Final Project

420-3P5-AB

The Fire Nation

Samuel Anderson #1360134

Jeffrey Bringolf #2075606

A picture containing graphical user interface

Description automatically generated

* A full pokemon box, this is where all the pokemon are stored and selected.

# Project Description

The goal of this project is to develop a simple WPF application. The app should use OOP concepts learned in class and demonstrate the knowledge we attained throughout the semester. In addition, it should have saving and loading from text-based documents to demonstrate an understanding of file IO. One additional requirement is the application must have multiple windows.

We chose to make a pc from the Pokémon games. A pc in the world of Pokémon is simply a container to store your Pokémon. We realized this was a perfect candidate for the project due to the object-oriented nature of the device. Not only is the pc itself an object that could be turned into a class, but so are the Pokémon. We had to stray from the exact functionality of the pc found in games for the purpose of the assignment. Ours has the following features:

* Select a Pokémon from a list of options and customize the Pokémon’s name, sex, shininess, move set and ability.
* Add the custom Pokémon to the pc’s storage section and select a Pokémon to display additional information.
* Edit a Pokémon once it has been added to the box.
* Release a Pokémon, removing it from the pc.
* Save your box to a csv file.
* Load your box from a csv file.
* Music and sounds

# Development Approach

To start off, we needed a concept. We knew it should be object oriented, as was the point of the project. The proposed project was the inventory tracker which seemed a bit bland since we both wanted to add our own spin on the project. Immediately our minds went to video games, since they are well known for generally using OOP. Now that we had the idea of video games, and the concept of an inventory tracker the concept was clear, a pc from Pokémon. Not only would this give us a clear path to OOP, but it would also be easy to make it aesthetically pleasing.

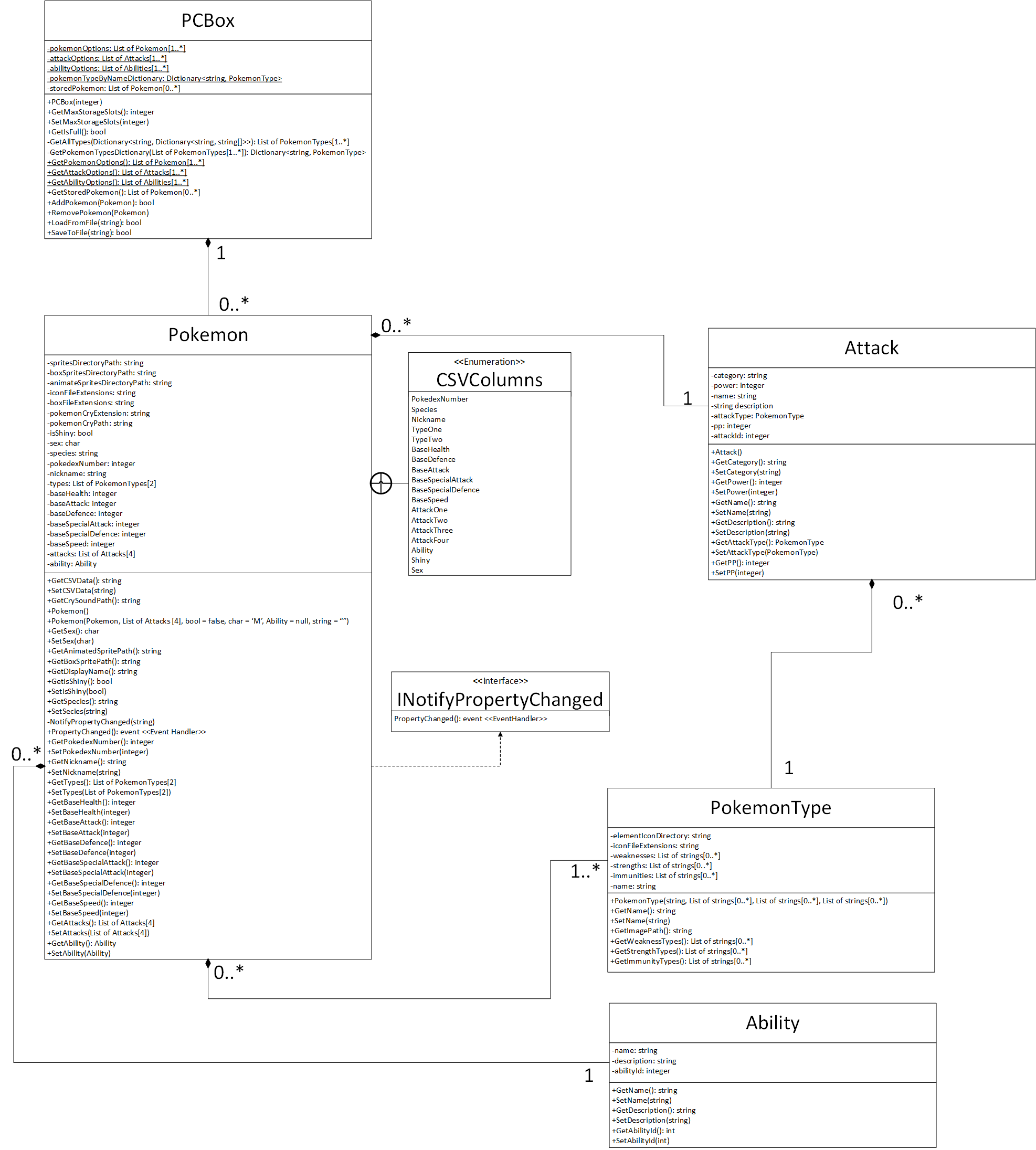
Once we had the concept, we began designing how the classes would be created. Thankfully, we based our project on an already existing program, so we had a clear path on how to implement it. We would need a class for the pc to be able to store the Pokémon and a class for the Pokémon themselves. The pc would have a list of Pokémon being stored and some methods to add functionality. The Pokémon however were more complicated to implement. In the games each Pokémon have some standard properties (stats, species, sex, …) however they also have more complicated properties; namely attacks, abilities and types. In game, these are used mainly in combat to add more interesting interactions between Pokémon, but our application didn’t have combat. We had to decide whether to implement these fully as classes, or to simply add them as strings in the Pokémon class and ignore their larger purposes. We decided to include the functionality for these three properties and make classes for them.

When developing the application, we decided to create the classes first; that way we would immediately be able to implement binding when creating the front end. The most logical way to start was to create the Pokémon class. We added the basic properties and then the more complex ones. The Pokémon would use composition with the PokemonType, Ability and Attack classes. The PokemonType class was the trickiest, since we would use it would be found in the Pokémon class, and the attack class. This seemed simple until we realized that an attack of one type, does not share the same properties as a Pokémon of the same type. This meant that we would need to create two instances of the PokemonType class for each type, one for the Pokémon, and one for the attacks. We debated having several type classes, (i.e., Poison, grass, water, …) all inheriting from one base PokemonType class, but then realized that the point of having a class is to have a template, which we had in the PokemonType class already. Rather than making different derived classes for each type, we would simply instantiate several PokemonType objects with differing values dependent on the type. This way, since the objects are never changing, we could just make a bunch of PokemonType objects and pass them to the Pokémon or attacks that use them without worrying that the properties in the PokemonType object would be manipulated. The ability and Attacks were simpler, they would just have data for them loaded from a file and passed to Pokémon that use them.

Implementing the application was simple, since we already had our classes designed, we simply had to translate what we had imagined, into code. We made the all the classes without much issue. The biggest issue we ran into was the loading from files. Since we were using pre-existing Pokémon, we needed to load the data for the object from a file. The issue was that the type was being loaded from the Pokémon data file as a string, but we had no way of turning that string into a type to add to the Pokémon. To solve this issue, we created a dictionary that has key value pairs of string: Type Name, PokemonType: type. This way we could read the type as a string and get the appropriate Type for the Pokémon.

Finally for testing, we tried all the edge cases we could think of, as well as some generic cases. We tried saving an empty box, which works, since we want to user to be able to save immediately to pick their save location. We tried loading invalid csv files, which threw errors and displayed them to the screen. We tried adding to the box with a full box, pressing all the buttons without a selected Pokémon, closing the window without saving, and several other simple functions to ensure everything worked smoothly.

# OOP Design



## PCBox

Text

Description automatically generated with medium confidence

The PC Box is the main storage system. It contains all the functionality to do with adding, deleting, loading and saving the contents of the box. Its static backing fields are the possible options for Pokémon, attacks and abilities, as well as a dictionary used to get PokemonType objects from a string. These options are shared amongst all instances of PCBox, so they were made static. Their accessors are also static. They are used in our application to set the items sources of the combo boxes in the Addition window and the Edit window.

The rest of the methods are used for storage and saving, there is no direct mutator for the storedPokemon backing field since only the PCBox object should be able to manipulate the contents in any way it wants. If another object wants to manipulate the contents it can use the AddPokemon or RemovePokemon methods. There are also LoadFromFile and SaveToFile methods which use the static load and save methods contained in the static DataReaderWriter class.

A picture containing outdoor, mounted, light, bright

Description automatically generated

* The PCBox represents the stored pokemon that you can select to edit, release or see more details about.

## Ability

Table

Description automatically generated

The ability is the simplest class, it is used as a property of Pokemon (Composition). It contains simple data fields like the strings name and description, and the integer abilityId. These are used for display.

Graphical user interface, application

Description automatically generated

* Abilities can be selected when creating or editing a pokemon and describe a special ability pokemon can use in or outside of battle.

## Pokemon

Graphical user interface

Description automatically generated with low confidence Text

Description automatically generated with low confidence

The Pokemon Class is the largest class in UML but, it isn’t very complicated. It only appears very large because there are lots of auto implemented properties which create an accessor and a mutator for each. The interesting Methods are the constructor, Get\*SpritePaths, PropertyChanged, NotifyPropertyChanged and SetIsShiny.

The constructors are simple, the default one exists to be able to use object initialization syntax and used mostly to copy pokemon. Since Pokemon are reference types, any time you want to create a copy of a pokemon, you need to instantiate one from scratch. This is extremely important since the addition window and the edit window allow you to manipulate preexisting pokemon. In the case of the addition window, the pokemon you’re adding begins as one of the preset pokemon, so without copying, you could never make more than one of the same species without editing the contents of all other pokemon sharing that species.

The Get\*SpritePaths, PropertyChanged and SetIsShiny methods are all related. All the Get\*SpritePath methods are calculated. Since they have no setter, the front end is never notified that there has been a change to the property’s value, so the binding doesn’t work as intended. To fix this, we need to call the invoke the event ourselves. To invoke the event, we need to implement the INotifyPropertyChanged interface. This is where the PropertyChanged event handler comes from. The sprite paths are reliant on only the shiny value of the pokemon, since there are different images for shiny or regular. So, we call the NotifyPropertyChanged function with the name of the property that has been changed to invoke the event and sync the binding.

A screenshot of a video game

Description automatically generated with medium confidence

* Pokemon can be picked from several species. Each species has their own stats and types. You get to pick their name, their sex, attacks, etc.

## PokemonType

Graphical user interface, text, application

Description automatically generated

The pokemon type demonstrates the type of a pokemon or an attack. The type contains weaknesses and strengths. In a pokemon battle in the real game, if you use a specific move on a specific type of pokemon it does more or less damage depending on if the opposing type is weak, strong, neutral, or immune to the other type. We don’t have any interactions between types in our program, but the logic is there regardless, meaning it would be relatively easy to implement a battling window with the logic we have right now. The only methods in this class are accessors and mutators. The image path is used to display the image associated with this type. It also has one constructor that takes the type’s name, as well as all the names of the types it’s weak to, strong to, or immune to.

A picture containing graphical user interface

Description automatically generated

* Pokemon can have one or two types which determine how well it matches up against other pokemon and attacks.

## Attack

Table

Description automatically generated

Attack is similar to ability, in that its only purpose in our application is aesthetic, however it would be used in combat in a real pokemon game. Each pokemon must contain at least one attack to be added to the box. The attack class contains simple properties like its name, description, pp (number of uses), an ID to match data loaded from a file, its power, and accuracy. It also contains a PokemonType (aggregation) which would be used in combat (as described above in the PokemonType class).

Graphical user interface, text, application

Description automatically generated

* Each pokemon has 1 to 4 attacks. Like pokemon, attacks have types and are effective and weak against certain other types. Unlike pokemon, attacks only have one type

## DataReaderWriter

Diagram

Description automatically generated

The DataReaderWriter is a static class that is used to read and write data from files. The Enums are used to help with the reading, they represent the columns of the csv files being read.

* GetPokemonOptions returns an array of pokemon from a csv file. These are used as the options of the combo boxes on the addition and edit pages.
* GetTypesInformation returns a dictionary. This dictionary’s keys are the names of all the possible PokemonTypes, and it returns another dictionary. This second dictionary’s keys are “Weak”, “Effective”, and “Immune” and returns a list of strings corresponding to the PokemonTypes that area weak, effecting or immune against the PokemonType input as the first key.
* GetAttackOptions does the same thing as GetPokemonOptions but for attacks.
* GetAbilityOptions does the same thing as GetPokemonOptions but without the need to a type dictionary, since abilities don’t have types.
* LoadBoxFromFile reads a csv file and tries to create a list of pokemon from the file, it returns the list if it read successfully, or returns null if there was an error.
* SaveBox saves the pokemon passed to it to the specified file.

# Contributions

## Jeffrey

* Did most of the backend.
* Created the models and most of the code in the .xaml.cs files.
* Wrote the project document and made the UML diagrams.
* Wrote python scripts to format csv files from the internet to meet the purposes of our app.
* Got music and sounds working

## Sam

* Did all the front-end development.
* Added functionality to the application like maximizing, minimizing.
* Got all the images from the internet and got them working with the application.
* Got the icon.
* Got all the music and sounds from the internet.

## Shared

* Concept development
* Testing
* Object Oriented Decomposition

# Future Work

There are several features we wanted to add but didn’t get the chance to.

One idea was the ability to see what types you are effective and weak against. Since all the type relationships are already in the classes, it would have been quite simple, but we didn’t think it was worth the effort. The idea would have been to be able to select a pokemon and display all the types that would be super effective, weak, or do no damage to it based on its single or dual typing.

Another feature we didn’t get the chance to add was the ability to sort selections by certain conditions. For example, there are 151 pokemon and they are not ordered in alphabetic order. This makes it quite difficult to find the one you’re looking for if you aren’t familiar with the PokeDex order. It would be nice to be able to show only species of a specific type, or maybe sort by name. This feature would be especially nice for the abilities and attacks which have around 500 options.

Finally, the obvious next step would be to add a combat system. We actually considered adding this but didn’t have the time to allocate to it. It would be simpler to add than you might expect since we already have all the attacks, types, and pokemon stats. We would just need to add a loop for the turned base combat and add an attack method to the pokemon class.