge: A wedge is a piece that fits in the mover and represents a category that a user has successfully answered a question in.

2. Software Architecture (Subsystems)

2.1. Gameplay Subsystem

Game Board

- **Encapsulated Information:** The encapsulated information in the Game Board subsystem is primarily other subsystems, including the Game Positions, Players, Card Deck, and Dice.
- **Inputs:** Terminal or GUI inputs come from the user, including the answer to the question asked, the desired move direction, dice roll trigger, and question response validation.
- **Outputs:** Terminal or GUI outputs are given to the user, including questions to be answered. Effects of player input on game state will be captured and reflected in Terminal or GUI outputs.
- **Intended Functionality:** The Game Board is the primary manager of the game during game play. Functionality includes the ability to setup/initialize the board, move a player, execute a position, and give a wedge. The initialization loads stored information to determine the geometry of the board and create Game Positions in addition to initializing the Card Deck which is populated from loading content from question bank subsystem. Moving a player involves a call to the Dice to determine the amount of positions to move. Executing a position is either asking the user a question based on the shown category or moving again if the Game Position requires it. A wedge is given to a player if the user answers the question correctly and they do not already own a wedge of that category.

- **Business Rules:** The Game Board moves a player if the associated user has answered the question correctly. The Game Board finishes a game when a player has answered the question correctly on the center piece. It also facilitates the movement back into the center of the board once a player has collected all the required wedges. The Game Board handles the interface when a player lands on a category Game Position and is asked to answer a question.
- **Constraints:** None
- **Interfaces:** Terminal and GUI outputs are graphically displayed strings and images. Terminal inputs are strings and GUI inputs are strings or subsystem calls.
- **Interactions:** The Game Board interacts with Card Deck, Dice, and Player subsystems.

Game Position

- **Encapsulated Information:** The information within a Game Position is its value , location index, next location index, and position type. The value of the Game Position is a category type or "roll again," depending on the static game board. "Location index" refers to the linear index as part of a linked list with values between 1 and N, where N is the total number of Game Positions. The "next index" describes where a mover would travel to next, which is most often the current index incremented by one, however, transitions from position type to another. The "Position Type" is "outside," "spoke," or "center," which are different paths on the Game Board.
- **Inputs:** Game Position layout during initialization, correctness of answer
- **Outputs:** Correct or incorrect response to the question asked
- **Intended Functionality:** Store Game Position encapsulated information and allow calling subsystem to ask a question pertaining to the Game Position. The user is presented a question and queried if they answered it correctly.
- **Business Rules:** When called from another subsystem, the Game Position must ask player question retrieved from Card Deck, give the user the answer once he/she has had a chance to guess, and user input provides "correct" or "incorrect" to allow the software to react accordingly, return Boolean
- **Constraints:** none
- **Interfaces:** Correctness of answer is a string provided by the user, either via the terminal or the GUI.

- **Interactions:** The Game Position interacts with the Game Board, which is the calling function, and the Card Deck.

Card Deck

- **Encapsulated Information:** Contains the number of cards in the deck, and interacts with the individual cards. Maintains the category of the cards and tracks which ones have been seen and not seen in the current match.
- **Inputs:** card category
- **Outputs:** questions corresponding to a card of the given category
- **Intended Functionality:** The card deck contains the entire set of cards. The calling subsystem will interact with the card deck, which abstracts the details of the individual cards and their questions.
- **Business Rules:** The calling subsystem initializes the card deck at the beginning of the game, loading question content from a flat file question bank. When a question is needed, the calling subsystem retrieves a card of a given category, marks that card as played, and provides the card's question.
- **Constraints:** none
- **Interfaces:** The input and output are both strings
- **Interactions:** The Card Deck interacts with the cards, which it "owns," and the game board, which is the calling subsystem.

Card

- **Encapsulated Information:** Contains one question related to the category of the color that has been landed on, and is immediately owned by the Card Deck subsystem. A Card also stores if it has been played to eliminate redundant questions.
- **Inputs:** Color/category of card, e.g. "red"
- **Outputs:** Returns one question
- **Intended Functionality:** Retrieve question from database and return
- **Business Rules:** Given a color (based on the category), the card retrieves a random question from the question database and returns the resulting string to the calling subsystem.
- **Constraints:** none
- **Interfaces:** Input and output are both strings
- **Interactions:** The card subsystem interacts with the card deck and question database

Player

- **Encapsulated Information:** The player subsystem contains the static information associated with a single player, which includes the player's name and picture (if applicable). Dynamic information, like the player's current mover position, earned wedges, and is contained in other subsystems. A mover is "owned" by every Player.
- **Inputs:** Player name, image, and start location
- **Outputs:** none
- **Intended Functionality:** The Player subsystem is instantiated for each player (or team) once a name, image, and start location is provided.
- **Business Rules:** A Player subsystem interacts with a user to gather the information that will be encapsulated. Once created, the subsystem is used to store the user's state as he or she advances through the game, and is used as an interface between the Game Board and the Mover.
- **Constraints:** None
- **Interfaces:** The Player subsystem's name is a string and the image (optional) is a .png.
- **Interactions:** The Mover and Game Board both interact with the Player subsystem.

Mover

- **Encapsulated Information:** The Mover subsystem contains the number and type of wedges that are currently owned by the player and current index location of the Mover.
- **Inputs:** Delta index when moving, wedge color when adding a wedge
- **Outputs:** None
- **Intended Functionality:** The Mover traverses the Game Position data structure depending on the input delta index provided by the Dice subsystem. A calling subsystem can add a wedge to the Mover when a player answers a question correctly in a new category.
- **Business Rules:** Given a delta index (integer), the Mover updates the new Game Position in its internal state, on which it will reside after a move. Separately, the earned wedges stored in the internal state will be updated to include a new given wedge color by the Game Board subsystem.
- **Constraints:** None
- **Interfaces:** The delta index input is an integer and the input wedge color is a string.
- **Interactions:** The Mover interacts with the Game Board during movement.

Dice

- **Encapsulated Information:** None

- **Inputs:** None
- **Outputs:** Face value of rolled dice
- **Intended Functionality:** The calling subsystem can use the Dice subsystem to retrieve a value to move the Mover subsystem.
- **Business Rules:** Rolling the dice must result in an integer between 1 and 6 (inclusive) to be returned
- **Constraints:** None
- **Interfaces:** The value of rolled dice output is an integer.
- **Interactions:** The Dice subsystem interacts with the Game Board when it needs to determine how far a user's mover will move.

2.2. Question Bank Subsystem

Loader

- **Encapsulated Information:** Maintains the path to the directory which houses flat files used in the storage of questions of various categories. In this regard it encapsulates the question bank, loading content from flat files to be returned as objects in memory.
- **Inputs:** NA
- **Outputs:** Data structure containing all questions in the test bank organized by category.
- **Intended Functionality:** Manage the parsing of files that contain questions to be used in playing the game and return them to various subsystems that involve viewing questions or retrieving them for a match.
- **Business Rules:** User or subsystem asks to view the questions in the game and the Loader retrieves them from file storage
- **Constraints:** Files containing questions must be of a specific, predetermined, and consistent format. They must also be located in the correct directory.
- **Interfaces:** Accessible endpoint to trigger the loading of questions and return them in a data structure.
- **Interactions:** Called by the UI subsystem of the Question Bank Subsystem for question viewing, editing, and adding. Also called by the game board subsystem when a match is initiated.

Writer

- **Encapsulated Information:** New questions to be added to the question bank

- **Inputs:** Initialization of the writer requires the current set of questions in the game loaded in memory. From there it takes input strings in the form of questions, answers, and question types from users to be added to the question bank. Upon receiving a signal, it then saves all the current questions in memory to the directory read by the Loader subsystem.
- **Outputs:** Set of files containing questions to be used in game matches
- **Intended Functionality:** To save questions in memory to files that are used for question storage
- **Business Rules:** Given a set of questions, facilitate the adding of new questions and the writing of all questions to files.
- **Constraints:** Must agree with the Loader subsystem as to where the files will be written and what format they will be written in.
- **Interfaces:** Can accept signals to save a current set of questions loaded in memory to files. Also interfaces with users to facilitate the addition of new questions.
- **Interactions:** Interacts with the Loader subsystem to retrieve all the questions currently stored in the question bank files and with UI displays the questions. Subsystem will also retrieve user input for the creation of new questions and save the complete set of historical and new questions to the file system.

Question Bank User Interface

- **Encapsulated Information:** Encapsulates the Loader and Writer in addition to handling the hand-off of in memory question bank from the Loader to the Writer.
- **Inputs:** Accepts signals from user to either load or save question bank from file system. Also receives strings from users that are to be added to the question bank. This information from users must be a question string, an answer string, and a category for each new question.
- **Outputs:** UI elements displaying if a load or save was successful, also if a new question has been added to the question bank.
- **Intended Functionality:** Facilitate user interaction with the question bank, providing options to load and view existing questions, add new questions, and save them.
- **Business Rules:** User may add questions to the question bank so long as they provide both a question and an answer and assign it to a currently supported category.
- **Constraints:** User interface for viewing questions must be effective on both large and small sets of questions.
- **Interfaces:** Interfaces with user via UI and Loader and Saver via APIs.

- **Interactions:** Interacts with users to trigger loading and saving of questions. Also interacts with users to receive new questions that are to be added to the current collection.

2.3. Start Game Subsystem

User Interface

- **Encapsulated Information:** Encapsulates initiating interactive state with Question Bank Subsystem and Game Board subsystem.
- **Inputs:** Response from user clicking buttons to either start the game with N players or to view/edit question bank.
- **Outputs:** UI elements indicating user selection.
- **Intended Functionality:** Serve as the main interface for when users start the software, orchestrating the selection of one of the two main usage modes: playing the game or editing questions.
- **Business Rules:** User will start the software and be presented with two options, start a match with N players or edit the question bank.
- **Constraints:** UI must be clean and minimal.
- **Interfaces:** UI of the Start Game subsystem will interface with both the Question Bank and Game Board subsystems. It will interface with the user via a collection of menu items and buttons.
- **Interactions:** Users can press buttons to transition from the Start Game UI to the UI of the Board Game subsystem or Question Bank subsystem. After the selection has been made, the UI of the selected subsystem will be in charge of interfacing with the user.

3. Functional Requirements (Use Cases)

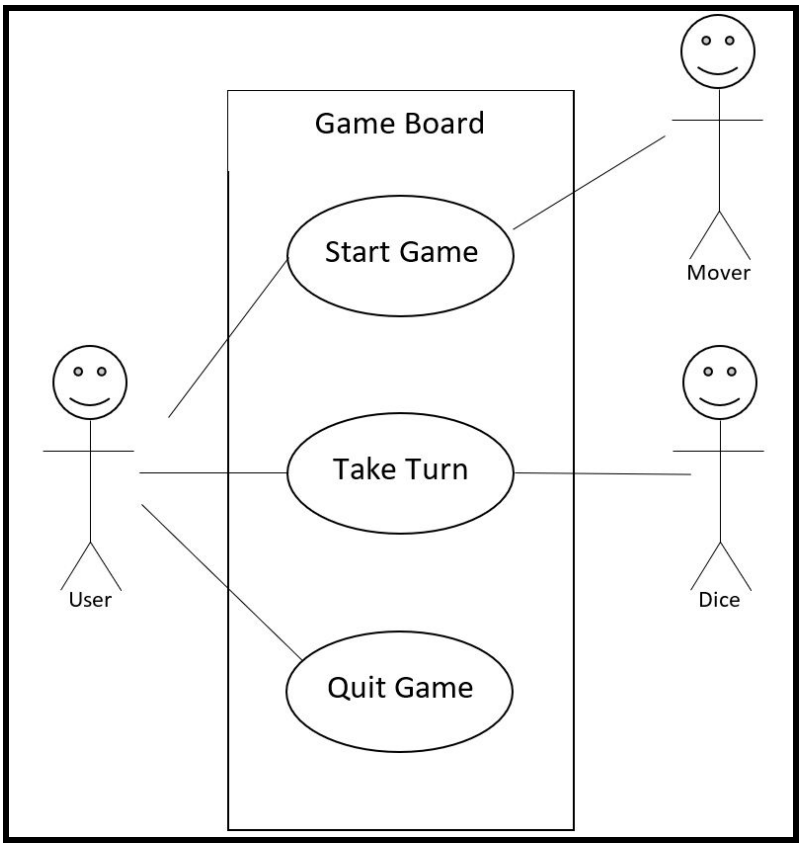
3.1. Take Turn

Summary Information	
Use Case Name	**Take Turn**
Use Case Number	**Game_Board_01**
Use Case Goal	**Take a turn, answer a question, and change positions (if applicable).**
Trigger	**The user whose turn it is selects the option to roll the dice.**
Actors	**User, Game Board, Game Position, Card Deck, Card, Player, Mover, Dice**
Pre-conditions	**It is the user's turn.**
Post-conditions	**If question answered correctly:** ● **User collects relevant wedge** ● **Use case restarts**
Assumptions	**None**
Related Use Cases	**Mover_01, Card_Deck_01, Card_01, Mover_02, Game_Board_01**

Main Success Scenario	
Actor Action	**System Response**
1. User selects to roll dice	2. Roll Dice
	3. System prompts the user to select a direction to move.
4. User selects a direction.	
	5. Update Game Position
	6. Retrieve Category
	7. Select Card
	8. Display Question

9. User answers question	10. Answer is correct.
	11. Collect Wedge
12. Take Turn Quit Game	
Alternate Scenarios	
User lands on roll again square (at step 5)	
	1. System prompts user to roll dice again
User's answer is incorrect (at step 10)	
	1. System updates whose turn it is.
2. Take Turn Quit Game	

Context Diagram for Take Turn (Game_Board_01)



3.2. Quit Game

Summary Information	
Use Case Name	**Quit Game**
Use Case Number	**Game_Board_02**
Use Case Goal	**End the game**
Trigger	**User whose turn it is selects the option to quit the game.**
Actors	**User, Game Board**
Pre-conditions	**It is the user's turn.**
Post-conditions	**Game is ended.**
Assumptions	**None**
Related Use Cases	**None**

Main Success Scenario	
Actor Action	**System Response**
1. User selects option to quit	2. System terminates
Alternate Scenarios	
N/A	

3.3. Start Game

Summary Information	
Use Case Name	**Start Game**
Use Case Number	**Game_Board_03**
Use Case Goal	**Start and display the game and pick starting position.**
Trigger	**User selects the option to start the game.**

Actors	**User, Game Board, Start Game Subsystem**
Pre-conditions	**None**
Post-conditions	**Game is active.**
Assumptions	**None**
Related Use Cases	**Game_Board_01**

Main Success Scenario	
Actor Action	**System Response**
1. User selects the option to start from Start Game UI	2. System begins and displays the game board.
	3. Randomize Position
Alternate Scenarios	
N/A	

3.4. Select Card

Summary Information	
Use Case Name	**Select Card**
Use Case Number	**Card_Deck_01**
Use Case Goal	**Retrieve a question from the question bank.**
Trigger	**Category is returned in Retrieve Category.**
Actors	**Mover, Card Deck**
Pre-conditions	**None**
Post-conditions	**Question is removed from the question bank.**
Assumptions	**None**

Related Use Cases	**Game_Position_01**

Main Success Scenario	
Actor Action	**System Response**
1. Category sent to system.	2. System retrieves a random question from the relevant category.
	3. Question is removed from system storage.
	4. Display Question
Alternate Scenarios	
Category has run out of questions (at step 2)	
	1. System changes the user.
	2. Take Turn

3.5. Display Question

Summary Information	
Use Case Name	**Display Question**
Use Case Number	**Card_01**
Use Case Goal	**To display the selected question to the user.**
Trigger	**Question is retrieved from the question bank.**
Actors	**Card Deck, Card**
Pre-conditions	**None**
Post-conditions	**None**
Assumptions	**None**
Related Use Cases	**Card_Deck_01**

Main Success Scenario	
Actor Action	**System Response**
1. The question is sent to the system.	2. System displays the question to the user.
Alternate Scenarios	
N/A	

3.6. Update Game Position

Summary Information	
Use Case Name	**Update Game Position**
Use Case Number	**Mover_01**
Use Case Goal	**To update the game position to be the current position plus the amount rolled by the dice.**
Trigger	**Dice roll returns a number.**
Actors	**Dice, Mover**
Pre-conditions	**None**
Post-conditions	**None**
Assumptions	**None**
Related Use Cases	**Dice_01**

Main Success Scenario	
Actor Action	**System Response**
1. Dice roll sends the number to the system.	2. System updates the position of the mover to the current position plus the number of the dice roll.
	3. Retrieve Category Re-Take Turn

Alternate Scenarios
N/A

3.7. Collect Wedge

Summary Information	
Use Case Name	**Collect Wedge**
Use Case Number	**Mover_02**
Use Case Goal	**Wedge for the relevant category is populated in the Mover.**
Trigger	**User correctly answers the posed question.**
Actors	**Card, Mover, User, Category**
Pre-conditions	**Mover does not already contain the relevant category.**
Post-conditions	**Mover contains the relevant wedge.**
Assumptions	**None**
Related Use Cases	**Game_Board_01**

Main Success Scenario	
Actor Action	**System Response**
1. User answers the question correctly.	2. System adds the relevant wedge to the mover.
Alternate Scenarios	
N/A	

3.8. Randomize Game Position

Summary Information	
Use Case Name	**Randomize Game Position**

Use Case Number	**Mover_03**
Use Case Goal	**To randomly select a position for the user to begin the game on.**
Trigger	**User starts the game.**
Actors	**Dice, Mover**
Pre-conditions	**None**
Post-conditions	**None**
Assumptions	**None**
Related Use Cases	**Game_Board_03**

Main Success Scenario	
Actor Action	**System Response**
1. User selects option to begin a new game.	2. System updates the position of the mover to a random position on the outside of the board.
	3. Take Turn Quit Game
Alternate Scenarios	
N/A	

3.9. Retrieve Category

Summary Information	
Use Case Name	**Retrieve Category**
Use Case Number	**Game_Position_01**
Use Case Goal	**Retrieve the category of the question from the position of the mover on the game board.**
Trigger	**Mover's position on the board is updated.**

Actors	**Mover, Game Position**
Pre-conditions	**Position is a category space and not a 'roll again' space.**
Post-conditions	**None**
Assumptions	**None**
Related Use Cases	**Mover_01**

Main Success Scenario	
Actor Action	**System Response**
1. System receives the updated position of the mover.	2. System retrieves and displays the category held by that position.
	3. Select Card
Alternate Scenarios	
N/A	

3.10. Re-take Turn

Summary Information	
Use Case Name	**Re-take Turn**
Use Case Number	**Game_Position_02**
Use Case Goal	**User takes another turn to move to another position.**
Trigger	**Mover lands on a 'roll again' space.**
Actors	**Mover, Game Position, User**
Pre-conditions	**Position is a 'roll again' space and not a category space.**
Post-conditions	**None**
Assumptions	**None**
Related Use Cases	**Mover_01**

Main Success Scenario	
Actor Action	**System Response**
1. Mover lands on a ‘roll again’ space.	2. System resets the turn for the same user.
3. User - Take Turn	
Alternate Scenarios	
User decides to quit instead of playing another turn (at step 3)	
1. Quit Game	

3.11. Roll Dice

Summary Information	
Use Case Name	**Roll Dice**
Use Case Number	**Dice_01**
Use Case Goal	**To simulate the rolling of dice and return a value of the sum.**
Trigger	**User selects the option to roll the dice to initiate their turn.**
Actors	**User, Game Board, Dice**
Pre-conditions	**None**
Post-conditions	**None**
Assumptions	**None**
Related Use Cases	**Game_Board_01**

Main Success Scenario	
Actor Action	**System Response**
1. Request for dice roll sent to system.	2. System simulates the rolling of dice.

	3. System returns and displays the sum of the rolled dice.
	4. Update Game Position
Alternate Scenarios	
N/A	

Supplementary Specification

Table of Contents

1. Introduction

Please reference section 1 (Introduction) of the Vision Document.

2. Usability

- **Maximum required training times to use product:**
    - Normal user: 10 minutes (learn game mechanics)
    - Teacher/Administrator: 15 minutes (learn game mechanics, set-up multiplayer game)
- **Maximum times to perform common tasks:**
    - Install program: 10 minutes (using recommended hardware specifications)
    - Set up game (single or multi-player): 5 minutes
    - Take turn: 30 seconds
    - Answer question: 30 seconds
    - Decide which direction to move: 30 seconds
    - Uninstall program: 5 minutes (using recommended hardware specifications)
- UI resembles physical Trivial Pursuit game board
- Menus include clearly defined options for users to navigate the program (back buttons, large/colored font, etc.)
- A tutorial is provided to instruct new users on game mechanics in the form of text
- User actions are primarily performed through mouse clicks on objects (such as clicking a die to roll)
- When a user types an answer to a question, the response is case-insensitive. However, an incorrectly spelled answer will be wrong.
- Answers to questions avoid using punctuation marks, in order to limit user frustration over grammatical errors

3. Reliability

Availability:

- Product must be available and usable 24/7 with no more than 1% downtime
- Online download and patching services must be available 24/7 with no more than 1% downtime. Online services may be down for a one-hour maintenance period once per week during U.S. nighttime hours.

Mean Time Between Failures: 1 year

Mean Time to Repair: 12 hours

Accuracy: User inputs must be accurately processed 99.9% of the time

Maximum Bugs/Defect Rate:

- Critical (leads to program failure): 0
- Significant (hinders proper gameplay): 1 bug per 100 lines of code
- Minor (does not affect overall gameplay): 5 bugs per 100 lines of code

4. Performance

(Based on recommended hardware specifications)

System Response Time:

- Average: 3 seconds
- Maximum: 15 seconds

Recovery Time:

- Average: 10 seconds
- Maximum: 30 seconds

Startup Time:

- Average: 10 seconds
- Maximum: 30 seconds

Capacity: 2-4 users

Degradation Mode: If online patching services are unavailable, user will still be able to run program with older software version.

Resource Utilization:

- Memory: 4 GB RAM
- Storage: 5 GB

5. Supportability

Testability: Standardized software tests will be produced to ensure that required functional and non-functional requirements are consistently met by existing software. Furthermore, testing criteria documents will be provided to alpha/beta testers with standardized instructions of what functions need to be tested.

Adaptability: Software will be modularized and well documented in order to promote high adaptability for future software changes.

Maintainability: Errors and exceptions discovered after initial release must be fixed via patches/hotfixes, which will be available online for users to download. All bugs must have patches developed within one week of discovery.

Compatibility: Program must be compatible with all major operating systems (Windows, MacOS, Linux) released after 2005 that have Python 3 installed.

Configurability: User must be able to configure/customize their own Player Name and create custom question banks. Program must be able to accept correctly formatted question banks and upload them into a new game.

Installability: Any computer that meets minimum required hardware specifications and has Python 3 installed should be able to install the program.

Scalability: Each individual game must support up to 4 players. All players can either play on a single computer, or networked together via a local area network.

6. Design Constraints

Software Language: Due to limited team knowledge on programming languages, the program must be developed in Python.

Software Process Requirement: Due to strict project deadlines, a Design-to-Schedule life cycle model must be used. This limits our ability to utilize Agile software development methods.

Developmental Tools: Due to a $0 budget, our team must use free developmental and video editing software.

Architecture: Program must follow Object-Oriented design.

Class Libraries: Due to limited team knowledge on Python GUI programming, Tkinter must be used for GUI functions.

7. Documentation & Help System Requirements

Program will include a README.txt file with each installation, granting users a local help file. There are currently no requirements for help features within the program (aka a Help button) or via an online webpage, however they could be implemented in future versions (Dream feature).

8. Purchased Components

There are no purchased component requirements. All software development and project planning is being conducted using free, open-source tools. Furthermore, there are no component requirements from Hasbro since they have waived their copyright rights for the product.

9. User Interfaces

User interface must be a GUI that closely resembles a physical Trivial Pursuit game board (though a square instead of a circle). This GUI must be developed via Python3 using the installed package, Tkinter due to the team's programming knowledge.

User interface will primarily be composed of buttons that users click. A few text boxes will be provided, the main ones being for users to input their names and for users to submit answers to questions.

For local area network gameplay, multiple users must have computers with Ethernet or wireless network interfaces.

10. Hardware Interfaces

User's computer must meet required hardware specifications:

- x86 64-bit CPU (Intel / AMD architecture)
- 4 GB RAM
- 5 GB storage
- Network Interface Card

Must support all Windows, Linux, and Mac operating systems released after 2005. It is expected that users have a keyboard, mouse, and display monitor attached to their computer.

11. Software Interfaces

Software Used	Description
Operating System	We have not specified a specific operating system for use of this program. Windows, Linux, and Mac operating systems compatible with Python3 will suffice.
Development Language	This program has been written in Python3 which was chosen for ease of use. Additionally, Tkinter is a package required for use of Python GUI methods.

Planning	Planning for this project has been tracked using Google Drive for easy sharing, updates and version control tracking

12. Communication Interfaces

Program must be capable of connecting to other user computers through an Ethernet or wireless local area network. All communication with the user will be done via Python3 Tkinter GUI's.

13. Licensing, Legal and Copyright Requirements

Hasbro has waived copyright rights for the product, therefore there are no licensing or usage restrictions and no requirement to interface with any other Hasbro product or service. However, we will need to copyright Trivial Purfuit to ensure that the brand is not used by third-parties for financial gain.

14. Applicable Standards

Our team adheres to IEEE software engineering standards as outlined in the Guide to the Software Engineering Body of Knowledge, version 3 (Bourque and Fairley). These provide our team guidance on common industry standards regarding requirements elicitation, software design, testing, maintenance, configuration management, and so on.

Works Cited

Bourque, Pierre and Richard E Fairley. "Guide to the Software Engineering Body of Knowledge." 2014. *IEEE Computer Society.* Document. 29 May 2020.