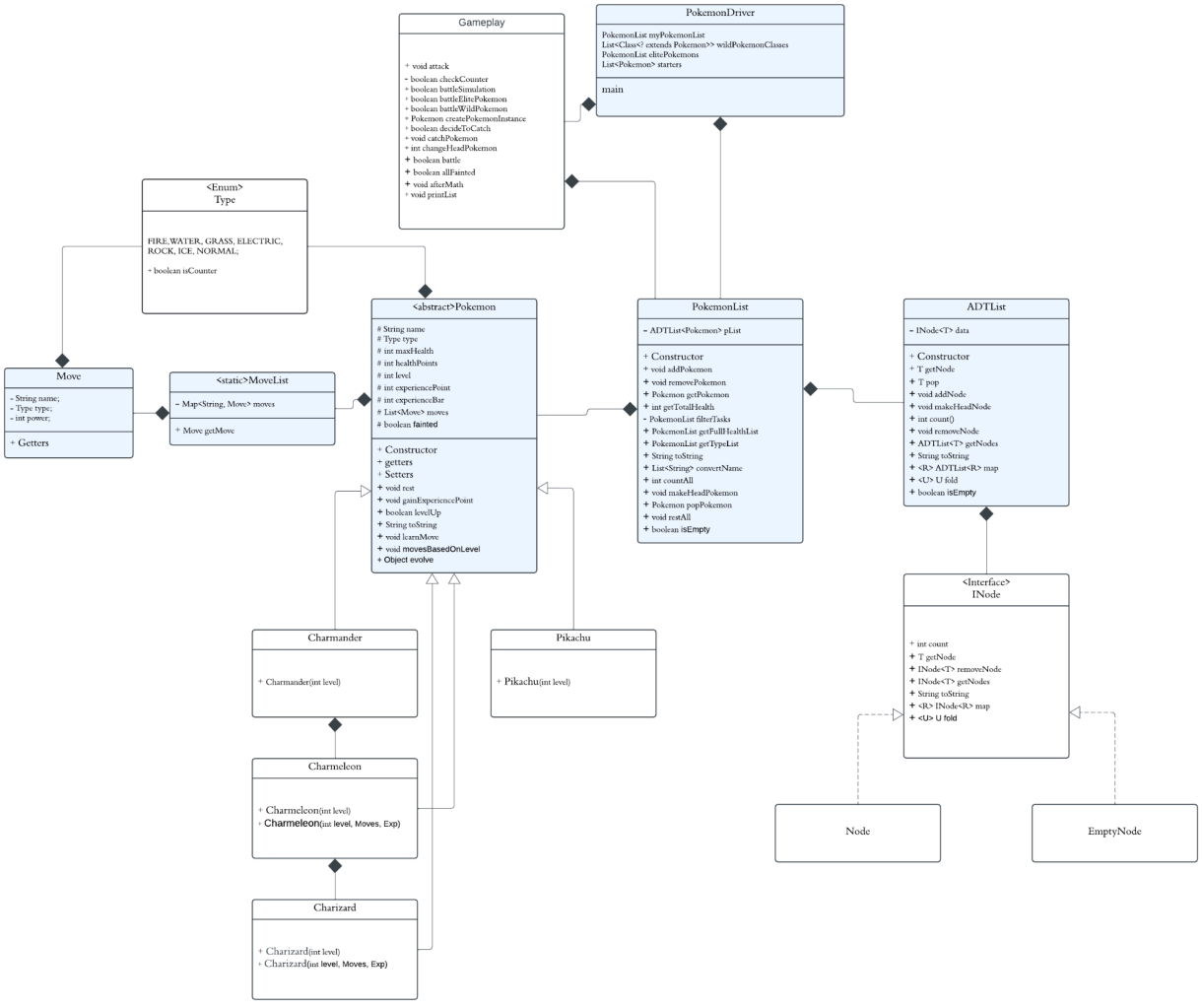
Fan Chen
Final Instruction
CS 5004
4/16/24

Overview:
In this project, I am developing a mini Pokemon game that allows players to engage in various activities typical to the Pokemon game. Players can select a starter Pokemon, battle the wild Pokemon, level up their Pokemon, and eventually face elite Pokemon with higher levels. The game is structured into multiple components, each handling distinct aspects of the game mechanics.

Below is the UML Diagram of this project:



Approach and Organization:

Pokemon Class:
An abstract class that forms the backbone of the game, encapsulating all the necessary attributes and methods that define a Pokemon. It includes properties such as health points, level,type, and other methods.

Key abstract methods in the Pokemon class are movesBasedOnLevel and evolve. Since each Pokemon learns different moves at different levels, movesBasedOnLevel is left abstract to allow for specific implementations in different Pokemon subclasses. Similarly, the evolve method is also abstract because each Pokemon evolves at varying levels into different forms.

Initially, I considered implementing evolve as an interface only for Pokemon capable of evolving. However, during gameplay, all these Pokemon are treated as instances of the generic Pokemon class, which complicates evolution functionality when using driver classes. For example, when adding a Charmander at level 5 to a Pokemon list, it's added as new Charmander(5). However, in a battle scenario where the specific type of Pokemon isn't predetermined, each Pokemon is referred to by the generic type. If evolve were implemented as an interface in the specific Charmander class, a Charmander assigned as a generic Pokemon wouldn’t be able to evolve in the driver class due to type abstraction.

To resolve this, evolve was defined as an abstract method shared across all Pokemon classes. For Pokemon capable of evolving, this method returns the evolved form, while for those that cannot evolve, it returns null.

Pokemon Type Enum:
An enum that defines the various types of Pokemon, such as Water, Fire, Grass, etc. This class influences the strengths and weaknesses of Pokemon in battles and is used with both Pokemon and Move classes.

Within this enum, I implemented the isCounter method, which assesses whether one type effectively counters another and returns a Boolean value. This method works in conjunction with the attack method in the Pokemon class to calculate the final damage dealt. It takes into account both the move's type and the type of the Pokemon being attacked. This integration helps in strategizing attacks based on type matchups, enhancing the tactical depth of battles.

Move Class:
Represents the different moves that Pokemon can learn. Each move has associated properties like power and type, which affect its performance in battles.

Move List:
A utility class that uses a hashmap to map move names to their corresponding Move objects. This allows for dynamic retrieval and assignment of moves to Pokemon.

ADT List Class and Nodes:
An abstract data type that provides a generic list implementation used to manage collections of Pokemon within the game. It supports basic operations such as adding, removing, and accessing elements in the list. The files were used in labs 4 and 5, I added a few methods such as pop and isEmpty for this project.

Pokemon List Class:
Extends ADT List class to specifically manage Pokemon objects. It includes methods tailored to the needs of the game.

Originally, this class included a battle method allowing an opponent's Pokemon to loop through the list and attack the current head Pokemon, and an aftermath method called after a victory to grant experience points and check for potential evolutions for the Pokemon. However, upon further evaluation, I determined that these methods are more closely related to the gameplay mechanics rather than the management of the Pokemon list itself. Therefore, I relocated them to the Pokemon driver class as static methods. This adjustment helps maintain the separation of concerns, keeping the Pokemon List Class focused on list management while gameplay-specific functions are handled by the driver.

In addition to its primary functions, this class includes several methods, such as filter methods, which were not used in actual gameplay. These methods were implemented primarily to demonstrate the mapping, filtering, and folding as part of the project. While these methods do not directly impact gameplay, they serve as examples of data manipulation techniques.

Gameplay Class:
This class functions as a central hub for managing the various mechanics and interactions within the game, particularly through the driver. It was introduced after recognizing that the driver file was becoming overly complex with numerous helper methods and print method are used all over files. By abstracting away these methods into the Gameplay class, the structure achieves a clear separation of concerns. This class effectively organizes and streamlines the interaction and functionality required for engaging and dynamic gameplay, simplifying the main driver's role to basic game flow control.

Pokemon Driver:

The Pokemon Driver acts as the main entry point for the application, controlling the game flow and linking the game logic with the user interface. It is responsible for initiating the game setup and managing transitions between different phases of the game.

After abstracting all the helper methods to the Gameplay class left the driver with just the core responsibilities of initializing game setups and managing the main game loop. As a result, the driver now serves a streamlined role, focusing primarily on launching the game and ensuring smooth progression through its various stages, thereby enhancing the overall structure and readability of the code.

Conclusion:
In addition to the techniques taught in class, this project provided me with the opportunity to learn several other programming methods on my own. Key learning highlights include the use of various data structures for storing data, the exploration of Java's reflection mechanism, and the understanding of the pros and cons associated with different design choices.

For data storage, I utilized different structures tailored to specific needs within the game. HashMaps were employed to store moves, allowing easy assignment of moves to different Pokemon. ArrayLists and a fixed-size array initialized with the asList method were used for starter Pokemon, due to their fixed quantity. Abstract Data Type (ADT) List was chosen for the elite Pokemon list to facilitate accessing the current opponent. For myPokemonList, I choose the ADT list over an ArrayList, to demonstrate the concepts for this assignment, despite the ArrayList is also a good option.

A particularly interesting aspect of the project was the storage and instantiation of wild Pokemon using Java's reflection mechanism. This approach allows for the random selection and creation of Pokemon during gameplay. By leveraging reflection, the game dynamically instantiates Pokemon objects based on their class type and specified level at runtime.

Throughout the development of this project, I was faced with numerous design decisions that significantly shaped the final implementation. These choices included selecting appropriate data structures to store different game elements, abstracting functionalities from the driver file to a separate Gameplay class, and picked abstract method for Pokemon evolution rather than utilizing an interface. Each decision was made in response to evolving requirements and challenges encountered during the implementation phase. Although these choices sometimes strayed from the original design outlined in the UML diagram, they were necessary for practical and functional reasons. Each decision brought its own advantages and trade-offs, underscoring the importance of adaptability in software development.

Concept Map:

Recursion in Practice - a logical use of recursion that simplifies your code	I implemented the concept in the ADTList and its Nodes. All methods in the INode interface utilize recursive calls to perform list operations, thereby simplifying the code structure and logic.
Logical Structure and Design using Abstract Classes and Interfaces	I implemented the concept in the abstract Pokemon class. Evolution is handled by an abstract method, ensuring consistency across all Pokemon. Refer to Pokemon.java on line 160 for the evolve and movesBaseOnLevel methods. INode and IView are both interfaces. INode serves as the parent for Node and EmptyNode. The MVC pattern is demonstrated in the MVC file for the starter picking process.
Useful and Logical Abstraction using Generics and Lambda Expressions	I apply this concept in PokemonList and Gameplay classes. See PokemonList.java on line 10, ADTList<Pokemon> was created that can only store Pokemon class and Gameplay.java on line 97 for reflective instantiation wild pokemon with Generics. I demonstrated Lambda Expressions in the PokemonList Class line 27 to get the total health of the list for the fold method implementation, and other filter methods which I didn't add to the final game.
Higher Order Functions Map, Filter, and Fold	All these three functions are demonstrated in the PokemonList Class. Map was used on line 54 to get a list of pokemon names. Filtering is illustrated but not used in gameplay; on lines 30-47. Fold was used on line 27 to get the total health.
Hierarchical Data Representation as an ADT or a Linked List ADT (Whatever makes the most sense for your application)	I applied the ADT linked list concept on PokemonList and was demonstrated on the PokemonDriver Class on lines 12 and 21 as myPokemonList and elitePokemons.
Architectures and Design Patterns MVC	I demonstrate this concept in the MVC file.

Design and a design pattern	Where I recreated the starter picking process in MVC design.
SOLID Design Principles	Throughout the project, SOLID principles guided the design and refactoring process. Specific principles such as Single Responsibility and Dependency Inversion are evident in the separation of game logic from the PokemonList and the use of interfaces for nodes.
Extension Concept: Reflect Constructor	I demonstrated this concept in Gameplay class line 95. The Constructor<? extends Pokemon> lets me construct a Pokemon subclass object.
Extension Concept: Special Evolve	I demonstrated this concept in the Gameplay class line 90. The if statement lets Slowpoke evolve into Slowbro after defeating Shellder

Acknowledgements and assistance received:

I did not use generative AI in any form to create this content and the final content was not adapted from generative AI created content.

I did not view content from any one else's submission including submissions from previous semesters nor am I submitting someone else's previous work in part or in whole.

I am the only creator for this content. All sections are my work and no one else's with the exception being any starter content provided by the instructor. If asked to explain any part of this content, I will be able to.

By putting your name and date here you acknowledge that all of the above is true and you acknowledge that lying on this form is a violation of academic integrity and will result in no credit on this assignment and possible further repercussions as determined by the Khoury Academic Integrity Committee.

Name:Fan Chen	Date: 4/16/24