Final Report

Avita Sharma, Eric Wyss, and David Taus

https://github.com/Perditian/DMorDie

Team Name: DragonSlayers

Group Members:

Avita Sharma, Eric Wyss, and David Taus

Project: DM or Die

A text based adventure game, where the user controls the dungeon master. They can manipulate intelligent 'player characters' who interact in the game. Example: the user can decide if a player succeeds or fails at a task. These players are concurrent processes which make demands on the dungeon master. These demands must be answered in real time or a default behavior is triggered, the default generally having a negative consequence. Multiple demands can come from each 'player character' at the same time, and the user must decide which one to interact with (or if they are quick, try to do them all). The user can directly control monsters and non-playable characters (NPCs) for the 'player characters' to interact with.

Outcome:

Minimum deliverable:

X Split Screen graphical interface.

X Two 'Player Characters' with a set alignment and class.

X One Encounter involving talking to one NPC and one battle.

X The default map has a dungeon and tavern.

X Ability to pause the game.

X Default/unchangeable inventory.

Maximum deliverable:

* * Implement up to six characters.

→ Implemented four different kinds of playable characters.

* * More classes, more monsters.

* * Ability to add more places and characters to the map.

X Add more encounters. Increase the complexity of encounters.

* * Basic Excitement meter for each ‘player character’. The game is scored on total excitement. Excitement is a metric for how much fun the ‘player characters’ are having with the user’s decisions. (i.e. not purposely failing the battle.) Excitement $\in \mathbb{R}$.
* * More user customizable options for the campaign.
* * Option to save the game.

We achieved our minimal deliverable. However, we decided to refine our program instead of adding more features to match our maximum deliverable. So our encounters with the characters are longer and more complex, but we only have two NPCs, up to four different playable characters, and one monster battle.

Challenges and Bug Report:

Definitely, testing was a big challenge as the playable characters decide to do their own actions, so we had to keep playing our game until the specific sequence of events that we wanted to test happened. Before we had the game state monitor, most bugs from the playable characters occured because an Event was not cleared or set properly. In the future, we could make scripts which run a different thread function and force the Playable Characters to do certain functions.

Basically, the biggest challenges with the post office were testing. On its face the system was fairly straight forward but making sure that it was going to work as intended was problematic. Also I tried to minimize all function calls because of the GIL. So I wanted to offload work onto AI for interpreting and responding.

Another challenge was making sure objects would have an expiration without potentially having to run into references to objects that were already deleted so just deleting them could be problematic which was why instead opted for the validity field.

Tkinter’s documentation was somewhat dated and inconsistent for some widgets.

Initially the screen was setup using multiple message widgets which each displayed a line of output and transferred text between them because the textbox widget wasn’t behaving properly with scrollbars and spacing. At first it worked fine but eventually as the game got bigger and more complex it made the gui unbearably slow so we had to reconstruct the layout and make it work for a single widget.

It was a little tricky making it so after you interrupt a character the next input addresses only that character and doesn’t perform other actions because we had to unbind and rebind events to different widgets and functions

Design Reflection:

Making the Game State class, which is a collection of all the shared memory of the Playable Characters, GUI, and Battle threads, a Monitor ensured that no race conditions or contradictions occur when we change the Game State.

Having the Post Office asynchronously send messages to other Playable Characters that expire models real life, and allows for ‘synchonous’ messaging without deadlocks. A playable character can send an expiring message and wait to see if it has been read with a timeout. If it does not get read in time, the message expires and can no longer be interacted with. If it does get read, then the sender stops waiting immediately.

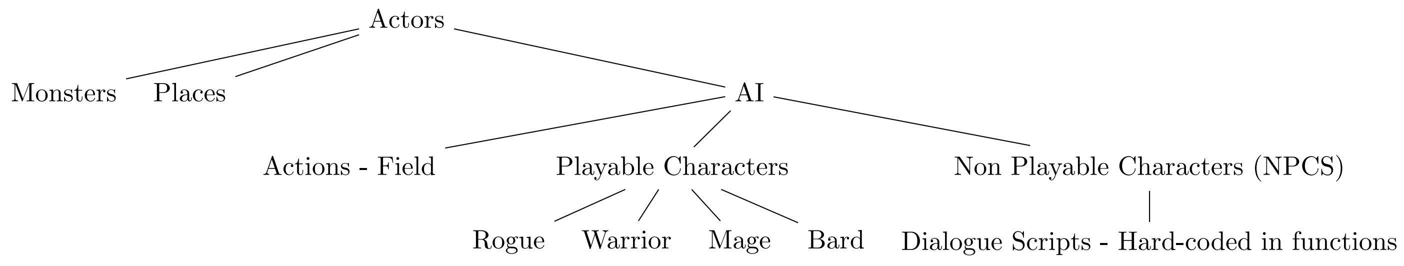
Integrating the Dungeon Master and GUI class definitely improved the functionality of our program. The Dungeon Master essentially parses commands from the GUI and sends those commands to the GUI or Playable character to complete.

Next time, I think we can generalize more functions, especially the action functions a playable character can do. Also, use more functional programming to optimize the code. (Using more first-order functions.) Also use more global constants!

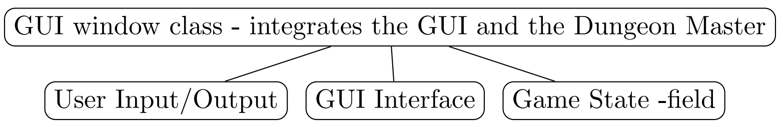
We have hard-coded the dialogue into the program, but if we make a more extensive dialogue system, we should try to implement it in XML or JSON, and have our program parse dialogue from these files instead.

Class Diagrams

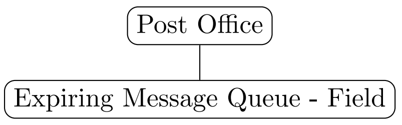
Character Class Hiearchy:



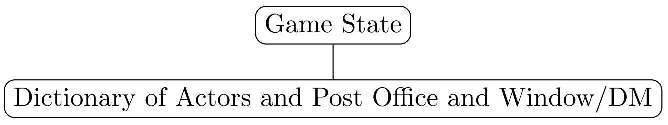
Dungeon Master:


GUI window class - integrates the GUI and the Dungeon Master
User Input/Output
GUI Interface
Game State -field

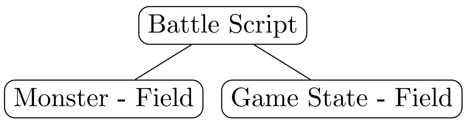
Messaging:



Game State:



Battle:

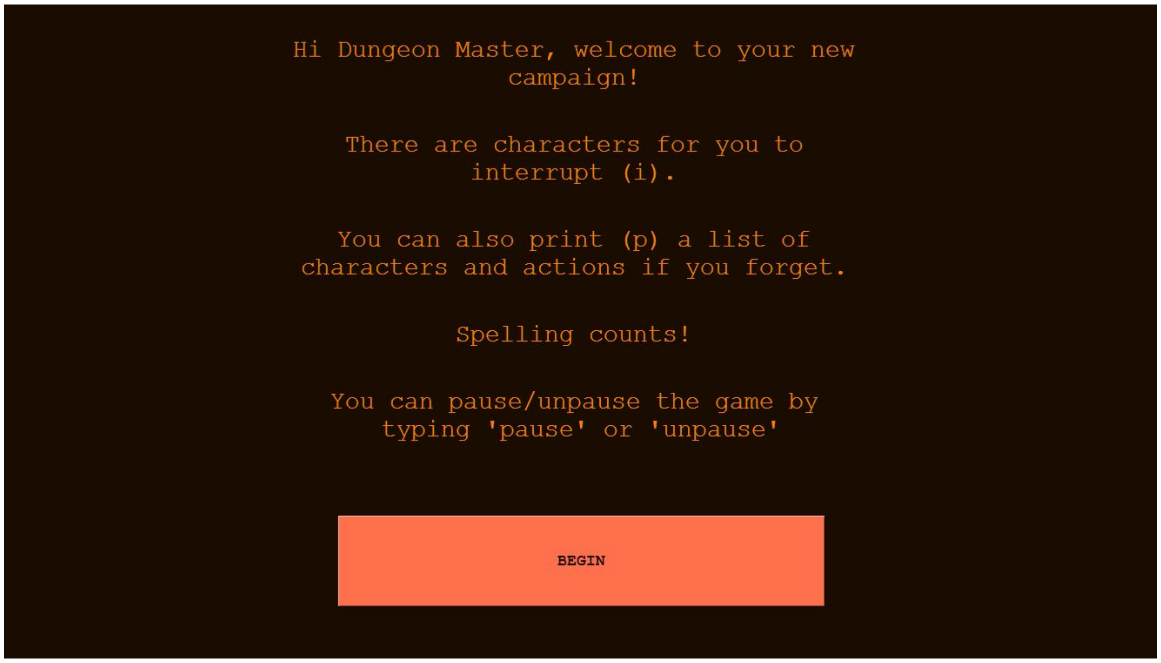


The Game State will be a tuple containing a dictionary of all the AI and Places in the Game (keys are the name or ID of the Actor, values are the Actor), The Message Queue, the Window (GUI) and the Game Lock. The Game State is a Monitor Class to ensure that clients use the lock when accessing or modifying the Game State. It will include a function that takes a function and calls it using the Game State's lock. The Game State will be passed to all Actors on the Board and the Dungeon Master.

AI are goal-based agents who use utility to decide actions. The current Goal is selected using a transition matrix (Markov Chain). For the current Goal, the action with the maximum expected utility is chosen, and the Actor attempts to perform the action. If a contradiction arises, the Actor chooses a new Goal (it can be the same one). Otherwise, the Actor compares the utility gained (or lost) to their expected utility. If they succeeded, their probability of success of performing that action increases. If they failed, their probability of success of that action decreases. This ensures that the AI is more likely to continue to do the action if they perform successfully, and less likely to do it if they fail.

The Battle class is a thread that checks each playable character and determines if they are all prepared to go to battle. If everyone is prepared, it signals to the playable character threads to kill themselves, and then sends the Game State to battle. There, it creates attack threads for each playable character during their turn, and joins them when their attacks have completed. Once the battle is completed, the game ends.

UML Diagrams



Hi Dungeon Master, welcome to your new campaign!

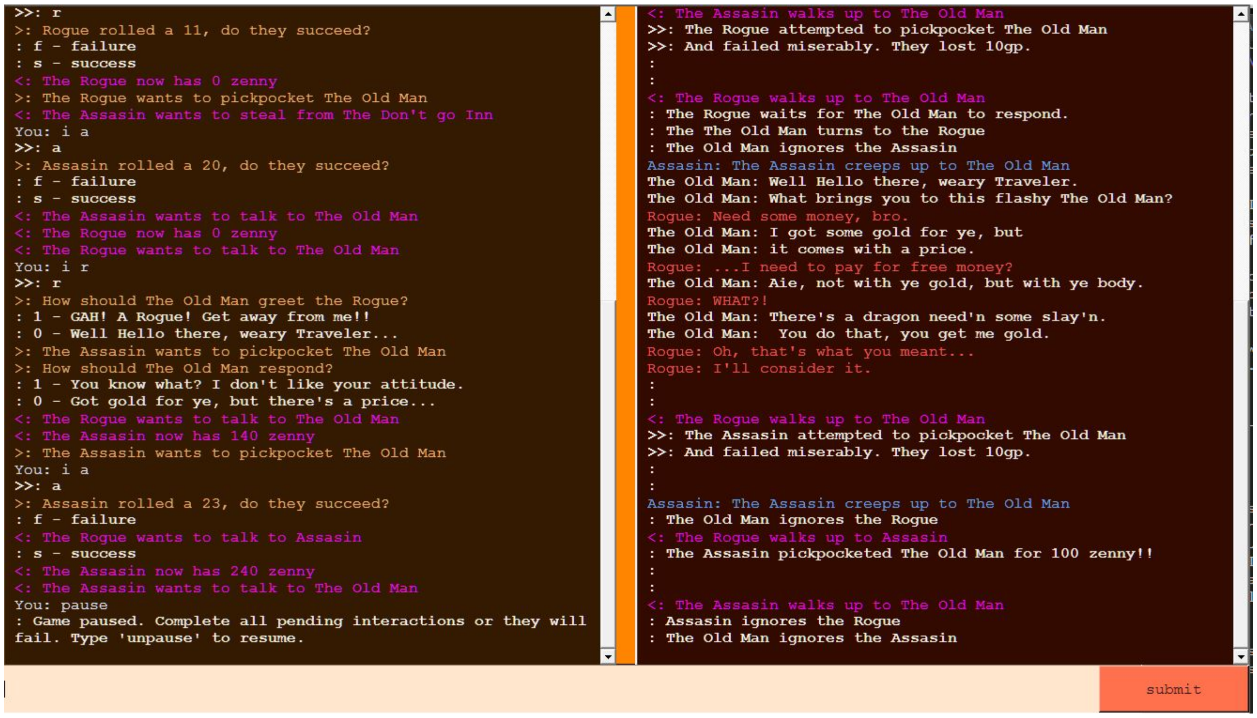
There are characters for you to interrupt (i).

You can also print (p) a list of characters and actions if you forget.

Spelling counts!

You can pause/unpause the game by typing 'pause' or 'unpause'

This figure displays the game start screen. Here, the Playable Character threads have already started, but are paused (waiting) until the user clicks the begin button. Then, the threads acquire the GUI lock and release it for others to use like a turnstile.


```
>>: r
>: Rogue rolled a 11, do they succeed?
: f - failure
: s - success
<: The Rogue now has 0 zenny
>: The Rogue wants to pickpocket The Old Man
<: The Assasin wants to steal from The Don't go Inn
You: i a
>>: a
>: Assasin rolled a 20, do they succeed?
: f - failure
: s - success
<: The Assasin wants to talk to The Old Man
<: The Rogue now has 0 zenny
<: The Rogue wants to talk to The Old Man
You: i r
>>: r
>: How should The Old Man greet the Rogue?
: 1 - GAH! A Rogue! Get away from me!!
: 0 - Well Hello there, weary Traveler...
>: The Assasin wants to pickpocket The Old Man
>: How should The Old Man respond?
: 1 - You know what? I don't like your attitude.
: 0 - Got gold for ye, but there's a price...
<: The Rogue wants to talk to The Old Man
<: The Assasin now has 140 zenny
>: The Assasin wants to pickpocket The Old Man
You: i a
>>: a
>: Assasin rolled a 23, do they succeed?
: f - failure
<: The Rogue wants to talk to Assasin
: s - success
<: The Assasin now has 240 zenny
<: The Assasin wants to talk to The Old Man
You: pause
: Game paused. Complete all pending interactions or they will
fail. Type 'unpause' to resume.
```

```
<: The Assasin walks up to The Old Man
>>: The Rogue attempted to pickpocket The Old Man
>>: And failed miserably. They lost 10gp.
:
:
<: The Rogue walks up to The Old Man
: The Rogue waits for The Old Man to respond.
: The The Old Man turns to the Rogue
: The Old Man ignores the Assasin
Assasin: The Assasin creeps up to The Old Man
The Old Man: Well Hello there, weary Traveler.
The Old Man: What brings you to this flashy The Old Man?
Rogue: Need some money, bro.
The Old Man: I got some gold for ye, but
The Old Man: it comes with a price.
Rogue: ...I need to pay for free money?
The Old Man: Aie, not with ye gold, but with ye body.
Rogue: WHAT?!
The Old Man: There's a dragon need'n some slay'n.
The Old Man: You do that, you get me gold.
Rogue: Oh, that's what you meant...
Rogue: I'll consider it.
:
:
<: The Rogue walks up to The Old Man
>>: The Assasin attempted to pickpocket The Old Man
>>: And failed miserably. They lost 10gp.
:
:
Assasin: The Assasin creeps up to The Old Man
: The Old Man ignores the Rogue
<: The Rogue walks up to Assasin
: The Assasin pickpocketed The Old Man for 100 zenny!!
:
:
<: The Assasin walks up to The Old Man
: Assasin ignores the Rogue
: The Old Man ignores the Assasin
```

This figure details an instance of the game running. Here, the Rogue and Assassin are the only Playable Characters, and they interact with each other and with an NPC, the Old Man. The Dungeon Master determines if an action from the Rogue or Assassin succeeds or fails. They also can choose how an NPC responds to the Playable Characters. Here, the Rogue talks to the Old Man. The Dungeon Master choose to let Old Man tell the Rogue about a Dragon that needs to be battled. Meanwhile, The Assassin wants to pickpocket the Old man. They wait for the Dungeon Master to interact with them. Since the Dungeon Master is quick, they are able to interact with the Assassin and chose to let the Assassin succeed at pickpocketing the Old Man. The game is then paused.

Division of Labor

The project was able to be broken down to allow for members to work independently. Integrating each component took some collaboration, but for the most part we were able to independently and simutaneously update our code and run the program without any drastic conflicts.

Code Overview:

File Name	Description
AI.py	Generalizes how each Playable Character functions. Includes common fields for all characters. (AI class)
Action.py	Maps the action to its utility, updates the action based on performance.
Battle.py	Includes the Monster and Battle Class.
DMorDie.py	Main Program, run to play game. Initializes and runs the game.
DungeonMaster.py	The GUI class. Creates and updates the GUI.
GameState.py	The Game State class. A collection of shared memory under a monitor.
NPC_and_Location.py	The NPC and Location classes. NPC extends AI.
Rogue.py	The Rogue Class. A Playable Character, includes the actions to send to the AI class, and an Attack function for the Battle.
Warrior.py	The Warrior Class. A Playable Character, includes the actions to send to the AI class, and an Attack function for the Battle.
expiringObject.py	Includes the expring messages to send with the Post Office.
postOffice.py	Handles messaging between Playable Characters. Messages are expiring messages.