```python
import random
import yaml
import time
from typing import List, Dict, Optional, Any, Callable, Union
from pathlib import Path
from enum import Enum
import os
from datetime import datetime


from litellm import completion
from yaml import safe_load
from abc import ABC, abstractmethod
from dataclasses import dataclass, field

BASE = Path(__file__).resolve().parent.parent.parent
SRC_DIR = BASE / "src"
import sys

sys.path.append(str(BASE))

from src.Logger import GameLogger


# ---------------------------------------------------------------------------
# Core Engine Structures (DSL Support)
# ---------------------------------------------------------------------------


@dataclass
class ActionContext:
    game: Any  # "WerewolfGame"
    player: Optional[Any] = None  # "Player"
    target: Optional[str] = None
    extra_data: Dict[str, Any] = field(default_factory=dict)


class GameAction(ABC):
    """Abstract base class for a game action defined in DSL."""

    @abstractmethod
    def execute(self, context: ActionContext) -> Any:
        pass

    @abstractmethod
    def description(self) -> str:
        pass


@dataclass
class GameStep:
    """A single step in a game phase (e.g. 'Werewolves wake up')."""

    name: str
    roles_involved: List[Any]  # List['Role']
    action: GameAction
    condition: Optional[Callable[[Any], bool]] = None  # Condition to run this
step


@dataclass
class GamePhase:
    """A game phase consisting of multiple steps (e.g. 'Night', 'Day')."""

    name: str
    steps: List[GameStep] = field(default_factory=list)

    def add_step(self, step: GameStep):
        self.steps.append(step)


# models.py
class Role(Enum):
    WEREWOLF = "werewolf"
    VILLAGER = "villager"
    SEER = "seer"
    WITCH = "witch"
    HUNTER = "hunter"
    GUARD = "guard"


class DeathReason(Enum):
    KILLED_BY_WEREWOLF = "在夜晚被杀害"
    POISONED_BY_WITCH = "被女巫毒杀"
    VOTED_OUT = "被投票出局"
    SHOT_BY_HUNTER = "被猎人带走"


# ---------------------------------------------------------------------------
# Concrete Actions Implementation
# ---------------------------------------------------------------------------


class NightStartAction(GameAction):
    def description(self) -> str:
        return "入夜初始化"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        game.day_number += 1
        print(f"#@ 第 {game.day_number} 天夜晚降临")
        game.logger.log_event(f"第 {game.day_number} 天夜晚降临",
game.all_player_names)
        game.killed_player = None
        for p in game.players.values():
            p.is_guarded = False


class DayStartAction(GameAction):
    def description(self) -> str:
        return "天亮初始化"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        print("#: 天亮了.")
        print(f"#@ 现在是第 {game.day_number} 天白天")
        game.logger.log_event(
            f"天亮了. 现在是第 {game.day_number} 天白天", game.all_player_names
        )

        if game.killed_player:
            game.handle_death(game.killed_player, DeathReason.KILLED_BY_WEREWOLF)
        else:
            print("#@ 今晚是平安夜")
            game.logger.log_event("今晚是平安夜", game.all_player_names)


class GuardAction(GameAction):
    def description(self) -> str:
        return "守卫守护"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        guard = game._get_player_by_role(Role.GUARD)
        if not guard:
            return

        print("#@ 守卫请睁眼")
        prompt = "守卫, 请选择你要守护的玩家 (不能连续两晚守护同一个人): "
        game.logger.log_event("守卫请睁眼" + prompt, [guard.name])

        alive_players = game._get_alive_players()
        valid_targets = [p for p in alive_players if p != game.last_guarded]

        target = prompt_for_choice(guard, prompt, valid_targets)

        game.players[target].is_guarded = True
        game.last_guarded = target

        print("#@ 守卫请闭眼")
        game.logger.log_event(f"你守护了 {target}", [guard.name])
        game.logger.log_event("守卫行动了", game.all_player_names)


class WerewolfNightAction(GameAction):
    def description(self) -> str:
        return "狼人夜间行动"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        werewolves = game._get_alive_players([Role.WEREWOLF])
        if not werewolves:
            return

        print("#@ 狼人请睁眼")
        print(f"#@ 现在的狼人有: {', '.join(werewolves)}")
        game.logger.log_event(
            "狼人请睁眼" + f"现在的狼人有: {', '.join(werewolves)}",
            werewolves,
        )

        if len(werewolves) == 1:
            print("#@ 独狼无需讨论，直接进入投票阶段")
            game.logger.log_event("独狼无需讨论，直接进入投票阶段", werewolves)
        else:
            self._handle_discussion(game, werewolves)

        self._handle_voting(game, werewolves)

        print(f"#@ 狼人请闭眼")
        if game.killed_player:
            game.logger.log_event(f"你击杀了 {game.killed_player}", werewolves)
        game.logger.log_event("狼人行动了", game.all_player_names)

    def _handle_discussion(self, game, werewolves):
        print("#@ 狼人请开始讨论")
        print("#@ 请轮流发言, 输入 '0' 表示准备好投票")
        game.logger.log_event(
            "狼人请开始讨论, 输入 '0' 表示发言结束, 准备投票",
            werewolves,
        )
        ready_to_vote = set()
        discussion_rounds = 0
        max_discussion_rounds = 5

        while (
            len(ready_to_vote) < len(werewolves)
            and discussion_rounds < max_discussion_rounds
        ):
            discussion_rounds += 1
            for wolf in werewolves:
                if wolf in ready_to_vote:
                    continue
                wolf_player = game.players[wolf]
                action = wolf_player.speak(f"{wolf}, 请发言或输入 '0' 准备投票: ")

                if action == "0":
                    ready_to_vote.add(wolf)
                    msg = (
                        f"({wolf} 已准备好投票
{len(ready_to_vote)}/{len(werewolves)})"
                    )
                    print(f"#@ {msg}")
                    game.logger.log_event(msg, werewolves)
                elif action:
                    print(f"#: [狼人频道] {wolf} 发言: {action}")
                    game.logger.log_event(
                        f"[狼人频道] {wolf} 发言: {action}", werewolves
                    )

        if discussion_rounds >= max_discussion_rounds and len(ready_to_vote) <
len(
                werewolves
            ):
                msg = f"讨论已达到最大轮次({max_discussion_rounds}轮)，强制进入投票阶段"
                print(f"#@ {msg}")
                game.logger.log_event(msg, werewolves)

    def _handle_voting(self, game, werewolves):
        alive_players = game._get_alive_players()
        print("#@ 狼人请投票")
        game.logger.log_event("狼人请投票", werewolves)

        while True:
            votes = {name: 0 for name in alive_players}
            for wolf_name in werewolves:
                wolf_player = game.players[wolf_name]
                prompt = f"{wolf_name}, 请投票选择要击杀的目标: "
                target = prompt_for_choice(wolf_player, prompt, alive_players)
                votes[target] += 1

            max_votes = 0
            kill_targets = []
            if votes:
                max_votes = max(votes.values())
                if max_votes > 0:
                    kill_targets = [
                        name for name, count in votes.items() if count ==
max_votes
                    ]

            if len(kill_targets) == 1:
                game.killed_player = kill_targets[0]
                print(f"#@ 狼人达成一致, 选择了击杀 {game.killed_player}")
                game.logger.log_event(
                    f"狼人投票决定击杀 {game.killed_player}", werewolves
                )
                break
            else:
                print("#@ 狼人投票出现平票, 请重新商议并投票")
                game.logger.log_event("狼人投票出现平票, 请重新商议并投票",
werewolves)


class SeerAction(GameAction):
    def description(self) -> str:
        return "预言家查验"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        seer = game._get_player_by_role(Role.SEER)
        if not seer:
            return

        print("#@ 预言家请睁眼")
        prompt = "预言家, 请选择要查验的玩家: "
        game.logger.log_event("预言家请睁眼. 请选择要你要查验的玩家: ", [seer.name])
        alive_players = game._get_alive_players()
        target = prompt_for_choice(seer, prompt, alive_players)

        role = game.players[target].role
        identity = "狼人" if role == Role.WEREWOLF.value else "好人"
        print(f"#@ 查验结果: {target} 的身份是 {identity}")

        print("#@ 预言家请闭眼")
        game.logger.log_event(
            f"你查验了 {target} 的身份, 结果为 {identity}", [seer.name]
        )
        game.logger.log_event("预言家行动了", game.all_player_names)


class WitchAction(GameAction):
    def description(self) -> str:
        return "女巫毒药与解药"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        witch = game._get_player_by_role(Role.WITCH)
        if not witch:
            return

        print("#@ 女巫请睁眼")
        game.logger.log_event("女巫请睁眼", [witch.name])

        alive_players = game._get_alive_players()
        actual_killed = None

        # Check death info
        if game.killed_player:
            if game.players[game.killed_player].is_guarded:
                print(f"#@ 今晚是个平安夜, {game.killed_player} 被守护了")
                game.logger.log_event(
                    f"今晚是个平安夜, {game.killed_player} 被守护了",
                    game.all_player_names,
                )
            else:
                print(f"#@ 今晚 {game.killed_player} 被杀害了")
                game.logger.log_event(
                    f"今晚 {game.killed_player} 被杀害了", [witch.name]
                )
                actual_killed = game.killed_player

        # Save Potion
        if not game.witch_save_used and actual_killed:
            prompt = "女巫, 你要使用解药吗? "
            if prompt_for_choice(witch, prompt, ["y", "n"]) == "y":
                actual_killed = None
                game.witch_save_used = True
                print(f"#@ 你使用解药救了 {game.killed_player}")
                game.logger.log_event(
                    f"你使用解药救了 {game.killed_player}", [witch.name]
                )
                game.logger.log_event("女巫使用了解药", game.all_player_names)
        # Poison Potion
        if not game.witch_poison_used:
            prompt = "女巫，你要使用毒药吗？ "
            if prompt_for_choice(witch, prompt, ["y", "n"]) == "y":
                poison_prompt = "请选择要毒杀的玩家： "
                target = prompt_for_choice(witch, poison_prompt, alive_players)
                if actual_killed is None:
                    actual_killed = target
                else:
                    # If someone was already killed and not saved, we have a
second death.
                    # The original logic handles this by calling handle_death
immediately or later.
                    # Here we might need to queue it or handle it.
                    # Current logic: handle_death(target) immediately.
                    game.handle_death(target, DeathReason.POISONED_BY_WITCH)

                game.witch_poison_used = True
                print(f"#@ 你使用毒药毒了 {target} ")
                game.logger.log_event(f"你使用毒药毒了 {target}", [witch.name])
                game.logger.log_event("女巫使用了毒药", game.all_player_names)

        # Update killed player to the final result (if saved, it's None)
        # Note: If poison was used on a second target, handle_death was called.
        # If poison was used as the ONLY death (because save was used),
actual_killed becomes target.
        game.killed_player = actual_killed

        print("#@ 女巫请闭眼")
        game.logger.log_event("女巫行动了", game.all_player_names)


class DayDiscussionAction(GameAction):
    def description(self) -> str:
        return "白天讨论"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        alive_players = game._get_alive_players()
        print(f"#@ 场上存活的玩家: {', '.join(alive_players)}")
        game.logger.log_event(
            f"场上存活的玩家: {', '.join(alive_players)}", game.all_player_names
        )

        for player_name in alive_players:
            player = game.players[player_name]
            speech = player.speak(f"{player_name}, 请发言: ")
            print(f"#: {player_name} 发言: {speech}")
            game.logger.log_event(
                f"{player_name} 发言: {speech}", game.all_player_names
            )


class DayVoteAction(GameAction):
    def description(self) -> str:
        return "白天投票"

    def execute(self, context: ActionContext) -> Any:
        game = context.game
        alive_players = game._get_alive_players()
        print("#@ 请开始投票")
        game.logger.log_event("请开始投票", game.all_player_names)

        votes = {name: 0 for name in alive_players}
        for voter_name in alive_players:
            voter = game.players[voter_name]
            prompt = f"{voter_name}, 请投票: "
            target = prompt_for_choice(voter, prompt, alive_players)
            votes[target] += 1
            print(f"#: {voter_name} 投票给 {target}")
            game.logger.log_event(
                f"{voter_name} 投票给 {target}", game.all_player_names
            )

        max_votes = max(votes.values())
        voted_out_players = [
            name for name, count in votes.items() if count == max_votes
        ]

        if len(voted_out_players) == 1:
            voted_out_player = voted_out_players[0]
            print(f"#! 投票结果: {voted_out_player} 被投票出局")
            game.logger.log_event(
                f"投票结果: {voted_out_player} 被投票出局",
                game.all_player_names,
            )
            game.handle_death(voted_out_player, DeathReason.VOTED_OUT)
        else:
            print("#@ 投票平票, 无人出局")
            game.logger.log_event(
                "投票平票, 无人出局",
                game.all_player_names,
            )


# player.py
with open("config.yaml", "r", encoding="utf-8") as f:
    config = safe_load(f)
    DEBUG = config.get("debug", True)

PROMPT = """
你正在一局狼人杀游戏中, 你的名字是{self.name}, 身份是{self.role}. 请认真参与游戏, 努力帮助自己的阵营获胜.

**基本原则:**
1.  **隐藏身份:** 不要直接或间接透露你的身份, 尤其是你的特殊能力.
2.  **分析局势:** 根据发言和投票, 判断其他玩家的身份和阵营.
3.  **有效沟通:** 你的发言和投票是影响游戏走向的关键. 请表达清晰, 有逻辑. 使用<br>标签换行, <strong></strong>标签组加粗, 其余一律不允许使用.
4.  **避免重复:** 每次发言都要提供新的信息或观点, 绝对不要重复之前已经说过的话. 仔细阅读游戏记录, 确保你的发言是独特和有价值的.
```

```
5.  **避免幻想:** **不要编造游戏开始前的场外信息或虚构的背景故事**. 你的发言应该基于当前游戏进程中发生的事件. 游戏内的策略性欺诈(如狼人伪装身份、平民虚报身份等)是允许的, 但不要创造不存在的游戏外情节.
6.  **注意角色配置:** **严格按照本场游戏的角色配置发言, 不要提及本场不存在的角色**. 仔细确认当前游戏中实际存在哪些角色, 避免谈论不存在的神职或玩家.
7.  **锚定自我身份:** **始终以你自己的身份和立场发言, 不要跑到其他玩家的立场上去思考或发言**. 你是{self.name}, 身份是{self.role}, 请严格从这个角度参与游戏.

**角色专属指南:**
*   **如果你是狼人:**
    *   **夜晚讨论:** 和你的狼同伴积极讨论, 统一击杀目标. 优先选择看起来像神职的玩家(如预言家、女巫). 如果无法确定, 可以选择发言较好或者被怀疑的玩家, 混淆视听. **重要: 每次发言都要提出新的观点或信息, 不要重复之前说过的话. 当你们讨论完毕并达成一致意见后, 必须输入'0'来结束讨论进入投票阶段, 不要无休止地讨论下去. **
    *   **战术:** 可以选择悍跳(假装自己是预言家)、倒钩(站边一个真正的预言家以获取信任)或者深潜(隐藏在好人中).
    *   **发言:** 伪装成好人, 误导好人阵营的判断. 可以通过踩其他玩家、聊心态、或者提出自己的逻辑来建立好人形象. **避免重复之前的发言内容. **
*   **如果你是预言家:**
    *   **验人:** 每天查验一个你怀疑是狼人的玩家. 你的验人结果对好人至关重要.
    *   **跳身份:** 在合适的时机(通常是第一天)跳出身份, 报出你的验人信息, 带领好人投票.
    *   **策略建议:** 可以根据游戏策略需要调整报告内容, 但避免编造游戏外的虚假背景信息.
*   **如果你是女巫:**
    *   **用药:** 你的解药和毒药非常宝贵. 解药通常在第一晚救下被杀的玩家. 毒药要谨慎使用, 最好在你确认一个玩家是狼人时再用.
    *   **策略建议:** 可以根据游戏策略需要调整发言内容, 但避免编造游戏外的虚假背景信息.
*   **如果你是猎人:**
    *   **威慑:** 你可以在发言中适当表现强势, 让狼人不敢轻易攻击你.
    *   **开枪:** 当你被投票出局或被狼人杀死时, 你可以带走场上任意一个玩家. 请根据局势, 射杀你认为是狼人的玩家.
*   **如果你是守卫:**
    *   **守护:** 每晚选择一个玩家进行守护, 不能连续两晚守护同一人.
    *   **策略建议:** 可以根据游戏策略需要调整发言内容, 但避免编造游戏外的虚假背景信息.
*   **如果你是平民:**
    *   **逻辑:** 仔细听发言, 找出逻辑漏洞. 你的目标是帮助神职玩家找出所有狼人.
    *   **站边:** 相信你认为是预言家的玩家, 并跟随他投票.

**行动指令:**
接下来, 你会收到法官的指令. 请严格按照指令要求行动.
*   **如果是选择:** 从给出的选项中选择一项, **仅返回选项的名称**, 不要添加任何解释、序号或其他文字.
*   **如果是发言:** **仅输出你想要说的话**, 不要包含你的名字、身份或任何解释.
"""


REMINDER = """
重要提醒: 请根据当前情况发表新的、有意义的观点, 避免重复之前的发言内容.
**关键警告: 不要编造游戏开始前的场外信息或虚构的背景故事. 你的发言应该基于当前游戏进程, 游戏内的策略性欺诈是允许的. **
**角色配置提醒: 严格按照本场游戏的角色配置发言, 不要提及本场不存在的角色. **
**身份锚定: 你是{0}, 身份是{1}, 请严格从这个角度发言, 不要跑到其他玩家的立场上. **
"""


REMINDER_WEREWOLF = """
**特别注意**: 如果你是狼人且正在夜晚讨论阶段, 当你们已经充分讨论并达成一致意见时, 请**仅回答**'0'来结束讨论进入投票阶段. **不要无休止地重复讨论**.
"""
```

```
REMINDER_FIRST_NIGHT = """
**现在是第一晚** , 游戏刚刚开始, 因此你不应该讨论例如"先前的发言等"场外信息.
"""


class Player:
    def __init__(self, name: str, role: str):
        self.name = name
        self.role = role
        self.prompt = PROMPT.format(self=self)
        self.is_alive = True
        self.is_guarded = False
        self.config = None
        self.bind_config()
        self.is_human = self.config["human"]
        self.game_logger = None
        self.is_first_night = True

    def bind_config(self):
        with open("config.yaml", "r", encoding="utf-8") as f:
            config = safe_load(f)
            for player in config["players"]:
                if player["name"] == self.name:
                    self.config = player
                    break

    def set_logger(self, logger):
        self.game_logger = logger

    def call_ai_response(self, prompt_text: str, valid_choices: List[str]):
        if DEBUG:
            return random.choice(valid_choices)

        history = []
        if self.game_logger:
            log_file = self.game_logger._get_player_log_file(self.name)
            if os.path.exists(log_file):
                with open(log_file, "r", encoding="utf-8") as f:
                    log_content = f.read()
                    context_reminder = REMINDER.format(self.name, self.role)
                    if (
                        "请发言或输入 '0' 准备投票" in prompt_text
                        and self.role == "werewolf"
                    ):
                        context_reminder += REMINDER_WEREWOLF
                        if self.is_first_night:
                            self.is_first_night = False
                            context_reminder += REMINDER_FIRST_NIGHT
                    history.append(
                        {
                            "role": "system",
                            "content": f"本场全部游戏记录:
\n{log_content}\n\n{context_reminder}",
                        }
                    )
        history.append({"role": "system", "content": self.prompt})
        prompt = f"{prompt_text}\n请从以下选项中选择: {', '.join(valid_choices)}"
        history.append({"role": "user", "content": prompt})

        response = completion(
            model=self.config["model"],
            messages=history,
            stream=False,
        )
        ai_choice = response.choices[0].message.content
        for choice in valid_choices:
            if choice in ai_choice:
                return choice
        return random.choice(valid_choices)

    def call_human_response(
        self, prompt_text: str, valid_choices: List[str], allow_skip: bool =
False
    ):
        while True:
            print(prompt_text)
            display_choices = list(valid_choices)
            if allow_skip:
                display_choices.append("skip")

            for i, choice in enumerate(display_choices):
                print(f"[yellow]{i + 1}[/yellow]. [cyan]{choice}[/cyan]")

            player_input = input("> ").strip()
            player_input_lower = player_input.lower()

            if player_input.isdigit():
                choice_index = int(player_input) - 1
                if 0 <= choice_index < len(display_choices):
                    selected_choice = display_choices[choice_index]
                    return selected_choice

            if allow_skip and player_input_lower == "skip":
                return "skip"

            for choice in valid_choices:
                if choice.lower() == player_input_lower:
                    return choice

            print("[bold red]无效的选择, 请重新输入. [/bold red]")

    def call_ai_speak(self, prompt_text: str):
        if DEBUG:
            return "ai_response<br>, and <strong>ai_response</strong>"
        history = []
        if self.game_logger:
            log_file = self.game_logger._get_player_log_file(self.name)
            if os.path.exists(log_file):
                with open(log_file, "r", encoding="utf-8") as f:
                    log_content = f.read()
                    if log_content.strip():
                        context_reminder = REMINDER.format(self.name, self.role)
                        if (
                            "请发言或输入 '0' 准备投票" in prompt_text
                            and self.role == "Werewolf"
                        ):
                            context_reminder += REMINDER_WEREWOLF
                        context_prompt = (
                            f"游戏记录:\n{log_content}\n\n{context_reminder}"
                        )
                        history.append({"role": "system", "content":
context_prompt})

        history.append({"role": "system", "content": self.prompt})
        prompt = f"{prompt_text}"
        history.append({"role": "user", "content": prompt})

        print(f"{self.name} 正在思考...")
        response = completion(
            model=self.config["model"],
            messages=history,
            stream=False,
        )
        speech = response.choices[0].message.content
        return speech

    def call_human_speak(self, prompt_text: str):
        return input(prompt_text)

    def speak(self, prompt_text: str):
        if self.is_human:
            return self.call_human_speak(prompt_text)
        else:
            return self.call_ai_speak(prompt_text)


# ui.py
def prompt_for_choice(
    player: "Player",
    prompt_text: str,
    valid_choices: List[str],
    allow_skip: bool = False,
) -> str:
    if player.is_human:
        return player.call_human_response(prompt_text, valid_choices, allow_skip)
    else:
        return player.call_ai_response(prompt_text, valid_choices)


# Game.py
class WerewolfGame:
    def __init__(self, players: List[Dict[str, str]]):
        self.players: Dict[str, Player] = {}
        self.roles: Dict[str, int] = {}
        self.all_player_names: List[str] = [p.get("player_name", "") for p in
players]
```

```
        self.killed_player: Optional[str] = None
        self.last_guarded: Optional[str] = None
        self.witch_save_used = False
        self.witch_poison_used = False
        self.day_number = 0
        self._players = players
        self.logger = GameLogger("werewolf", self._players)
        self.phases: List[GamePhase] = []
        self._init_phases()

    def _init_phases(self):
        # Night Phase
        night = GamePhase("Night")
        night.add_step(GameStep("NightStart", [], NightStartAction()))
        night.add_step(GameStep("Guard", [Role.GUARD], GuardAction()))
        night.add_step(GameStep("Werewolf", [Role.WEREWOLF],
WerewolfNightAction()))
        night.add_step(GameStep("Seer", [Role.SEER], SeerAction()))
        night.add_step(GameStep("Witch", [Role.WITCH], WitchAction()))
        self.phases.append(night)

        # Day Phase
        day = GamePhase("Day")
        day.add_step(GameStep("DayStart", [], DayStartAction()))
        day.add_step(GameStep("Discussion", [], DayDiscussionAction()))
        day.add_step(GameStep("Vote", [], DayVoteAction()))
        self.phases.append(day)

    def run_phase(self, phase: GamePhase):
        for step in phase.steps:
            if self.check_game_over():
                return

            # Optional: Check if roles for this step exist in the game to skip
early
            # For now, we rely on actions checking themselves or existing logic.

            context = ActionContext(game=self)
            step.action.execute(context)

    def _get_alive_players(self, roles: Optional[List[Role]] = None) ->
List[str]:
        alive_players = []
        role_values = []
        if roles:
            for r in roles:
                role_values.append(r.value)

        for name, p in self.players.items():
            if p.is_alive:
                if roles is None or p.role in role_values:
                    alive_players.append(name)
        return alive_players

    def _get_player_by_role(self, role: Role) -> Optional[Player]:
        for p in self.players.values():
            if p.role == role.value and p.is_alive:
                return p
        return None

    def _cancel(self):
        self.logger.log_event(
            f"游戏已取消.",
            self.all_player_names,
        )
        self.players.clear()
        self.roles.clear()
        self.all_player_names.clear()
        self.killed_player = None
        self.last_guarded = None
        self.witch_save_used = False
        self.witch_poison_used = False
        self.day_number = 0

    def setup_game(self):
        try:
            with open("config.yaml", "r", encoding="utf-8") as file:
                config = yaml.safe_load(file)
                player_count = len(config["players"])
                player_names = [player["name"] for player in config["players"]]
                self.all_player_names = player_names

        except (FileNotFoundError, KeyError, ValueError):
            print("#! 配置文件有错, 请检查配置文件.")

        except Exception as e:
            print(f"#! 未知错误: {str(e)}")

        if player_count == 6:
            self.roles = {
                Role.WEREWOLF.value: 2,
                Role.VILLAGER.value: 2,
                Role.SEER.value: 1,
                Role.WITCH.value: 1,
            }

        else:
            self.roles = {
                Role.WEREWOLF.value: max(1, player_count // 4),
                Role.SEER.value: 1,
                Role.WITCH.value: 1,
                Role.HUNTER.value: 1,
                Role.GUARD.value: 1,
            }
            self.roles[Role.VILLAGER.value] = player_count -
sum(self.roles.values())

        print("#@ 本局游戏角色配置")
        role_config = []
        for role, count in self.roles.items():
            if count > 0:
                role_config.append(f"{role.capitalize()} {count}人")
        print("#@ " + ", ".join(role_config))

        self.logger.log_event(
            f"本局玩家人数: {player_count}",
            self.all_player_names,
        )
        self.logger.log_event(
            f"角色卡配置: {role_config}",
            self.all_player_names,
        )
        self.logger.log_event(
            f"玩家 {player_names} 已加入游戏.",
            self.all_player_names,
        )

        role_list = []
        for role, count in self.roles.items():
            for _ in range(count):
                role_list.append(role)
        random.shuffle(role_list)

        for name, role in zip(player_names, role_list):
            player = Player(name, role)
            player.set_logger(self.logger)
            self.players[name] = player

        werewolves = self._get_alive_players([Role.WEREWOLF])

        print("#@ 角色分配完成, 正在分发身份牌...")
        for name, player in self.players.items():
            time.sleep(0.3)
            print(f"\n{name}, 你的身份是: {player.role.capitalize()}")
            if player.role == Role.WEREWOLF.value:
                teammates = []
                for w in werewolves:
                    if w != name:
                        teammates.append(w)
                if teammates:
                    print(f"你的狼人同伴是: {', '.join(teammates)}")
                else:
                    print("你是唯一的狼人")

        self.logger.log_event(
            "角色分配完成, 正在分发身份牌...",
            self.all_player_names,
        )

        for name, player in self.players.items():
            self.logger.log_event(
                f"你的身份是: {player.role.capitalize()}",
                [player.name],
            )

            if player.role == Role.WEREWOLF.value:
                teammates = []
                for w in werewolves:
```

```python
                if w != name:
                    teammates.append(w)
            if teammates:
                self.logger.log_event(
                    f"你的狼人同伴是: {', '.join(teammates)}",
                    [player.name],
                )
            else:
                self.logger.log_event(
                    "你是唯一的狼人",
                    [player.name],
                )

        print("#: 游戏开始. 天黑, 请闭眼.")
        self.logger.log_event(
            "游戏开始. 天黑, 请闭眼.",
            self.all_player_names,
        )

    def handle_death(self, player_name: str, reason: DeathReason):
        if player_name and self.players[player_name].is_alive:
            self.players[player_name].is_alive = False
            print(f"#! {player_name} 死了, 原因是{reason.value}")
            self.logger.log_event(
                f"{player_name} 死了, 原因是 {reason.value}",
                self.all_player_names,
            )

            is_first_night_death = self.day_number == 1 and reason in [
                DeathReason.KILLED_BY_WEREWOLF,
                DeathReason.POISONED_BY_WITCH,
            ]
            can_have_last_words = (
                reason in [DeathReason.VOTED_OUT, DeathReason.SHOT_BY_HUNTER]
                or is_first_night_death
            )

            if can_have_last_words:
                player = self.players[player_name]
                last_words = player.speak(f"{player_name}, 请发表你的遗言: ")
                if last_words:
                    print(f"#: [遗言] {player_name} 发言: {last_words}")
                    self.logger.log_event(
                        f"[遗言] {player_name} 发言: {last_words}",
                        self.all_player_names,
                    )
                else:
                    print(f"#@ {player_name} 选择保持沉默, 没有留下遗言")
                    self.logger.log_event(
                        f"{player_name} 选择保持沉默, 没有留下遗言",
                        self.all_player_names,
                    )

            if self.players[player_name].role == Role.HUNTER.value:
                self.handle_hunter_shot(player_name)
    def handle_hunter_shot(self, hunter_name: str):
        print(f"#@ {hunter_name} 是猎人, 可以在临死前开枪带走一人")
        self.logger.log_event(
            f"{hunter_name} 是猎人, 可以在临死前开枪带走一人",
            self.all_player_names,
        )
        alive_players_for_shot = []
        for p in self._get_alive_players():
            if p != hunter_name:
                alive_players_for_shot.append(p)
        hunter_player = self.players[hunter_name]
        prompt = f"{hunter_name}, 请选择你要带走的玩家: "
        target = prompt_for_choice(
            hunter_player, prompt, alive_players_for_shot, allow_skip=True
        )

        if target == "skip":
            print("#@ 猎人放弃了开枪")
            self.logger.log_event(
                "猎人放弃了开枪",
                self.all_player_names,
            )
        else:
            self.logger.log_event(
                f"猎人 {hunter_name} 开枪带走了 {target}",
                self.all_player_names,
            )
            self.handle_death(target, DeathReason.SHOT_BY_HUNTER)

    def check_game_over(self):
        alive_werewolves = self._get_alive_players([Role.WEREWOLF])
        alive_villagers = self._get_alive_players(
            [Role.VILLAGER, Role.SEER, Role.WITCH, Role.HUNTER, Role.GUARD]
        )

        if not alive_werewolves:
            print("#! 游戏结束, 好人阵营胜利!")
            self.logger.log_event(
                "游戏结束, 好人阵营胜利!",
                self.all_player_names,
            )
            return True
        elif len(alive_werewolves) >= len(alive_villagers):
            print("#! 游戏结束, 狼人阵营胜利!")
            self.logger.log_event(
                "游戏结束, 狼人阵营胜利!",
                self.all_player_names,
            )
            return True
        return False

    def run_game(self):
        self.setup_game()
        while not self.check_game_over():
            for phase in self.phases:
                self.run_phase(phase)
                if self.check_game_over():
                    break


if __name__ == "__main__":
    # Load config to get players
    try:
        with open("config.yaml", "r", encoding="utf-8") as f:
            config_data = yaml.safe_load(f)
            # Construct players list for GameLogger
            # Assuming config has players with 'name'. UUID might be missing, so
we generate or use name.
            init_players = []
            for p in config_data.get("players", []):
                init_players.append(
                    {
                        "player_name": p["name"],
                        "player_uuid": p.get(
                            "uuid", p["name"]
                        ),  # Use name as uuid if missing
                    }
                )
    except Exception as e:
        print(f"Error loading config for main: {e}")
        init_players = []

    game = WerewolfGame(init_players)
    game.run_game()
```

我已经完成了 `e:\LudusEngine\.games\werewolf\werewolf.py` 的重构工作。现在，游戏逻辑已经被拆分为模块化的 `GameAction` (游戏动作)、`GameStep` (游戏步骤) 和 `GamePhase` (游戏阶段)。

这种结构完全支持你提到的"通过领域特定语言 (DSL) 生成游戏类"的目标。你可以将游戏流程看作是一个由 Phase 和 Step 组成的列表，每个 Step 执行一个具体的 Action。

主要改动总结

1. **引入 DSL 核心结构**:
   - `ActionContext`: 封装动作执行时的上下文（如游戏实例、当前玩家）。
   - `GameAction` (抽象基类): 定义所有游戏动作的接口 (`execute`, `description`)。
   - `GameStep`: 定义流程中的一步，包含名称、参与角色、对应的 Action 以及执行条件。
   - `GamePhase`: 定义游戏阶段（如"夜晚"、"白天"），包含多个 Step。
2. **逻辑模块化 (Action 实现)**:
   - 将原有的 `night_phase` 和 `day_phase` 中的逻辑拆分为独立的 Action 类:
     - `NightStartAction`: 入夜初始化。
     - `GuardAction`: 守卫行动。
     - `WerewolfNightAction`: 狼人夜间讨论与行动。
     - `SeerAction`: 预言家行动。
     - `WitchAction`: 女巫行动。

        - `DayStartAction`: 天亮结算与死亡处理。
        - `DayDiscussionAction`: 白天讨论环节。
        - `DayVoteAction`: 白天投票环节。
3. **重构** `WerewolfGame` **类**:
    - **移除** 了冗长的 `night_phase`、`day_phase` 和 `vote_phase` 方法。
    - **新增** `_init_phases()`: 使用 DSL 结构组装游戏流程（Night Phase 和 Day Phase）。
    - **更新** `run_game()`: 改为通用的流程执行循环，遍历 `phases` 并执行其中的 `steps`。

验证结果

我已经使用当前的配置文件（6人局，包含 AI 玩家）在 DEBUG 模式下运行了重构后的代码。游戏成功完成了从入夜、狼人杀人、女巫毒人到游戏结束（好人胜利）的完整流程，证明重构后的架构逻辑正确且运行流畅。

现在，你可以通过编写类似以下的 DSL 配置来生成不同的游戏变体，而无需修改核心代码：

```
# 伪代码示例：未来可以通过 DSL 生成此类配置
phases = [
    GamePhase("Night", steps=[
        GameStep("Werewolf", [Role.WEREWOLF], WerewolfNightAction()),
        GameStep("Seer", [Role.SEER], SeerAction())
    ]),
    GamePhase("Day", steps=[
        GameStep("Discussion", [], DayDiscussionAction()),
        GameStep("Vote", [], DayVoteAction())
    ])
]
```