Escape From Hellspawn (EFH) - Design Patterns

Toronto Metropolitan University

Course: CCPS406 - Intro to Software Engineering - P2022

Student Names: Jay Sharma, Dryden Zarate, Clement Zhang

Date: June 11, 2022

Creational Design Patterns

Factory Method

From EFH's architecture design document (please see Figure 1.0 below), the starting point of the program is main.py which has the game loading code that reads dynamic game data in JSON format and creates various program objects based on that information.

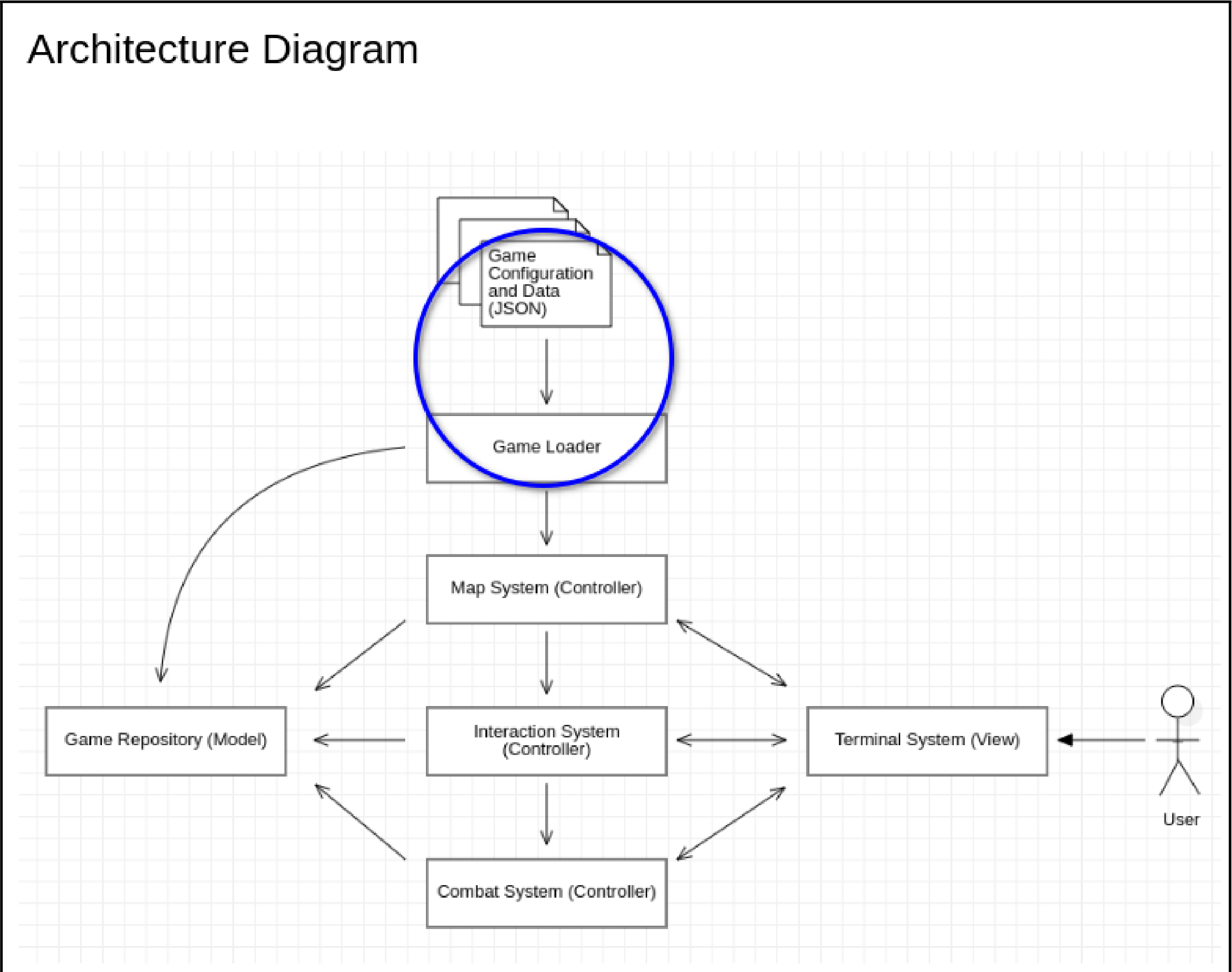
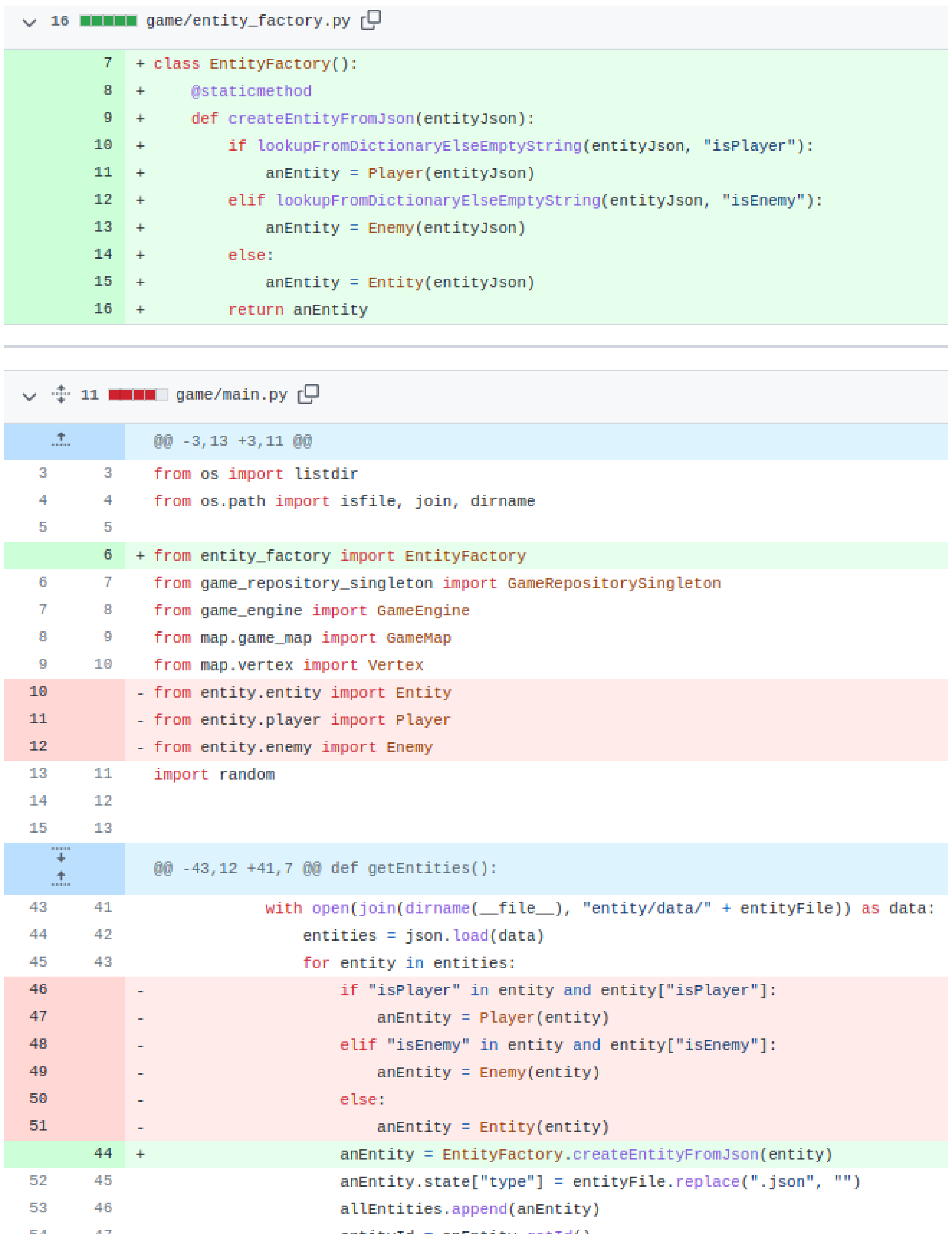


Figure 1.0: A look back at EFH's architectural design pattern

From [2(pg68)], the factory method creational design pattern is recommended "*when we want to call a method such that we pass in a string and get as a return value a new object … the type of the object is determined by the string that is passed to the method*". We've identified an area in the codebase where we can apply this design pattern (please see Figure 1.1 below).



```
16 game/entity_factory.py

+ class EntityFactory():
+     @staticmethod
+     def createEntityFromJson(entityJson):
+         if lookupFromDictionaryElseEmptyString(entityJson, "isPlayer"):
+             anEntity = Player(entityJson)
+         elif lookupFromDictionaryElseEmptyString(entityJson, "isEnemy"):
+             anEntity = Enemy(entityJson)
+         else:
+             anEntity = Entity(entityJson)
+         return anEntity
```

```
11 game/main.py

@@ -3,13 +3,11 @@
  from os import listdir
  from os.path import isfile, join, dirname

+ from entity_factory import EntityFactory
  from game_repository_singleton import GameRepositorySingleton
  from game_engine import GameEngine
  from map.game_map import GameMap
  from map.vertex import Vertex
- from entity.entity import Entity
- from entity.player import Player
- from entity.enemy import Enemy
  import random


@@ -43,12 +41,7 @@ def getEntities():
              with open(join(dirname(__file__), "entity/data/" + entityFile)) as data:
                  entities = json.load(data)
                  for entity in entities:
-                     if "isPlayer" in entity and entity["isPlayer"]:
-                         anEntity = Player(entity)
-                     elif "isEnemy" in entity and entity["isEnemy"]:
-                         anEntity = Enemy(entity)
-                     else:
-                         anEntity = Entity(entity)
+                     anEntity = EntityFactory.createEntityFromJson(entity)
                      anEntity.state["type"] = entityFile.replace(".json", "")
                      allEntities.append(anEntity)
```

Figure 1.1: EFH using the factory method design pattern

From Figure 1.1 above, we introduced a new class called EntityFactory which has a static method called “createEntityFromJson” and this refactor has led to less code in main.py. Based on the entity’s type defined in the JSON, this factory method will create the corresponding class instance e.g. Player, Enemy, or generic Entity. Should the game require more custom entity types, EntityFactory’s createEntityFromJson could be expanded to account for it.

Singleton

From EFH’s architecture design document (please see Figure 1.2 below), there is a GameRepository class instance whose responsibilities are reading and manipulating the game’s state.

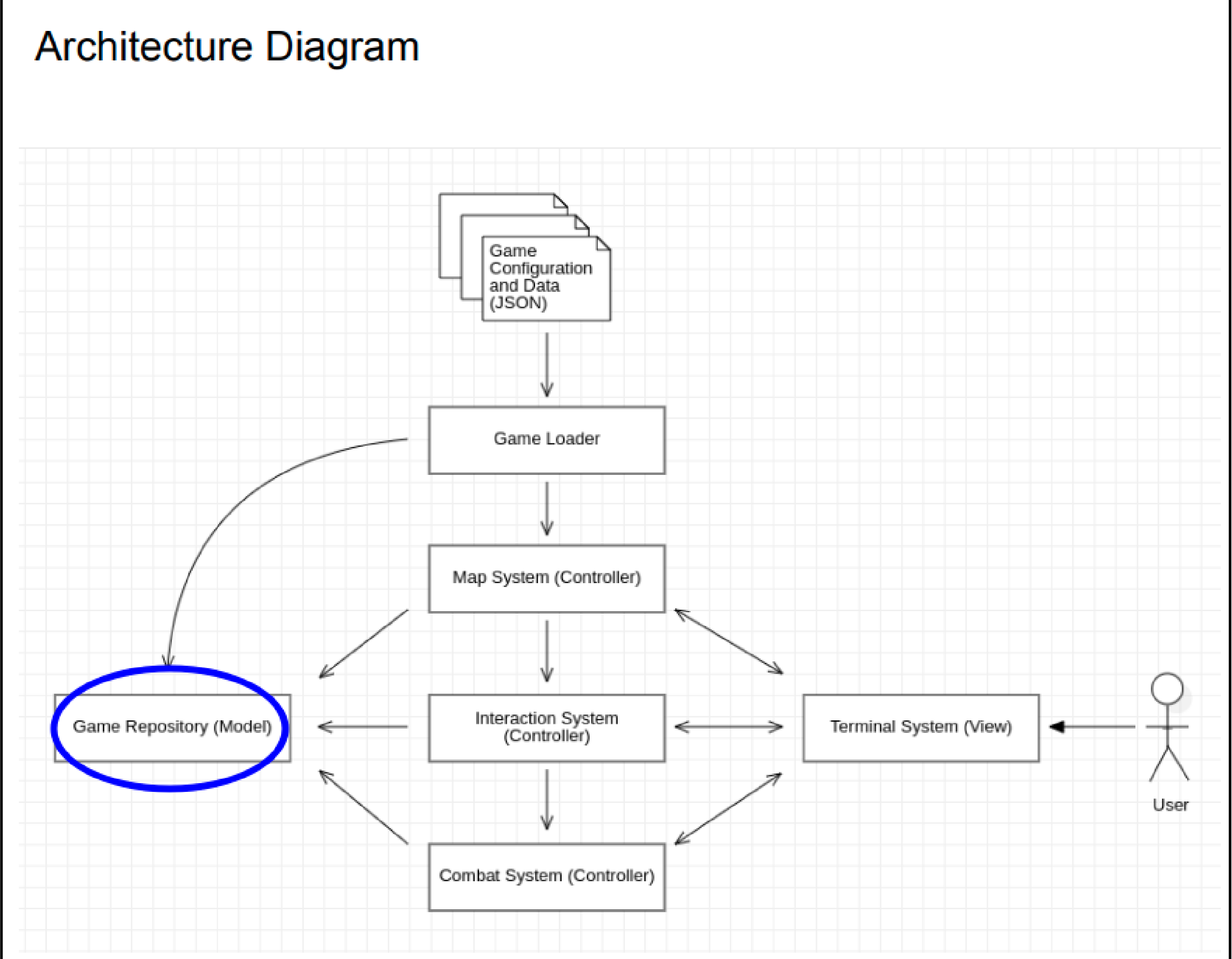
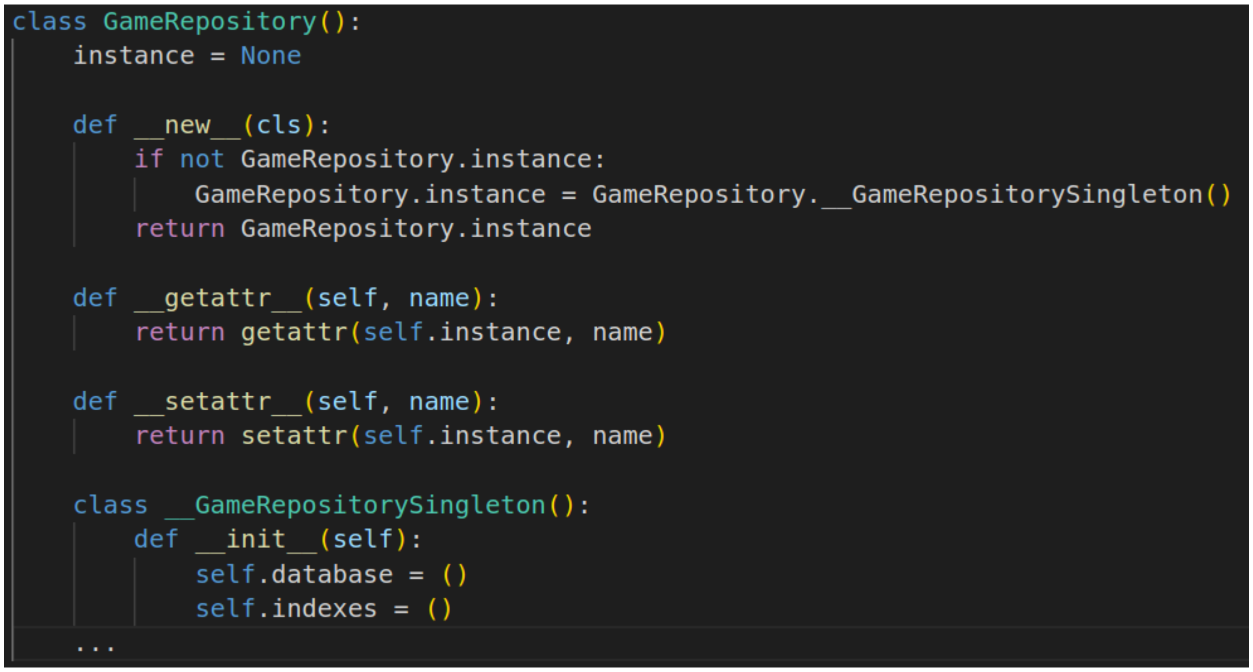


Figure 1.2: A look back at EFH’s architectural design pattern

Various MVC controllers above have a reference to this GameRepository instance and ideally this instance should be a **singleton** to better control access to this shared resource[1(pg159)] (namely the game's state). Furthermore, we wouldn't want to reinitialize this instance for each controller which could be expensive from a memory standpoint as the game evolves. EFH is also not doing any multithreading so there wouldn't be any concurrency concerns such as a deadlock situation among controllers having access to this one singleton. To convert our GameRepository instance as a singleton, we used [2(pg23)] as a reference (please see Figure 1.3 below).



```python
class GameRepository():
    instance = None

    def __new__(cls):
        if not GameRepository.instance:
            GameRepository.instance = GameRepository.__GameRepositorySingleton()
        return GameRepository.instance

    def __getattr__(self, name):
        return getattr(self.instance, name)

    def __setattr__(self, name):
        return setattr(self.instance, name)

    class __GameRepositorySingleton():
        def __init__(self):
            self.database = ()
            self.indexes = ()
    ...
```

Figure 1.3: EFH's GameRepository Singleton

Essentially, the first instantiation of a GameRepository would initialize "instance" above which is defined at the class-level – that is, it is the same object for every GameRepository instance. Method **__new__**, which is called when a class is being instantiated, is overridden to return this singleton and in doing so the singleton pattern is achieved. We confirmed this, as per [2(pg34)], that subsequent instantiation of GameRepository after the first instantiation would return the same object created by the first instance.

Structural Design Patterns

Adapter

From EFH's architecture design document (please see Figure 1.4 below), there is a GameTerminal class instance whose responsibilities are getting input from and printing output to the user.

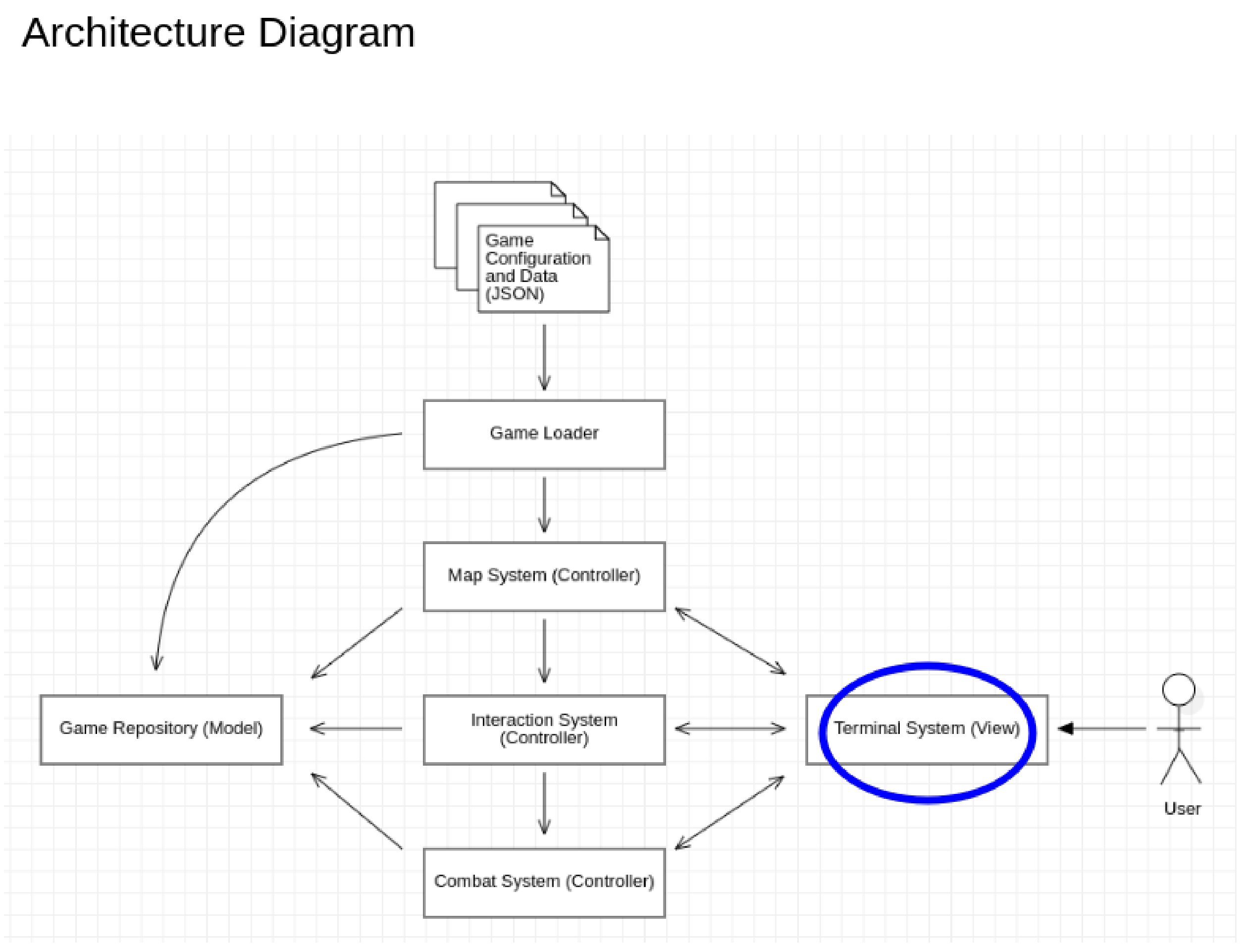


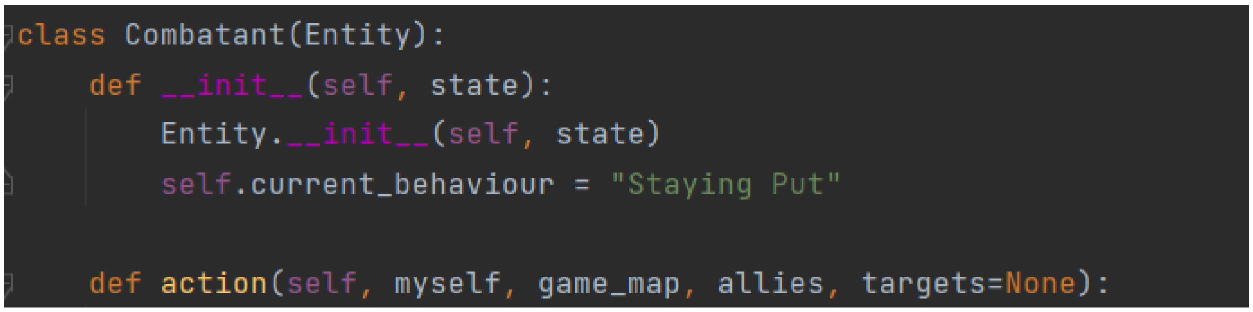
Figure 1.4: A look back at EFH's architectural design pattern

Rather than calling Python "input" and "print" directly from many areas in the codebase, we decided to introduce a class called GameTerminal which is an **adapter** to user I/O related functionalities. The advantage of which is EFH having a central place to prompt commands from the user and prettify the output as per project requirements. For example, it made it easier for us to ensure ALL print calls add an extra newline for better readability and we were able to cap each line by a certain number of characters from this central place in the code. We identified that this is an adapter design pattern as opposed to a proxy design pattern because *“adapter provides a* ***different interface*** *to the wrapped object, (whereas) Proxy provides it with the same interface”* [2(pg245)].

Facade

Facade is a structural design pattern that provides a simplified interface for the user when a complex set of classes with a lot of moving parts are involved. Entities are not static, in that they still take actions even when the player does not notice. By setting up a series of behaviours (e.g. Patrolling, Attacking, Escaping, etc) and actions that occur when under that behaviour (e.g. Patrolling has the entity go to a random adjacent room every turn), it creates the illusion that the monsters and NPCs in our game are more alive than they are in actuality.

This functionality is built into every single non-player entity as part of their class, while also allowing a user to tweak it on a case-by-case basis (e.g. the Zombie King acts differently from a Zombie). When another programmer wants to utilise this basic AI for other entities, there is no need to understand the specific code, as the entity has a built-in action() function. This automatically determines what the entity should do, whether in or out of combat and executes that action. This makes it so a change to the underlying AI only needs to happen on the Combatant class and it will apply to every other entity in the game, while also making it so that the user doesn’t need to know how it works, just that it will work.[3]



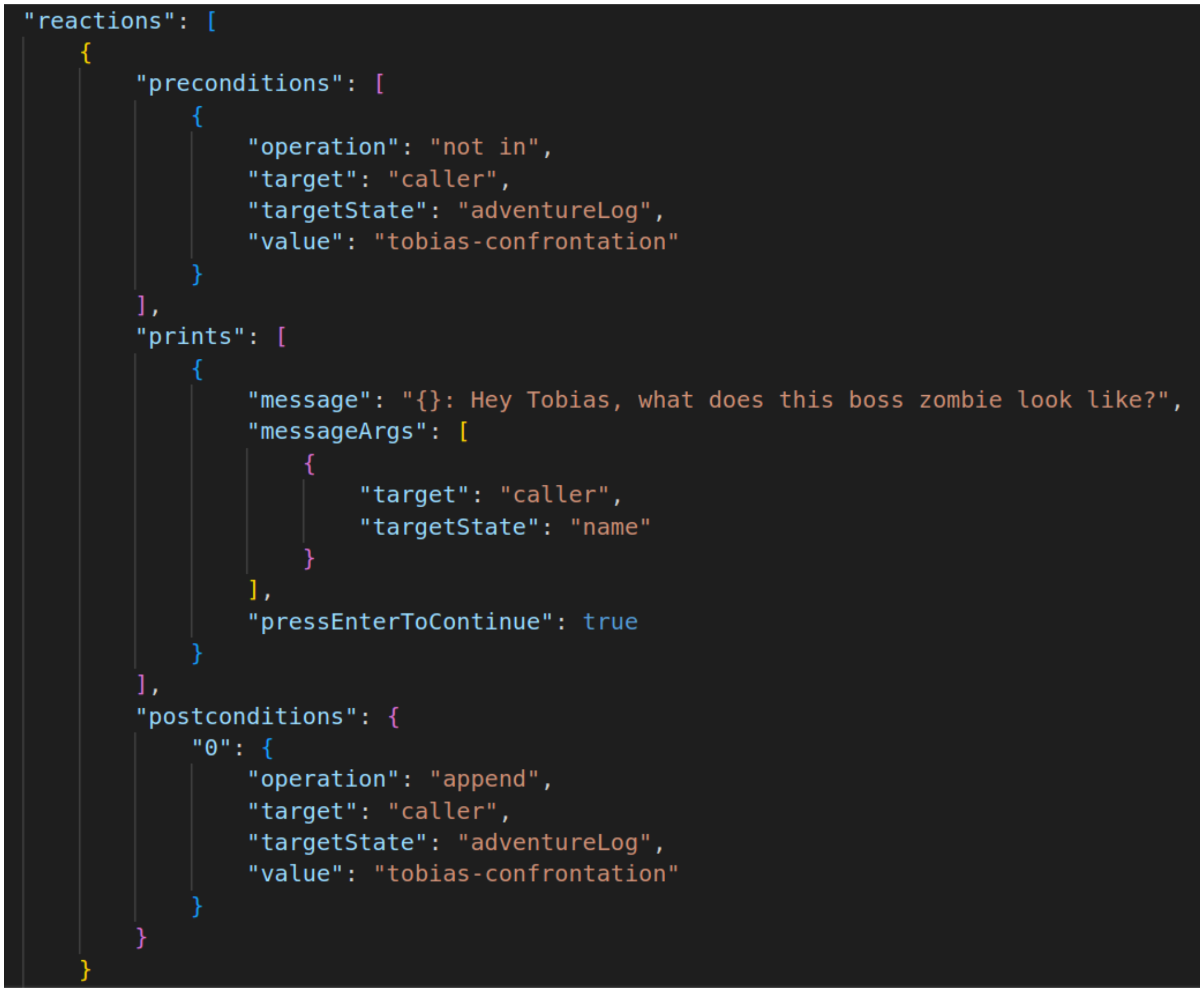
```
class Combatant(Entity):
    def __init__(self, state):
        Entity.__init__(self, state)
        self.current_behaviour = "Staying Put"

    def action(self, myself, game_map, allies, targets=None):
```

Figure 1.5: A look back at the entity.action() command

Behavioural Design Patterns

Command

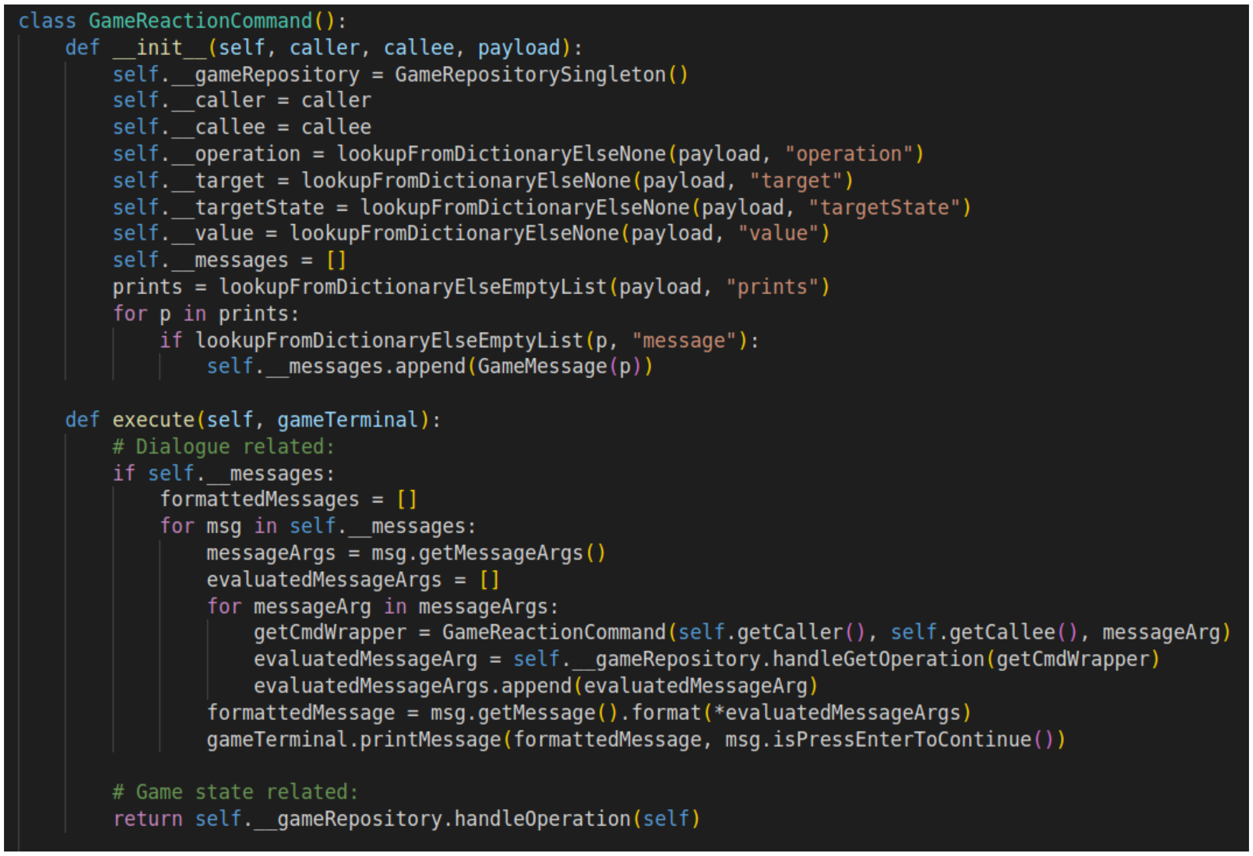
From EFH's textual data document (please see Figure 1.6 below), an entity's reaction to another entity is configured via JSON:



```
"reactions": [
    {
        "preconditions": [
            {
                "operation": "not in",
                "target": "caller",
                "targetState": "adventureLog",
                "value": "tobias-confrontation"
            }
        ],
        "prints": [
            {
                "message": "{}: Hey Tobias, what does this boss zombie look like?",
                "messageArgs": [
                    {
                        "target": "caller",
                        "targetState": "name"
                    }
                ],
                "pressEnterToContinue": true
            }
        ],
        "postconditions": {
            "0": {
                "operation": "append",
                "target": "caller",
                "targetState": "adventureLog",
                "value": "tobias-confrontation"
            }
        }
    }
```

Figure 1.6: A look back at EFH's textual data document

The **command** behavioural design pattern was used to encapsulate the above commands which read or update game state. As per [1(pg268)], the command design pattern *"turns a request into a stand-alone object that contains all information about the request"* which is exactly EFH's GameReactionCommand purpose (please see Figures 1.7):



```python
class GameReactionCommand():
    def __init__(self, caller, callee, payload):
        self.__gameRepository = GameRepositorySingleton()
        self.__caller = caller
        self.__callee = callee
        self.__operation = lookupFromDictionaryElseNone(payload, "operation")
        self.__target = lookupFromDictionaryElseNone(payload, "target")
        self.__targetState = lookupFromDictionaryElseNone(payload, "targetState")
        self.__value = lookupFromDictionaryElseNone(payload, "value")
        self.__messages = []
        prints = lookupFromDictionaryElseEmptyList(payload, "prints")
        for p in prints:
            if lookupFromDictionaryElseEmptyList(p, "message"):
                self.__messages.append(GameMessage(p))

    def execute(self, gameTerminal):
        # Dialogue related:
        if self.__messages:
            formattedMessages = []
            for msg in self.__messages:
                messageArgs = msg.getMessageArgs()
                evaluatedMessageArgs = []
                for messageArg in messageArgs:
                    getCmdWrapper = GameReactionCommand(self.getCaller(), self.getCallee(), messageArg)
                    evaluatedMessageArg = self.__gameRepository.handleGetOperation(getCmdWrapper)
                    evaluatedMessageArgs.append(evaluatedMessageArg)
                formattedMessage = msg.getMessage().format(*evaluatedMessageArgs)
                gameTerminal.printMessage(formattedMessage, msg.isPressEnterToContinue())

        # Game state related:
        return self.__gameRepository.handleOperation(self)
```

Figure 1.7: EFH’s GameReactionCommand class

The GameReactionCommand encapsulates an instruction to read or update a game’s state and it contains all the relevant information needed to fulfil that command including what to print to the user. This class has an **execute** method which would print to the user and/or mutate the game’s state.

By introducing and making use of this class, the codebase was improved because the details of such instructions are decoupled from classes which would otherwise need to parse these reactions from the JSON.

References

1. Shvets, A. (2021). Dive Into Design Patterns. Alexander Shvets.

2. Badenhorst, W. (2017). Practical Python Design Patterns: Pythonic Solutions to Common Problems. Apress L. P. E-Book Central. https://ebookcentral.proquest.com/lib/ryerson/detail.action?docID=5107901

3. Facade. (2022). Retrieved 12 June 2022, from https://refactoring.guru/design-patterns/facade