Introduction:

This project involves the development of a Discord bot featuring various mini-games and interactive commands. The bot is designed to entertain users within Discord servers.

Overview:

The Discord bot includes the following mini-games and features:

- Magic 8-Ball for answering user questions
- Tic-Tac-Toe game with player vs. player functionality
- Rock, Paper, Scissors game against the bot
- Guessing Game where users attempt to guess a randomly generated number
- Useless Facts command to provide users with random facts

The Discord bot is implemented in Python using the Discord.py library. Each mini-game has its own set of functions and commands. Key design elements include:

- Modular approach with separate classes for certain games
- Use of APIs for fetching random facts
- Integration of various commands for user interaction

Solution Design:

Architecture:

The bot is implemented using the Discord.py library, which offers a Python interface for interacting with the Discord API. The architecture follows a modular design, with distinct components for each game and additional features.

Modules and Features:

Magic 8-ball:

The Magic 8-Ball feature allows users to pose questions to the bot and receive random responses akin to the classic Magic 8-Ball toy. Users can ask questions using the '!8ball' command, and the bot responds with a randomized answer, adding an element of fun and unpredictability to user interactions.

Command Handling:

- The !8ball command is defined in the bot script using the '@bot.command' decorator from the Discord.py library.
- The command takes an optional argument, question, which represents the user's query.

```python
import discord
from discord.ext import commands
import random
#8 ball responses
magic_8_ball_responses = [
    "Yes",
    "No",
    "Ask again later",
    "Cannot predict now",
    "Don't count on it",
    "Most likely",
    "Outlook not so good",
    "You may rely on it"
]

# Command to start Magic 8-Ball
@bot.command(name='8ball')
async def magic_8_ball(ctx, *, question: str = "Will I get good luck today?"):
```

Response Generation:

- A list of possible responses is predefined (magic_8_ball_responses).
- The bot randomly selects a response from the list for each user query.

```python
import discord
from discord.ext import commands
import random
 # Select a random response from the list
 response = random.choice(magic_8_ball_responses)

 # Send the response to the user
 await ctx.send(f"**Question:** {question}\n**Answer:** {response}"')
```

Useless Facts:

The Useless Facts feature enriches user engagement by providing random, interesting facts. The bot fetches data from an external API to deliver a diverse range of useless yet entertaining information. Users can trigger the fact retrieval using the '!fact' command, adding an element of curiosity to the server.

Command Handling and API Integration:

- The requests library is used to make HTTP requests to the Useless Facts API (USELESS_FACTS_API_URL)
- The '!fact' command is defined similarly to the Magic 8-Ball command.

```python
import discord
from discord.ext import commands
import random
import requests
#command to generate facts
@bot.command(name='fact')
async def get_useless_fact(ctx):
    try:
        response = requests.get(USELESS_FACTS_API_URL)
        data = response.json()
        useless_fact = data['text']
        await ctx.send(f"Here's a random useless fact: {useless_fact}")
    except Exception as e:
        print(f"An error occurred: {e}")
        await ctx.send("Error: Unable to fetch a random useless fact at the moment.")
```

Tic-Tac-Toe:

Tic-Tac-Toe offers users a classic two-player game experience within the Discord server. Users initiate a new game with the '!tictactoe' command and make moves using the '!move command'. The bot updates the game board, checks for a winner, and switches players accordingly, creating an engaging gaming environment.

Board Representation:

- The game board is represented as a 2D list (board).
- The 'display_board' function is responsible for formatting and displaying the current board state.

```python
import discord
from discord.ext import commands
import random
import requests

# Function to display the Tic-Tac-Toe board
2 usages
def display_board(board):
    return '\n'.join([' '.join(row) for row in board])
```

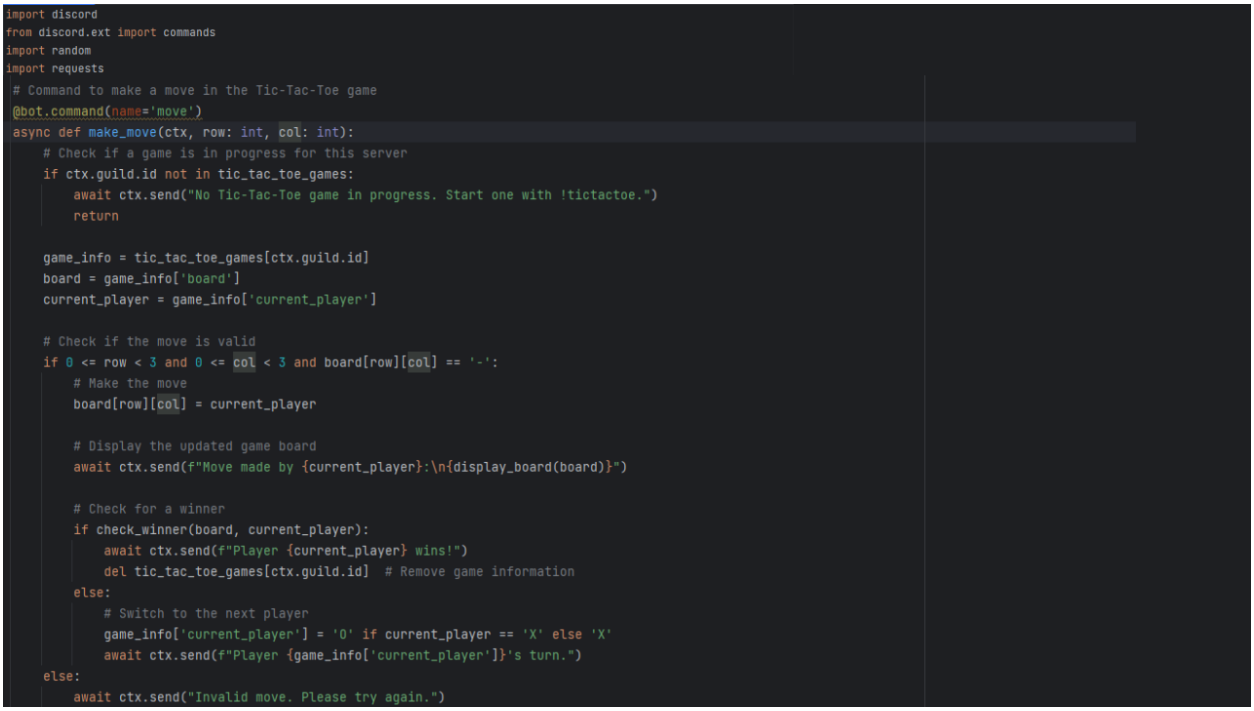
Game Initialization:

- The ‘start_tic_tac_toe’ command initializes a new game by creating an empty board and setting the current player to 'X'.

```python
import discord
from discord.ext import commands
import random
import requests

# Command to start a new Tic-Tac-Toe game
@bot.command(name='tictactoe')
async def start_tic_tac_toe(ctx):
    # Check if a game is already in progress for this server
    if ctx.guild.id in tic_tac_toe_games:
        await ctx.send("A Tic-Tac-Toe game is already in progress!")
        return

    # Initialize a new game board
    board = [['-' for _ in range(3)] for _ in range(3)]
    tic_tac_toe_games[ctx.guild.id] = {'board': board, 'current_player': 'X'}

    # Display the initial game board
    await ctx.send(f"New Tic-Tac-Toe game started!\n{display_board(board)}\nPlayer X's turn.")
```

Move Handling:

- The ‘make_move’ command processes player moves, checks for move validity, updates the board, and checks for a winner.



```python
import discord
from discord.ext import commands
import random
import requests
# Command to make a move in the Tic-Tac-Toe game
@bot.command(name='move')
async def make_move(ctx, row: int, col: int):
    # Check if a game is in progress for this server
    if ctx.guild.id not in tic_tac_toe_games:
        await ctx.send("No Tic-Tac-Toe game in progress. Start one with !tictactoe.")
        return

    game_info = tic_tac_toe_games[ctx.guild.id]
    board = game_info['board']
    current_player = game_info['current_player']

    # Check if the move is valid
    if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == '-':
        # Make the move
        board[row][col] = current_player

        # Display the updated game board
        await ctx.send(f"Move made by {current_player}:\n{display_board(board)}")

        # Check for a winner
        if check_winner(board, current_player):
            await ctx.send(f"Player {current_player} wins!")
            del tic_tac_toe_games[ctx.guild.id]  # Remove game information
        else:
            # Switch to the next player
            game_info['current_player'] = 'O' if current_player == 'X' else 'X'
            await ctx.send(f"Player {game_info['current_player']}'s turn.")
    else:
        await ctx.send("Invalid move. Please try again.")
```

Rock, Paper, Scissors:

The choice (rock, paper, or scissors). The bot then reveals its choice, determines the winner, and provides a response, adding an interactive gaming aspect to the server.

Rock, Paper, Scissors (RPS) feature enables users to play the classic hand game against the bot. Users can make their move using the '!rps' command followed by their

Game Initialization:

- An instance of the 'RPSGame' class is created when a new game is started, and the bot's choice is randomly set.

```python
import discord
from discord.ext import commands
import random
import requests

#Command to start RPS Game
1 usage
class RPSGame:
    def __init__(self):
        self.choices = ["rock", "paper", "scissors"]
        self.bot_choice = None

    2 usages (2 dynamic)
    def start_game(self):
        self.bot_choice = random.choice(self.choices)

    2 usages (2 dynamic)
    def determine_winner(self, player_choice):
        if player_choice == self.bot_choice:
            return f"It's a tie! I also chose {self.bot_choice}."
        elif (
            (player_choice == "rock" and self.bot_choice == "scissors")
            or (player_choice == "paper" and self.bot_choice == "rock")
            or (player_choice == "scissors" and self.bot_choice == "paper")
        ):
            return f"You win! I chose {self.bot_choice}."
        else:
            return f"You lose! I chose {self.bot_choice}."
```

Move Handling:

- The ‘make_move’ command initiates the game, checks for a valid player move, determines the winner, and displays the result.

```python
import discord
from discord.ext import commands
import random
import requests

# Command for the player to make a move in the Rock, Paper, Scissors game
@bot.command(name='rps')
async def make_move(ctx, player_choice: str = None):
    global rps_game_info

    if ctx.guild.id not in rps_game_info:
        rps_game_info[ctx.guild.id] = RPSGame()
        rps_game_info[ctx.guild.id].start_game()

    rps_game = rps_game_info[ctx.guild.id]

    if player_choice:
        result = rps_game.determine_winner(player_choice.lower())
        del rps_game_info[ctx.guild.id]  # Remove game information
        await ctx.send(result)
    else:
        await ctx.send("Please provide your move: !rps [rock/paper/scissors].")
```

Guessing Game:

The Guessing Game feature challenges users to guess a randomly generated number within a specified range. Users start the game with the '!guess' command, and the bot provides feedback on each guess, indicating whether the correct number has been guessed. The game concludes with a congratulatory message upon successful guessing

Game Initialization:

- The 'start_guessing' command initiates a new game by generating a random number between 1 and 100 for the user to guess.

```python
import discord
from discord.ext import commands
import random
import requests
#command to start guessing game
@bot.command(name='guess')
async def start_guessing(ctx):
    # Check if a game is already in progress for this server
    if ctx.guild.id in guessing_game_info:
        await ctx.send("A guessing game is already in progress!")
        return

    # Generate a random number for the user to guess (between 1 and 100)
    secret_number = random.randint( a: 1,  b: 100)

    # Store game information for this server
    guessing_game_info[ctx.guild.id] = {'secret_number': secret_number, 'attempts': 0}

    await ctx.send("Welcome to the Guessing Game! I have selected a number between 1 and 100.")
```

User Guess Handling:

- The 'on_message' event is used to capture user messages and process them as guesses. The 'guess_number' function checks if the guessed number is correct, too low, or too high.

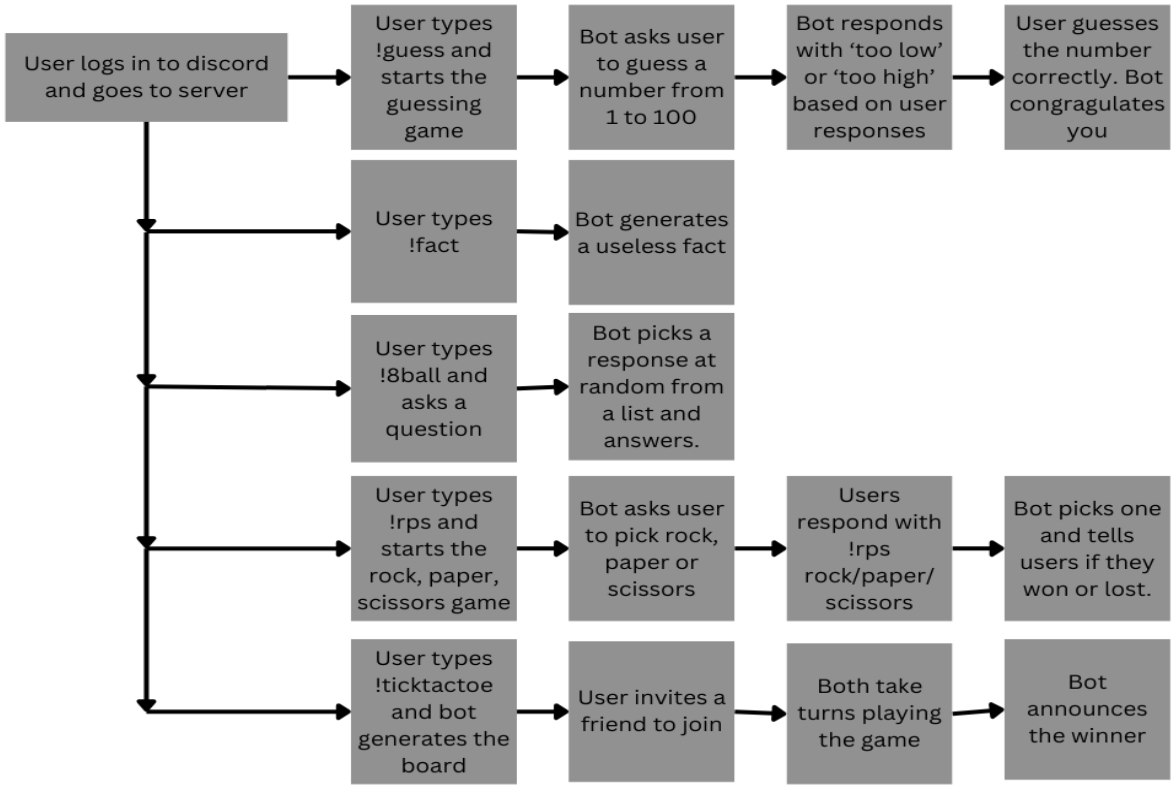
```python
import discord
from discord.ext import commands
import random
import requests
# Event to handle user messages
@bot.event
async def on_message(message):
    # Check if a game is in progress for this server
    if message.guild and message.guild.id in guessing_game_info:
        try:
            number = int(message.content)
            await guess_number(message.guild.id, number, message.channel)
        except ValueError:
            pass

    await bot.process_commands(message)
```

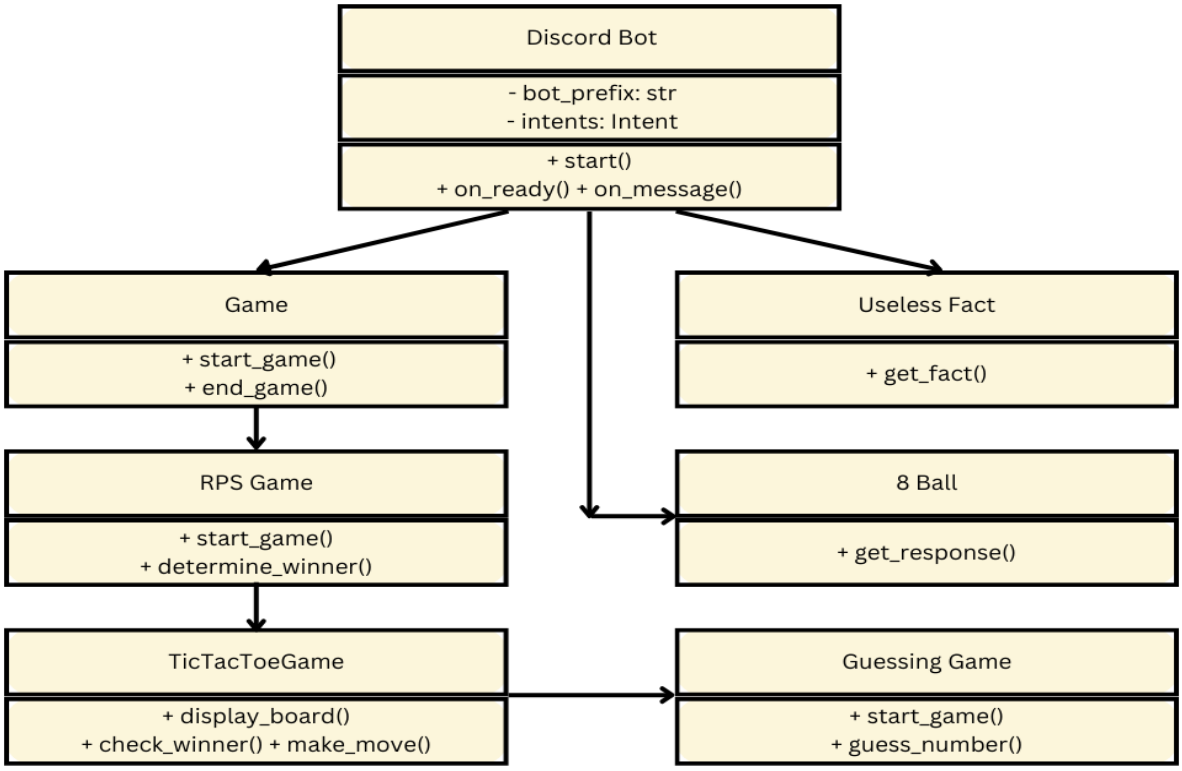
Guess Evaluation:

- The ‘guess_number’ function evaluates the user's guess, updates the number of attempts, and sends appropriate messages based on the correctness of the guess.

```python
import discord
from discord.ext import commands
import random
import requests
async def guess_number(guild_id, number, channel):
    # Check if a game is in progress for this server
    if guild_id not in guessing_game_info:
        return

    game_info = guessing_game_info[guild_id]
    game_info['attempts'] += 1

    # Check if the guessed number is correct
    secret_number = game_info['secret_number']
    attempts = game_info['attempts']

    if number == secret_number:
        del guessing_game_info[guild_id]  # Remove game information
        await channel.send(f"Congratulations! You guessed the number {secret_number} in {attempts} attempts.")
    elif number < secret_number:
        await channel.send("Too low! Try again.")
    else:
        await channel.send("Too high! Try again.")
```

Activity and Class Diagrams:





Dependencies:

- **Discord.py:** Provides an interface for interacting with the Discord API.
- **Requests:** Used for making HTTP requests to fetch useless facts from an external API.
- **Random:** Python built-in module for generating random numbers and choices.

Conclusion:

The solution design prioritizes modularity, code organization, and external API integration for additional features. The architecture allows for easy scalability, facilitating the addition of new games and features in the future. Additionally, the inclusion of standard Python modules such as ‘random’ enhances the bot's functionality.