Sviluppo di bot telegram in Python

Progetto di ingegneria informatica
Docente Tutor: Prof. Giovanni Agosta
Repository GitHub: https://github.com/FlavioMili/LupusInTabula

Flavio Milinanni
10915991

Obiettivo del Progetto

L’obiettivo del progetto era la realizzazione di un bot Telegram per una variante digitale del celebre gioco di ruolo **Lupus in Tabula**, o anche conosciuto come **Werewolf**.
Il bot doveva gestire automaticamente le fasi di gioco, assegnare ruoli ai giocatori, permettere l’interazione tramite comandi e pulsanti, e determinare il vincitore in base alle regole specificate.
Per raggiungere questo obiettivo, è stato necessario definire chiaramente le specifiche da implementare, tra cui:

- **Gestione dei ruoli**: Il bot deve assegnare casualmente i ruoli ai giocatori e comunicare privatamente a ciascuno la propria identità.
- **Dinamiche di gioco**: Il bot deve supportare la sequenza di turni (notte e giorno) con meccanismi automatizzati per la gestione delle azioni notturne e delle votazioni diurne.
- **Interfaccia utente**: Gli utenti devono poter interagire con il bot tramite comandi intuitivi e pulsanti per semplificare l’esperienza di gioco.
- **Automazione del flusso di gioco**: Il bot deve gestire i timer per limitare la durata delle fasi e garantire il corretto svolgimento della partita.
- **Condizioni di vittoria**: Il sistema deve controllare se sono state soddisfatte le condizioni per decretare la vittoria di uno dei due schieramenti.

Queste specifiche hanno permesso di delineare una struttura chiara per lo sviluppo del progetto, assicurando che il gioco seguisse un flusso logico e coerente. In una seconda fase, verrà presentata la soluzione implementata per soddisfare questi requisiti.

La specifica originale può essere visionata qui:
https://docs.google.com/document/d/1HOELOXX7dnxqk7l_ooBUNe2E4qlkrwUDj0p4hejnUGw/edit?usp=sharing

Processo di Sviluppo

Tecnologie Utilizzate

Per lo sviluppo del progetto sono state utilizzate le seguenti tecnologie:

- **Python**: Linguaggio di programmazione principale.
- **python-telegram-bot**: Libreria per l'interazione con l'API di Telegram e la gestione del bot.
- **dotenv**: Per la gestione sicura delle variabili d'ambiente, inclusa la chiave API del bot.
- **Git e GitHub**: Per il versionamento del codice.

Scelta della Libreria

Per lo sviluppo del bot si è scelto di utilizzare la libreria `python-telegram-bot`, che offre un'interfaccia robusta e ben documentata per l'interazione con l'API di Telegram.
Questa libreria permette di gestire aggiornamenti asincroni, handler per i comandi e un sistema di job scheduler per automatizzare fasi di gioco basate sul tempo.
La documentazione si trova al seguente link: https://docs.python-telegram-bot.org/en/v21.10/.

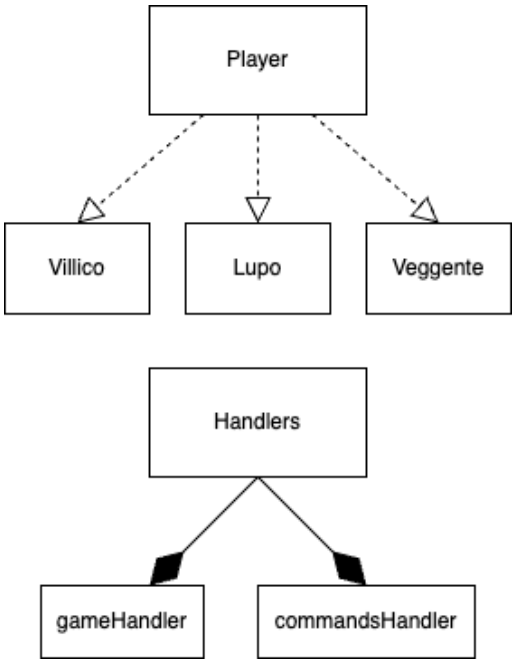
Scelte Implementative

L'implementazione del bot è stata progettata seguendo un'architettura modulare, separando la logica del gioco dalla gestione dei comandi e dell'interfaccia utente. Questa scelta ha permesso di mantenere un codice più organizzato e facilmente estendibile.
L'uso della libreria `python-telegram-bot` ha semplificato la gestione delle interazioni tra bot e utenti, mentre la suddivisione in classi ha garantito una chiara definizione delle responsabilità di ciascun componente.
Per ottimizzare l'esperienza di gioco, sono stati introdotti timer automatici che regolano il flusso della partita senza la necessità di intervento manuale, migliorando la fluidità del gameplay. Questi possono essere modificati e adattati allo stile di gioco del gruppo attraverso la modifica del file /config.py.
A livello di interfacce, molti comandi sono gestiti da bottoni in modo da non dover scrivere nulla per effettuare le proprie mosse. L'unico modo in cui gli utenti devono interagire con la tastiera è per discutere della strategia di gioco.

Struttura del Progetto

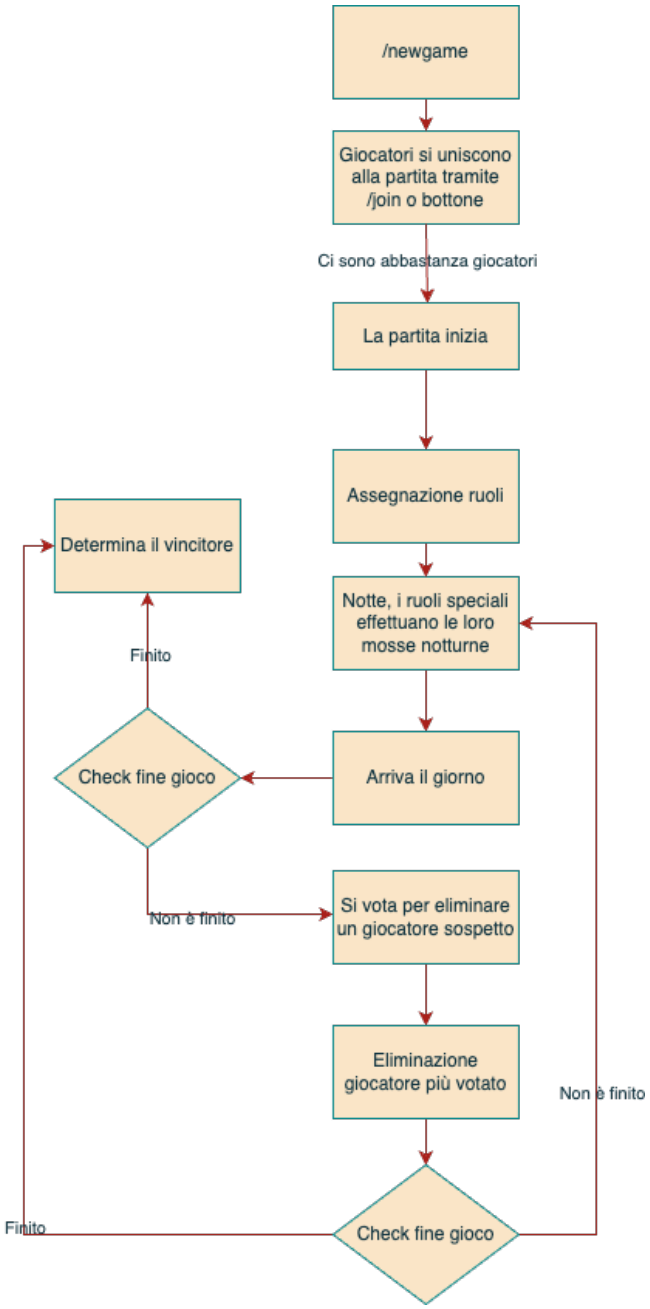
Il progetto è stato sviluppato in Python utilizzando la libreria `python-telegram-bot`. La struttura dei file è la seguente:

- `main.py`: Punto di ingresso del bot, gestisce l'inizializzazione del gioco e l'ascolto dei comandi.
- `config.py`: Contiene messaggi predefiniti e timer configurabili.
- `handlers/`:
  - commandsHandler: Contiene la logica per la gestione dei comandi.
  - gameHandler: Contiene la logica per la gestione delle partite.
- `classes/`:
  - Game: Gestisce lo stato del gioco, i turni e i giocatori.
  - Player: Rappresenta un giocatore e il suo ruolo.
  - Classi dei Ruoli: Implementano i comportamenti specifici per ogni ruolo.
  - enums.py: Definisce gli stati di gioco e i ruoli disponibili (anche come string).



Funzionamento del Bot

- **Avvio della partita**: Un giocatore crea una partita con il comando `/newgame`, gli altri si uniscono con `/join` o usando il bottone “Partecipa”.
- **Inizio automatico**: Dopo un tempo di attesa predefinito o dopo il comando `/startgame`, il gioco assegna i ruoli e inizia la fase notturna.
- Fasi di gioco:
  - **Notte**: I Lupi selezionano una vittima in chat privata, il Veggente può vedere il ruolo di un altro giocatore.
  - **Giorno**: I giocatori discutono e votano tramite pulsanti in chat privata.
- **Fine partita**: Il gioco termina automaticamente quando si verifica la condizione di fine gioco, ovvero se non ci sono più lupi o se il numero di lupi è pari a quello dei contadini



Il commandsHandler gestisce la funzione newGame utilizzata per iniziare la partita.

```
async def newGame(update: Update, context: ContextTypes.DEFAULT_TYPE):
    chat_id = update.effective_chat.id
    if chat_id in gameHandler.games:
        await update.message.reply_text("Partita gia' in corso")
        return
    if update.effective_chat.type == 'private':
        await update.message.reply_text("Puoi iniziare una partita solo in un gruppo")
        return

    gameHandler.newGame(chat_id, context.application)
    keyboard = [[InlineKeyboardButton("Partecipa!", callback_data="join")]]
    reply_markup = InlineKeyboardMarkup(keyboard)

    join_msg = await update.message.reply_text(
        f"Nuova partita iniziata, premi il pulsante per partecipare.\nMin: {config.MIN_GIOCATORI} giocatori, Max: {config.MAX_GIOCATORI}
giocatori.\nRicorda di iniziare una chat con il bot e di impostare un username",
        reply_markup=reply_markup
    )
    gameHandler.games[chat_id].join_message_id = join_msg.message_id
```

Ci si accerta che non vi siano partite già attive nel gruppo ed eventualmente ne inizia una, dando la possibilità ai giocatori di partecipare schiacciando un bottone che utilizza la funzione joinGame.

```
async def joinGame(update: Update, context: ContextTypes.DEFAULT_TYPE, from_button=False):
    if from_button:
        query = update.callback_query
        await query.answer()
        chat_id = query.message.chat.id
        user_id = query.from_user.id
        username = query.from_user.username or query.from_user.first_name
    else:
        chat_id = update.effective_chat.id
        user_id = update.effective_user.id
        username = update.effective_user.username or update.effective_user.first_name

    if chat_id not in gameHandler.games:
        response_text = "Nessuna partita in corso, usa /newgame per iniziarne una"
    else:
        game = gameHandler.games[chat_id]
        if game.state != GameState.WAITING:
            response_text = "La partita e' gia' iniziata"
        elif await game.addPlayer(user_id, username):
            response_text = f"@{username} si e' unito al gioco\nGiocatori: {len(game.players)}/10"
        else:
            response_text = f"@{username} non puoi unirti in questo momento."

    if from_button:
        await context.bot.send_message(chat_id, response_text)
    else:
        await update.message.reply_text(response_text)
```

L’inizio della partita viene gestito dalla classe game, così come l’assegnazione dei ruoli.

```
async def startGame(self, context=None):
    if self.state == GameState.WAITING and len(self.players) >= config.MIN_GIOCATORI:
        await self.assignRoles()
        self.playersAlive = set(self.players.keys())
        self.state = GameState.NIGHT
        await self.sendMessage(config.MSG_GIOCO_INIZIATO)
        await self.nightPhase()
        return True
    else:
        print(f"Errore Stato: {self.state}, Giocatori: {len(self.players)}")
        return False
```

I ruoli vengono assegnati casualmente, a partire dai ruoli speciali e poi dando ai restanti giocatori il ruolo base.

```
async def assignRoles(self):
    players_ids = list(self.players.keys())
    num_players = len(players_ids)
    numLupi = max(1, num_players // 4)
    numVeggenti = 1 if num_players >= 5 else 0

    roles = ([Role.LUPO] * numLupi +
             [Role.VEGGENTE] * numVeggenti +
             [Role.VILLICO] * (num_players - numLupi - numVeggenti))
    random.shuffle(players_ids)
    random.shuffle(roles)

    wolf_ids = []

    for uid, role in zip(players_ids, roles):
        username = self.players[uid]['username']
        if role == Role.LUPO:
            self.players[uid] = Lupo(uid, username, self)
            wolf_ids.append(uid)
        elif role == Role.VEGGENTE:
            self.players[uid] = Veggente(uid, username, self)
        else:
            self.players[uid] = Villico(uid, username, self)
```

Durante la notte le classi dei ruoli speciali gestiscono le azioni notturne, i lupi e il veggente selezionano il target della mossa attraverso bottoni.

```
async def nightAction(self, bot, context: ContextTypes.DEFAULT_TYPE):
    targets = [
        player for player in self.game.players.values()
        if player.user_id != self.user_id and player.role != Role.LUPO and player.user_id in self.game.playersAlive
    ]
    if not targets:
        await bot.send_message(self.user_id, "Non ci sono bersagli validi per attaccare.")
        return None

    keyboard = [
        [InlineKeyboardButton(target.username, callback_data=f"kill_{target.user_id}")]
        for target in targets
    ]
    reply_markup = InlineKeyboardMarkup(keyboard)
    await bot.send_message( self.user_id,
        "Scegli un giocatore da eliminare:",
        reply_markup=reply_markup)
```

```
async def nightAction(self, bot, context: ContextTypes.DEFAULT_TYPE):
    buttons = [
        [InlineKeyboardButton(player.username, callback_data=f"see_{pid}")]
        for pid, player in self.game.players.items() if pid != self.user_id and pid in self.game.playersAlive
    ]
    reply_markup = InlineKeyboardMarkup(buttons)
    await bot.send_message(
        self.user_id,
        "Scegli un giocatore da controllare per scoprire il suo ruolo:",
        reply_markup=reply_markup
    )
```

La classe game gestisce la fase diurna, dove vengono eliminati i giocatori e viene rivelato al veggente il ruolo del giocatore selezionato.

```
async def dayPhase(self):
    self.state = GameState.DISCUSSING

    # Night actions
    if self.wolf_kills:
        counter = Counter(self.wolf_kills)
        sorted_victims = sorted(counter.items(), key=lambda x: x[1], reverse=True)
        victim_id, _ = sorted_victims[0]
        await self.eliminatePlayer(victim_id)
        victim = self.players.get(victim_id)
        await self.sendMessage(f"@{victim.username} e' stato mangiato dai lupi! Era un {victim.role}.")
        self.wolf_kills = []
    else:
        await self.sendMessage("Questa notte i lupi sono rimasti a digiuno.")

    for seer_id, target_id in self.seer_vision.items():
        seer = self.players.get(seer_id)
        target = self.players.get(target_id)
        if seer and target:
            if seer.user_id not in self.playersAlive:
                await self.application.bot.send_message(seer.user_id, f"Sei stato eliminato prima di poter vedere il ruolo di @{target.username}")
            else:
                await self.application.bot.send_message(seer.user_id, f"Hai avuto una visione: @{target.username} e' un {target.role}.")
    self.seer_vision.clear()

    if self.checkGameOver():
        await self.endGame()
    else:
        await self.sendMessage(config.MSG_GIORNO)
        self.startTimer(self.votingPhase, config.TEMPO_DISCUSSIONE)
```

Ogni volta che viene eliminato un giocatore, come ad esempio dopo che un lupo mangia un giocatore, viene controllato lo stato del gioco. Funzione presente all’interno della classe game.

```
def checkGameOver(self):
    wolves_count = sum(1 for pid in self.playersAlive if self.players[pid].role == Role.LUPO)
    villagers_count = len(self.playersAlive) - wolves_count
    return wolves_count == 0 or wolves_count >= villagers_count
```

Se la condizione è falsa si continua con la discussione e poi con i voti per eliminare i giocatori sospetti. La classe game manda a tutti i giocatori una chiamata alla loro funzione doVote presente nella classe player.

```
async def doVote(self, bot):
    buttons = [
        [InlineKeyboardButton(f"{self.game.players[pid].username}", callback_data=f"vote_{pid}")]
        for pid in self.game.playersAlive if pid != self.user_id
    ]
    reply_markup = InlineKeyboardMarkup(buttons)
    sent_msg = await bot.send_message(self.user_id, "Scegli chi eliminare:", reply_markup=reply_markup)
    self.vote_message_id = sent_msg.message_id
```

Infine, la classe game gestisce l'eliminazione del giocatore più votato e ricontrolla la condizione di fine gioco. Se non si verifica, riparte la fase notturna.

```
async def endVotingPhase(self, context=None):
    print("Voting Phase ended")
    for player_id in self.playersAlive:
        if player_id not in self.votes:
            player = self.players[player_id]
            if hasattr(player, 'vote_message_id'):
                try:
                    await self.application.bot.edit_message_text(
                        "Non hai fatto in tempo a votare",
                        chat_id=player.user_id,
                        message_id=player.vote_message_id
                    )
                except Exception as e:
                    print(f"Error editing vote message for {player.user_id}: {e}")

    if not self.votes:
        await self.sendMessage("Nessun voto espresso. Nessuno viene eliminato.")
        await self.nightPhase()
        return
    vote_counts = {}
    for voted_id in self.votes.values():
        vote_counts[voted_id] = vote_counts.get(voted_id, 0) + 1

    most_voted = max(vote_counts, key=vote_counts.get)
    max_votes = vote_counts[most_voted]
    tied_players = [pid for pid, votes in vote_counts.items() if votes == max_votes]

    if len(tied_players) > 1:
        await self.sendMessage("Pareggio nei voti. Nessuno viene eliminato.")
    else:
        victim = self.players.get(most_voted)
        if victim:
            await self.eliminatePlayer(most_voted)
            await self.sendMessage(f"@{victim.username} e' stato eliminato per decisione del villaggio! Era un {victim.role}")

    if self.checkGameOver():
        await self.endGame()
    else:
        await self.nightPhase()
```

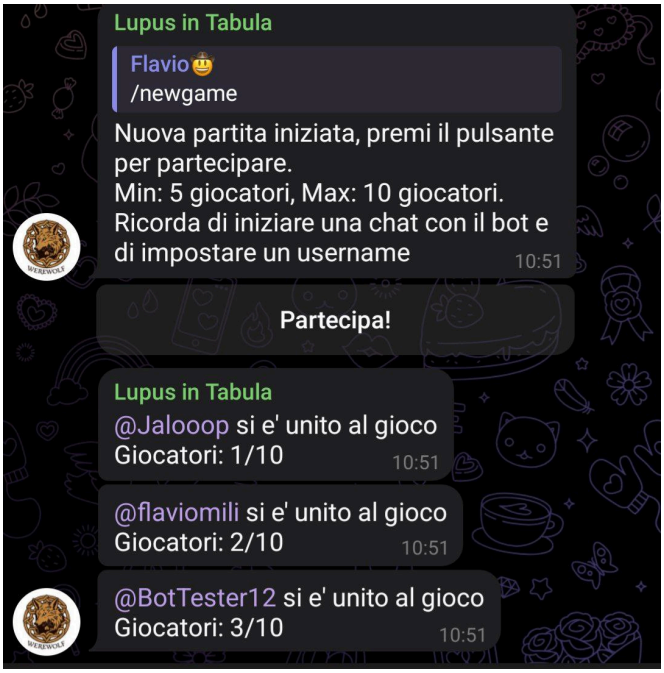
Il gioco prosegue finché non ci sono più lupi o il numero di lupi è pari a quello degli altri giocatori.

Appena viene determinato un vincitore, viene mandato al gruppo un messaggio con il risultato e il gioco finisce.

A questo punto è possibile cominciare una nuova partita.

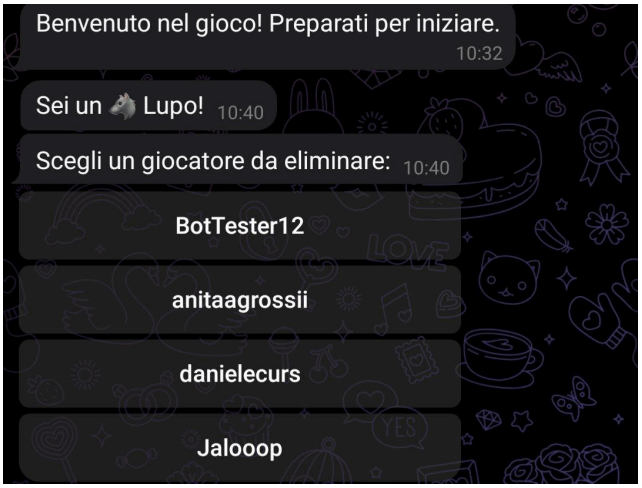
Simulazione di una partita

La partita inizia quando viene scritto
`/newgame` in chat, a quel unto i giocatori possono entrare in partita clickando su partecipa o scrivendo `/join`.

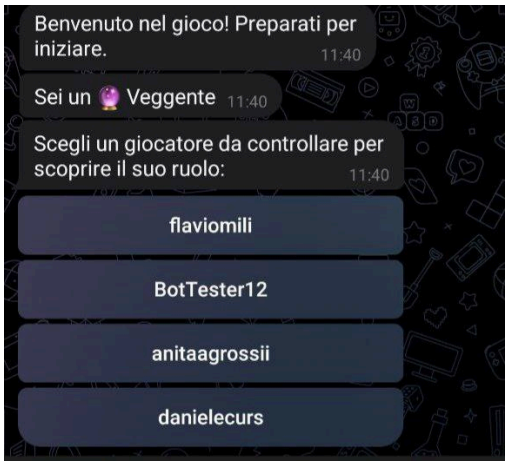


l gioco inizia e a tutti i giocatori viene assegnato un ruolo, I lupi e i veggenti possono effettuare la loro mossa speciale.

Lupo: decide un giocatore da mangiare e che verrà eliminato dal gioco.

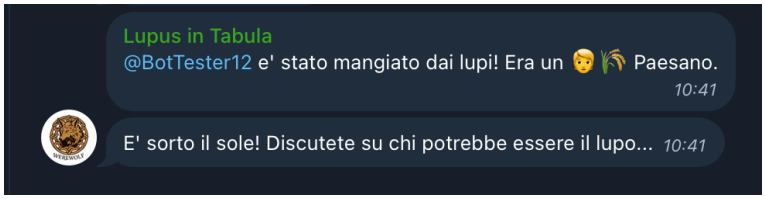


Veggente: decide un giocatore di cui scoprire il ruolo

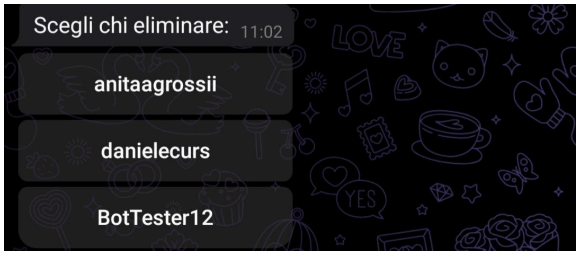


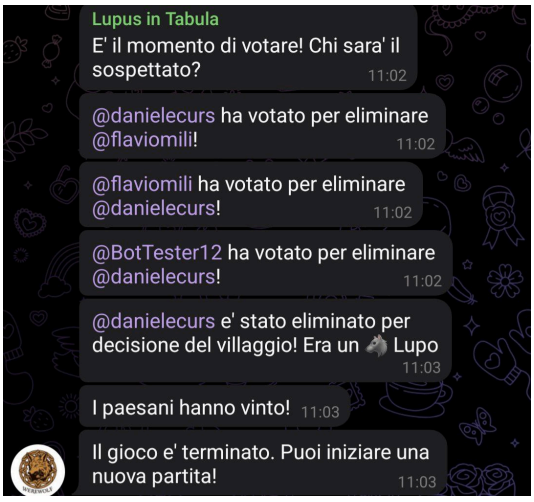
I contadini invece non hanno una mossa notturna quindi il loro obbiettivo è semplicemente rimanere vivi e votare per l'eliminazione del lupo durante il giorno.

Appena arriva il giorno viene eliminato il giocatore selezionato dal lupo e il veggente vede il ruolo del giocatore scelto.



Alla fine del giorno si sceglie di votare per l'eliminazione di un giocatore sospetto.





Nel caso di questa foto il lupo era un solo giocatore quindi appena viene eliminato la vittoria viene assegnata ai paesani.

Possibili aggiunte future

Il bot è funzionante ed è possibile fare delle partite complete però potrebbe giovare da alcune aggiunte:

- Nuovi ruoli speciali che permettono di avere un’esperienza di gioco più varia.
- Gestione impostazioni tramite il bot, direttamente in chat con gli admin che decidono come impostare i timer.
- Traduzione automatica del gioco in diverse lingue

Conclusioni

Lo sviluppo di questo bot Telegram ha rappresentato un'importante esperienza pratica nell'ingegneria del software, permettendo di affrontare problematiche reali legate alla gestione della concorrenza, alla comunicazione tra utenti e all'automazione del flusso di gioco. L'uso della libreria `python-telegram-bot` ha facilitato l'integrazione con l'API di Telegram, mentre l'architettura modulare ha reso il codice più leggibile e scalabile.

Durante il progetto, ho approfondito concetti fondamentali come la programmazione asincrona in python, la gestione degli eventi e il design di software object-oriented. Inoltre, il lavoro di sviluppo ha evidenziato l'importanza del debugging e del testing in un contesto in cui le interazioni tra utenti possono generare scenari complessi e difficili da prevedere.

Questa esperienza ha fornito competenze utili non solo per la creazione di chatbot, ma più in generale per la progettazione e l’implementazione di sistemi software interattivi, consolidando un approccio metodico alla risoluzione dei problemi e allo sviluppo iterativo del codice.