
Werewolf
+ AI Storyteller
Group 167


OI
Specification

The Game

1

Set Up

4 - 8 Players + 1 Gamemaster, 1-2 Werewolves

2

Gameplay

Kill villagers + Vote players out, AI stories

3

Ending

No villagers or werewolves remain


02
API Usage

Story Generation

- GPT 4 - Powerful + Slow
- Generate Prompt → Save History → Call API → Parse JSON → Save History
- Maintain Conversation Context

```java
public class GPT4DataAccessObject implements ChatAPIAccessInterface {
    6 usages  yhz3 +1 *
    public String getResponse(String prompt) {
        String url = "https://api.openai.com/v1/chat/completions";
        String APIKEY = "enter api key here";
        String model = "gpt-4-1106-preview";

        try {
            URL obj = new URL(url);
            HttpURLConnection connection = (HttpURLConnection) obj.openConnection();
            connection.setRequestMethod("POST");
            connection.setRequestProperty("Authorization", "Bearer " + APIKEY);
            connection.setRequestProperty("Content-Type", "application/json");
```

```java
public String generatePlayerKilledPrompt(String playerKilled) {
    String preamble = "One night has passed and this player has been killed: ";
    String instruction = ". Describe in detail how they were killed. Reveal the victim's name only at the end of " +
            "the story. Keep it under 250 words.";
    String prompt = conversationHistory.getConversationHistory() + " User: " + preamble + playerKilled + instruction;
    // We don't want to add conversationHistory to the conversation history again, so we cannot just add prompt
    conversationHistory.addUserMessage(preamble + playerKilled + instruction);
    return prompt;
}
```

Context Compression

- GPT 3.5 - Cheap + Fast
- Save History: Compress History if length > 3
- ↓ Costs, ↑ Speed of Story Generation

```java
public class ConversationHistory {
    private final ArrayList<String> history;

    public ConversationHistory() { this.history = new ArrayList<String>(); }

    public void addGPTMessage(String message) { this.history.add("ChatGPT: " + message); }

    public void addUserMessage(String message) { this.history.add("User: " + message); }

    public String getConversationHistory() {
```

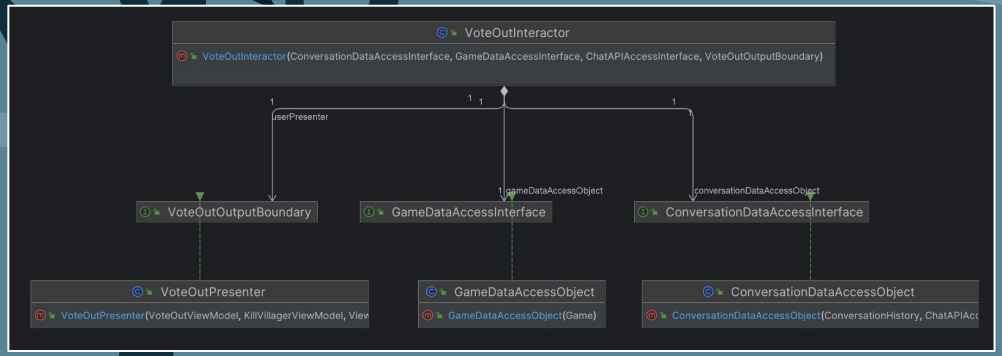
```java
public void save(PromptGenerator promptGenerator) {
    ConversationHistory conversationHistory = promptGenerator.getConversationHistory();

    // Compress old conversation before saving.
    String conversationToCompress = conversationHistory.getConversationToCompress();
    String prompt = "Summarize the following conversation between the User and ChatGPT: "
            + conversationToCompress;
    if (conversationToCompress != null) {
        String compressedConversation = compressionAPI.getResponse(prompt);
        conversationHistory.addCompressedConversation(compressedConversation);
    }
    this.promptGenerator = promptGenerator;
}
```


03
Product Demonstration


Vote out view
Enter player name:
Kevin
Meldung
No such player exists
OK

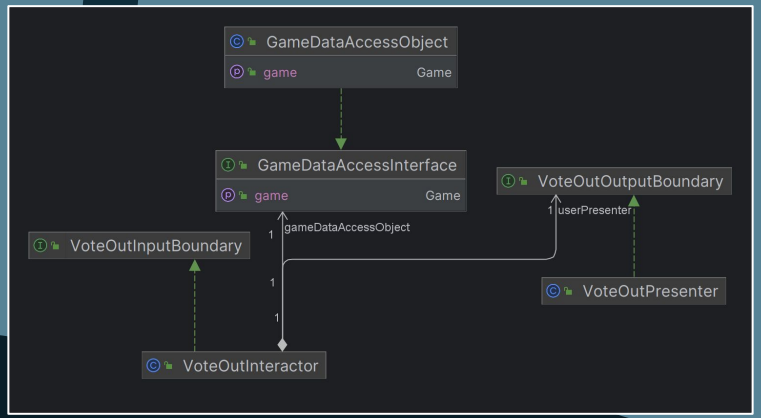

04
Design

SOLID



Interface Segregation Principle

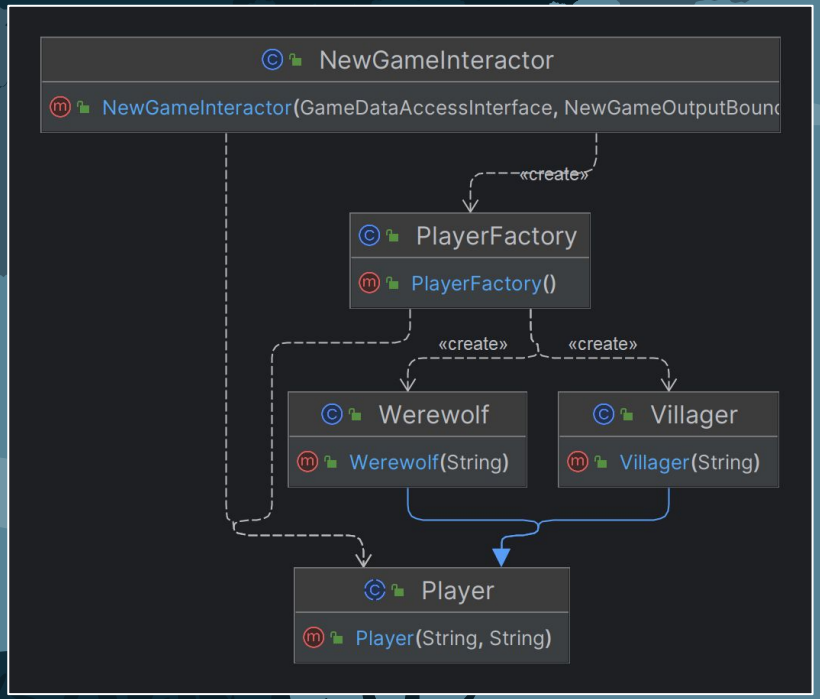
- The Data Access Interface was split up into GameDataAccessInterface and ConversationDataAccessInterface
- Thus, data access objects only implement the interfac they need



Dependency Inversion Principle

- VoteOutPresenter and GameDataAccessObject do not depend on low level modules, but abstractions
- VoteOutInteractor is composed of these abstractions

SOLID



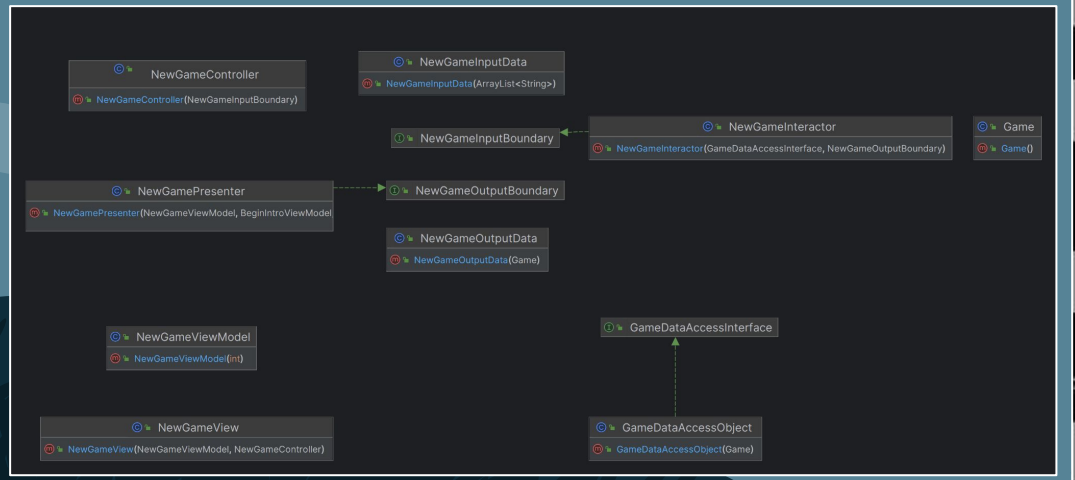
Open-Closed Principle

- *Easily open for extension*; new Player roles can be added through PlayerFactory
  - More info in **Design Patterns**
- *Closed for modification*; Interactor doesn't need to be changed to add new Player type

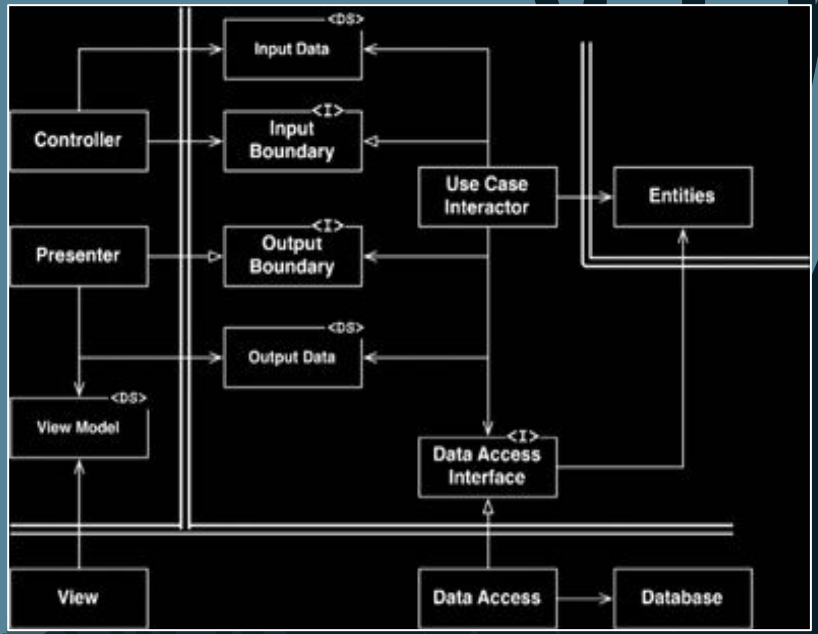
Clean Architecture

New Game Use Case UML Diagram

*Left out 'create'/composition lines



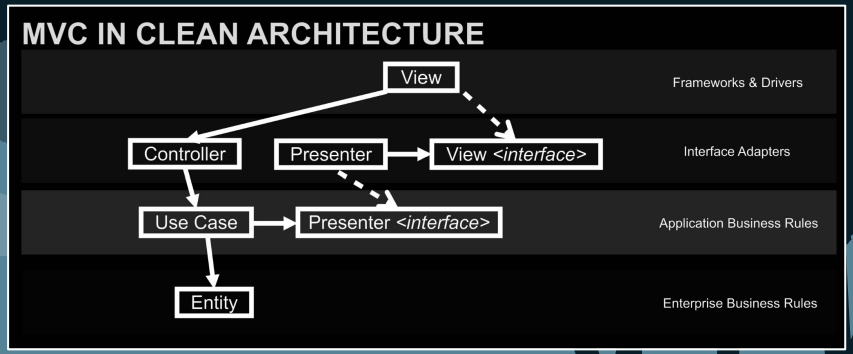
Clean Architecture Diagram



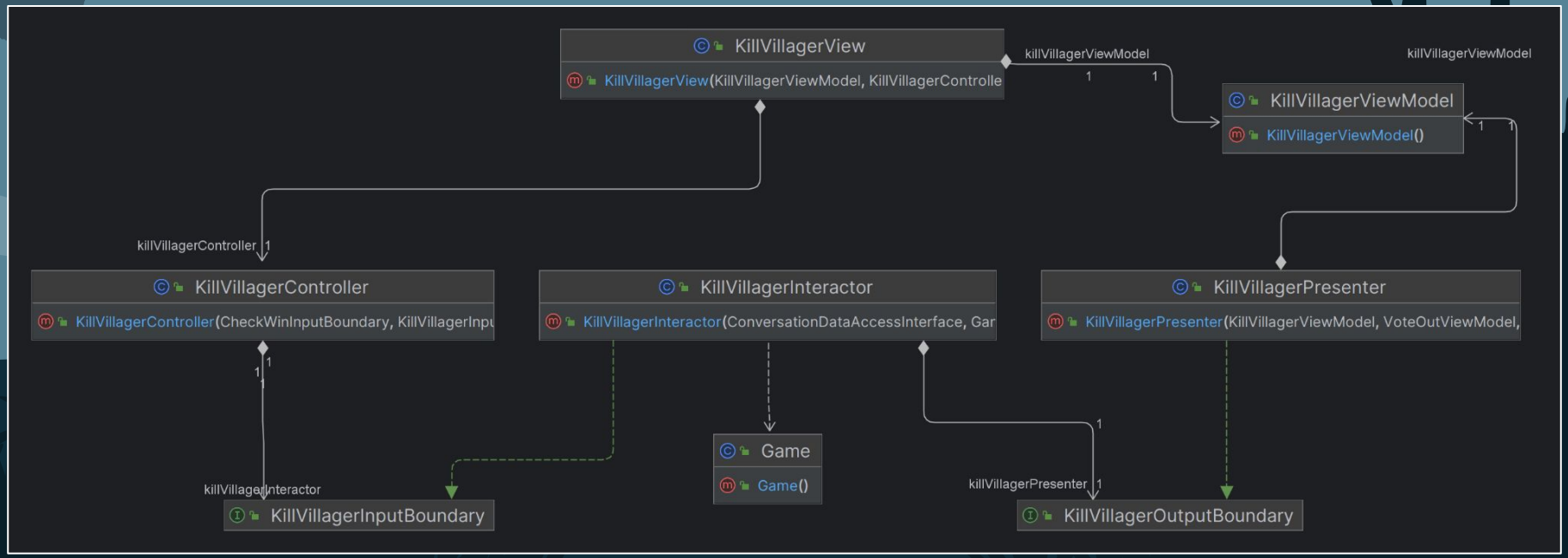
Clean Architecture

- We used the MVC model to update our views

*MVC Diagram is from Lecture 20 slides



Kill Villager Use Case UML Diagram



Design Patterns

1

Factory

PlayerFactory

2

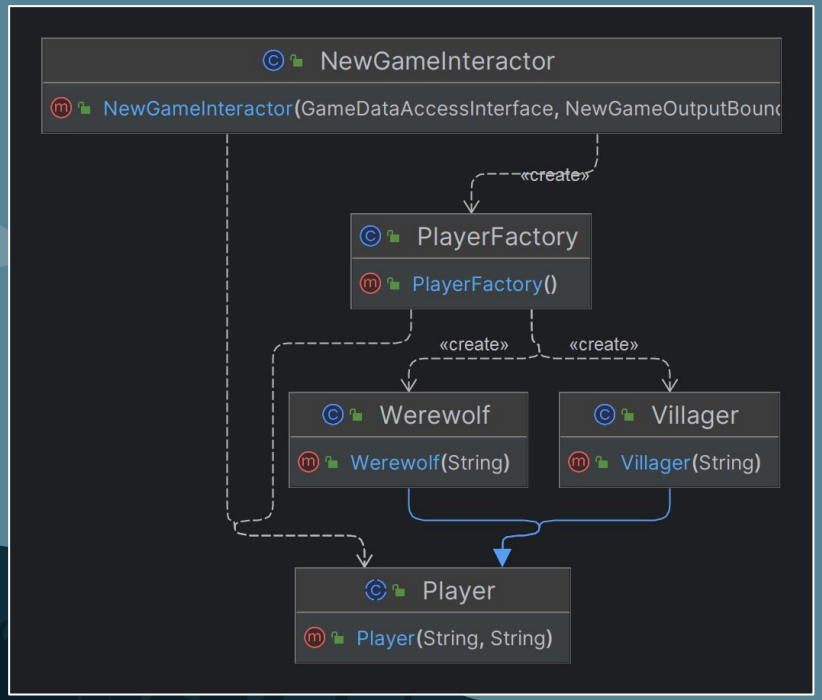
Dependency Injection

Input & Output Boundary, Data Access Interfaces

3

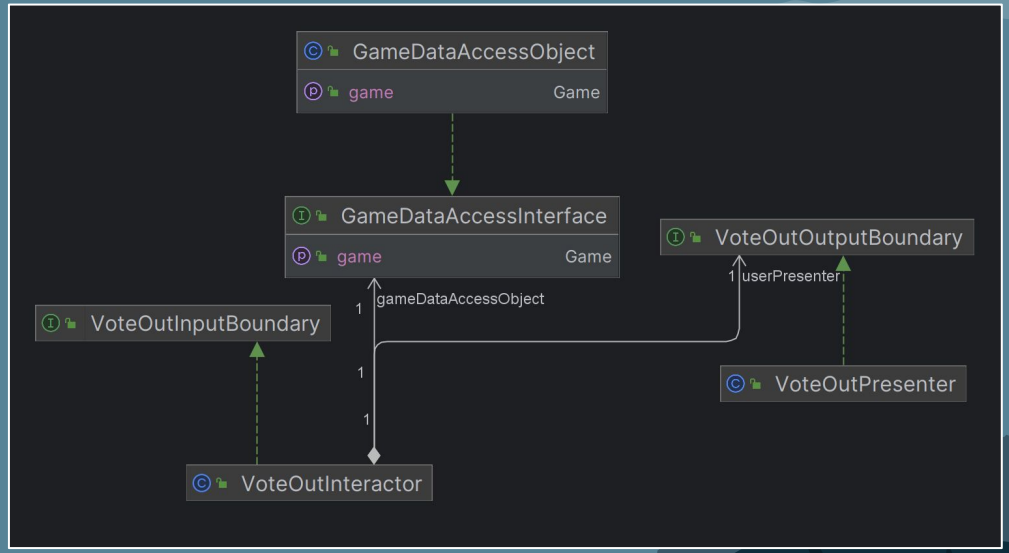
Observer

Views and ViewModels

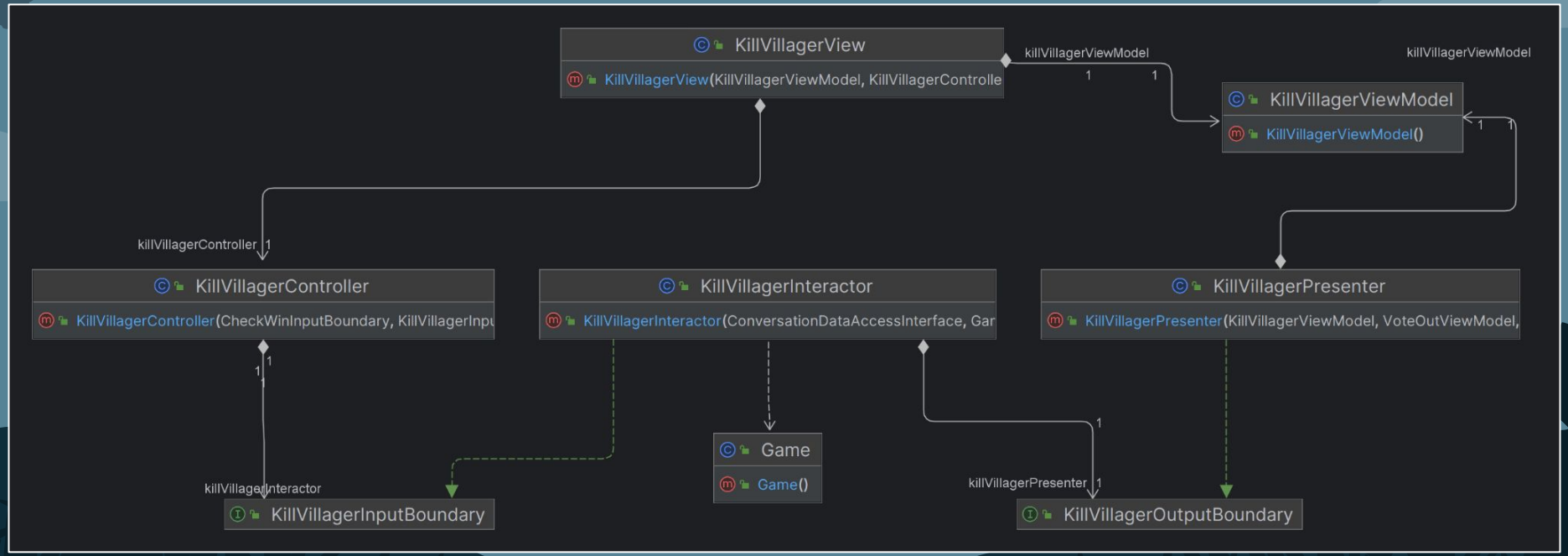

NewGameInteractor
NewGameInteractor(GameDataAccessInterface, NewGameOutputBound
«create»
PlayerFactory
PlayerFactory()
«create»
«create»
Werewolf
Werewolf(String)
Villager
Villager(String)
Player
Player(String, String)

Factory

Dependency Injection



Observer


KillVillagerView
KillVillagerView(KillVillagerViewModel, KillVillagerControlle
killVillagerViewModel
1
1
killVillagerViewModel
KillVillagerViewModel
KillVillagerViewModel()
1
1
killVillagerController
1
KillVillagerController
KillVillagerController(CheckWinInputBoundary, KillVillagerInpu
KillVillagerInteractor
KillVillagerInteractor(ConversationDataAccessInterface, Gar
KillVillagerPresenter
KillVillagerPresenter(KillVillagerViewModel, VoteOutViewModel,
1
1
1
1
Game
Game()
killVillagerInteractor
KillVillagerInputBoundary
killVillagerPresenter
1
KillVillagerOutputBoundary


05
Testing

- JUnit used to test use case interactors, data access objects, and entities
- Java Abstract Window Toolkit used to test the UI (Robot specifically)

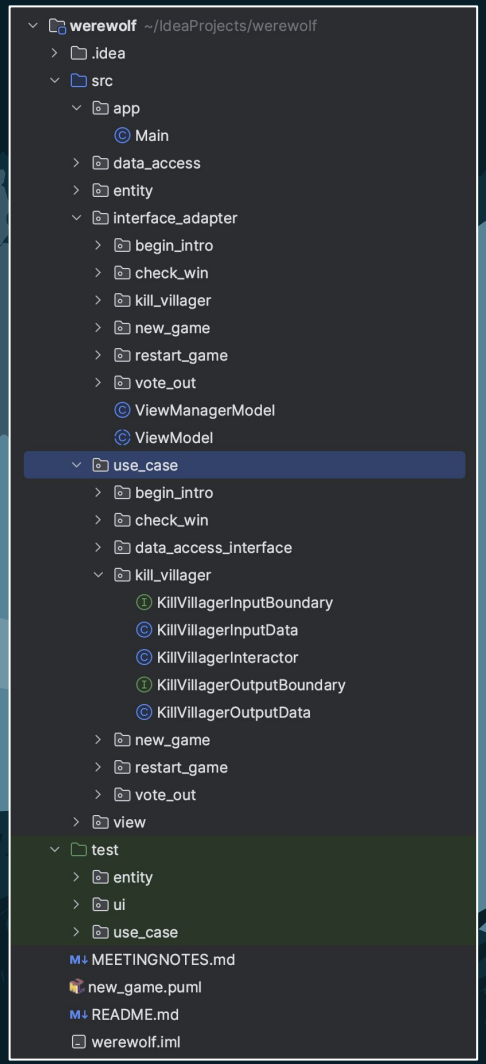
Coverage | All in werewolf

Element	Class, %	Method, %	Line, %
all	94% (69/73)	80% (198/245)	85% (721/848)
app	100% (1/1)	100% (7/7)	100% (69/69)
data_access	66% (4/6)	75% (9/12)	53% (45/84)
entity	100% (7/7)	100% (39/39)	99% (111/112)
interface_adapter	92% (24/26)	68% (72/105)	73% (151/205)
use_case	100% (16/16)	97% (35/36)	99% (149/150)
begin_intro	100% (2/2)	100% (5/5)	100% (20/20)
check_win	100% (4/4)	100% (8/8)	100% (36/36)
data_access_interface	100% (0/0)	100% (0/0)	100% (0/0)
kill_villager	100% (3/3)	100% (6/6)	100% (26/26)
new_game	100% (3/3)	100% (7/7)	100% (31/31)
restart_game	100% (1/1)	50% (1/2)	66% (2/3)
vote_out	100% (3/3)	100% (8/8)	100% (34/34)
view	100% (17/17)	78% (36/46)	85% (196/228)

```
@BeforeEach
void init() {
    Game game = new Game();
    Villager villager = new Villager( name: "Player1");
    game.addPlayer(villager);
    gameDataAccessInterface = new GameDataAccessObject(game);
    gptDataAccessObject = new DummyChatGPTAPI();
    conversationDataAccessInterface = new ConversationDataAccessObject(new ConversationHistory(),
            gptDataAccessObject);
}
```

```
@Test
void failTest() {
    killVillagerInputData = new KillVillagerInputData("Player2");
    yhz3
    killVillagerPresenter = new KillVillagerOutputBoundary() {
        yhz3
        @Override
        public void prepareSuccessView(KillVillagerOutputData villagerDeathStory) {
            fail("The interactor is supposed to fail.");
        }

        yhz3
        @Override
        public void prepareFailView(String error) {
            assertEquals( expected: "Person is not a live villager.", error);
        }
    };
}

yhz3
@AfterEach
void teardown() {
    KillVillagerInteractor killVillagerInteractor = new KillVillagerInteractor(conversationDataAccessInterface,
            gameDataAccessInterface, killVillagerPresenter, gptDataAccessObject);
    killVillagerInteractor.killVillager(killVillagerInputData);
}
```


06
Code
Organization



1

Ease of understanding

Will someone other than God understand our code in a month?

2

Relevance to *our* workflow

By Layer vs Screaming Architecture

```
General Process for Implementing UI

1. Add the View. Follow examples like NewGameView, BeginIntroView. It needs to take in the ViewModel and Controller.
    Make sure to:
    - Add a PropertyChangerListener to the ViewModel, edit the propertyChange method to do something
    - Add titles, input fields (with KeyListeners, labels, buttons (with ActionListeners), set the layout
2. Construct the View in the main program. You will essentially link everything together.
    I have organized it in the way that you just need to:
    a) Add the View Model under "// View Models"
    b) Create and add the View under "// Add the ""
        - Be sure to use a helper method i.e. get<Name>View to organize code. See examples at the end.
3. That's basically it, but there's going to be probably many issues. Some common things you might need to do:
    - Add stuff to the View Models / States that are missing (titles, button labels etc.)
    - Add the NEXT ViewModel to the current Presenter's prepareSuccessview if you want it to automatically switch.
        For example, see NewGamePresenter. This means the current Presenter depends on the next ViewModel.
 */


Yunfei (Kevin) Wang +2
public class Main {
    Yunfei (Kevin) Wang +2
    public static void main(String[] args) {
        // Build the main program window, the main panel containing the
        // various cards, and the layout, and stitch them together.

        // The main application window.
        JFrame application = new JFrame( title: "Werewolf Game Example");
        application.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        CardLayout cardLayout = new CardLayout();


        // Objects That will be used commonly
        GameDataAccessObject gameDataAccessObject = new GameDataAccessObject(new Game());
        ChatAPIAccessInterface compressionAPI = new GPT3TurboDataAccessObject();
        ChatAPIAccessInterface chatAPIAccessInterface = new GPT3TurboDataAccessObject();
```

Design Discussions

Ethan Cheung edited this page now · 5 revisions

Nov 13

We were originally going to use text files to store the game info and conversation history, but are now instead creating a class that just stores a game object instead as it is much easier and has the same functionality we want. Since by our design, games should be played in one sitting, we do not care about data persistence after we close the program, which would be where a text file would help.

Nov 18

We originally thought of having a different View and ViewModel for the stories (i.e. when the werewolves killed a villager or when a player was voted out), but we decided upon just having a popup display the story instead, as it would be simpler to implement while maintaining the same functionality, as we wouldn't need to create an entire new Continue use case just to switch from a story back into the game.

Nov 27

We talked about how we wanted to implement a Player Factory so that it would make role assignment more scalable as we create more Player roles, but since our MVP only has two roles, we have left it out for now, and will consider it for the future.

We also discussed implementing the logic of CheckWin in the VotePlayer use case since technically wins are decided during VotePlayer (i.e. if there are 2 players and 1 werewolf left and a player is voted out, the werewolf is guaranteed to win as he can kill the remaining player at night), but at that point the application and business logic were already tied to doing it in the way such that games end when either all werewolves are dead or all villagers are dead.


Thank you!