Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e Multimédia (LEIM)

Modelação e Programação (MP) – 20/21

Projecto Final – Parte A

Trabalho realizado por:	número
Miguel Ginga	46292

Lisboa, 9 de Junho de 2021

Índice de matérias

1. Introdução 5
2. Apresentação da Aplicação 6
2.1. Treinadores 7
2.2. Pokémon 7
2.3. Ataques de Pokémon (Pokémon *Moves*) 7
2.4. Tipos de Pokémon (Pokémon *Types*) 7
2.5. Tipos de Status (Pokémon *Status*) 8
3.Diagrama UML de Classes 9
4.Classe Pokémon 11
5. PokemonType e TypeManager 13
6. PokemonMove, DamageMove, StatusMove, MultiplierMove 15
7. Trainer, PlayerTrainer e AITrainer 17
8. PokemonPool 20
9. Battle 21
10. Conclusões 23

Índice de Imagens

Figura 1 – Exemplo de uma batalha no jogo Pokemon Red 6
Figura 2 – Diagram UML das classes do package 9
Figura 3 – Diagrama UML da Classe Pokémon 11
Figura 4 – Diagrama UML do Enumeration PokemonType e Classe TypeManager 13
Figura 5 – Exemplo de uma lista moveEffectiveness 13
Figura 6 – Diagrama da Interface PokemonMove 15
Figura 7 – Diagrama da classe DamageMove 15
Figura 8 – Fórmula dos jogos 15
Figura 8– Diagrama da classe StatusMove 16
Figura 9– Exemplo de um turno em que é aplicado o efeito POISON 16
Figura 10– Diagrama da classe MultiplierMove 17
Figura 11– Diagrama da Interface Trainer 17
Figura 12– Diagrama da classe PlayerTrainer 18
Figura 13– Diagrama da classe AITrainer 18
Figura 14– Método chooseActivePokémon da classe AITrainer 19
Figura 15– Diagrama da classe PokemonPool 20
Figura 16– Exemplo da criação e adição de um Pokémon à PokémonPool 20
Figura 17– Diagrama da classe Battle 21

1. Introdução

No âmbito da unidade curricular Modelação e Programação, foi nos proposto como trabalho final o desenvolvimento de uma aplicação com interface gráfica. Neste trabalho devemos aplicar e consolidar os conhecimentos desenvolvidos ao longo de todo o semestre. O projecto está dividido em duas partes:

- Parte A - Desenvolvimento da aplicação em Java em modo de texto/consola.
- Parte B – Desenvolvimento da interface gráfica para aplicação desenvolvida na parte A.

Este relatório é exclusivo à Parte A do projecto, pelo que irá abordar todo o desenvolvimento da aplicação, desde os diagramas UML ao desenvolvimento das classes. Irá ser descrito quais as finalidades da aplicação, regras para o funcionamento da mesma e os passos dados para o desenvolvimento a aplicação final.

2. Apresentação da Aplicação

A aplicação desenvolvida é um simulador de batalha dos famosos jogos da série Pokémon. Os jogos Pokémon, entre outros aspectos, focam-se em batalhas entre dois treinadores que têm os seus Pokémon de escolha e podem escolher entre várias opções como escolher um ataque ou mudar de Pokémon. A batalha funciona por turnos, um Pokémon ataca o outro e vice-versa, a ordem de quem ataca primeiro é definida pela velocidade dos Pokémon, sendo que o mais rápido naturalmente irá atacar primeiro. Um Pokémon tem pontos de vida (HP) e quando estes chegam a 0 o Pokémon desmaia (*fainted* é o termo utilizado nos jogos). A batalha termina quando um treinador não tiver nenhum Pokémon capaz de lutar, ou seja, quando todos os Pokémon de um treinador tiverem 0 pontos de vida.

Figura 1 – Exemplo de uma batalha no jogo Pokemon Red

Existem diversas variáveis num jogo de Pokémon que terão de ser simplificadas devido à gigantesca escala da série. Um dos exemplos é que actualmente existem mais de 800 Pokémon, para o desenvolvimento desta aplicação iremos utilizar uma lista de 10 Pokémon, pois a criação de um requer obtenção de vários dados como iremos ver adiante na criação de um objecto da classe Pokémon. Outro exemplo é que um treinador pode ter até 6 Pokémon, mas em cenários de batalhas competitivas ou no *end-game*(conteúdo final dos jogos) as batalhas são realizadas com apenas 3 Pokémon de cada lado, sendo esse o modelo que vai ser perseguido, para que as batalhas não sejam desnecessariamente longas e para que o utilizador possa experienciar com mais combinações de Pokémon.

Tal como descrito na introdução deste projecto, terão de existir regras que terão de ser cumpridas para dar consistência e coerência à aplicação.

2.1. Treinadores

Um treinador de Pokémon terá de ter exactamente 3 Pokémon. O jogador irá escolher da lista disponível 3 Pokémon não sendo possível dar início a uma batalha enquanto esta operação não estiver concluída. Por sua vez o adversário (computador) irá escolher 3 Pokémon aleatoriamente.

2.2. Pokémon

Um Pokémon terá de ter:

- Nome
- Necessariamente entre um a dois tipos associados (mais sobre isto adiante)
- Valores para defesa, ataque e velocidade, todos superiores a 0
- Uma lista com exactamente 4 ataques (ou *moves*)

2.3. Ataques de Pokémon (Pokémon *Moves*)

Tal como referido em cima, um Pokémon terá de ter 4 ataques que podem variar em tipo, podem ser ataques que causa dano (podendo também causar *status*), ataques que causem unicamente *status* ou ataques que modifiquem as estatísticas do Pokémon (ataque, defesa, velocidade). Um move também terá um número de usos possíveis.

2.4. Tipos de Pokémon (Pokémon *Types*)

Os Pokémon podem ter um ou dois tipos associados, enquanto que os ataques têm necessariamente de ter exactamente um tipo associado. Os tipos num Pokémon definem a que tipo de ataques são mais vulneráveis, mais resistentes e até imunes. Existem 17 tipos de Pokémon:

FIRE, ***GRASS***, ***WATER***, ***NORMAL***, ***FLYING***, ***ELECTRIC***, ***ICE***, ***FIGHTING***, ***POISON***, ***GROUND***, ***PSYCHIC***, ***BUG***, ***ROCK***, ***GHOST***, ***DRAGON***, ***DARK***, ***STEEL***, ***FAIRY.***

2.5. Tipos de Status (Pokémon *Status*)

Os Pokémon podem ter Status associados, estes Status são causados pelos ataques. Os tipos de Status que existem são:

- Paralyzed – o Pokémon tem uma chance do início do seu turno ficar paralisado e não executar qualquer acção e a sua velocidade é diminuída em 50%.
- Poisoned – o Pokémon no fim do turno perde uma percentagem dos seus pontos de vida.
- Burned – o Pokémon no fim do turno perde uma percentagem dos seus pontos de vida e o seu ataque é diminuído em 50%.
- Sleep – o Pokémon não pode atacar, mas tem uma chance de acordar todos os turnos sendo que após 3 turnos acorda.
- Freeze – o Pokémon não pode atacar, mas todos os turnos tem uma chance de se descongelar.

3.Diagrama UML de Classes

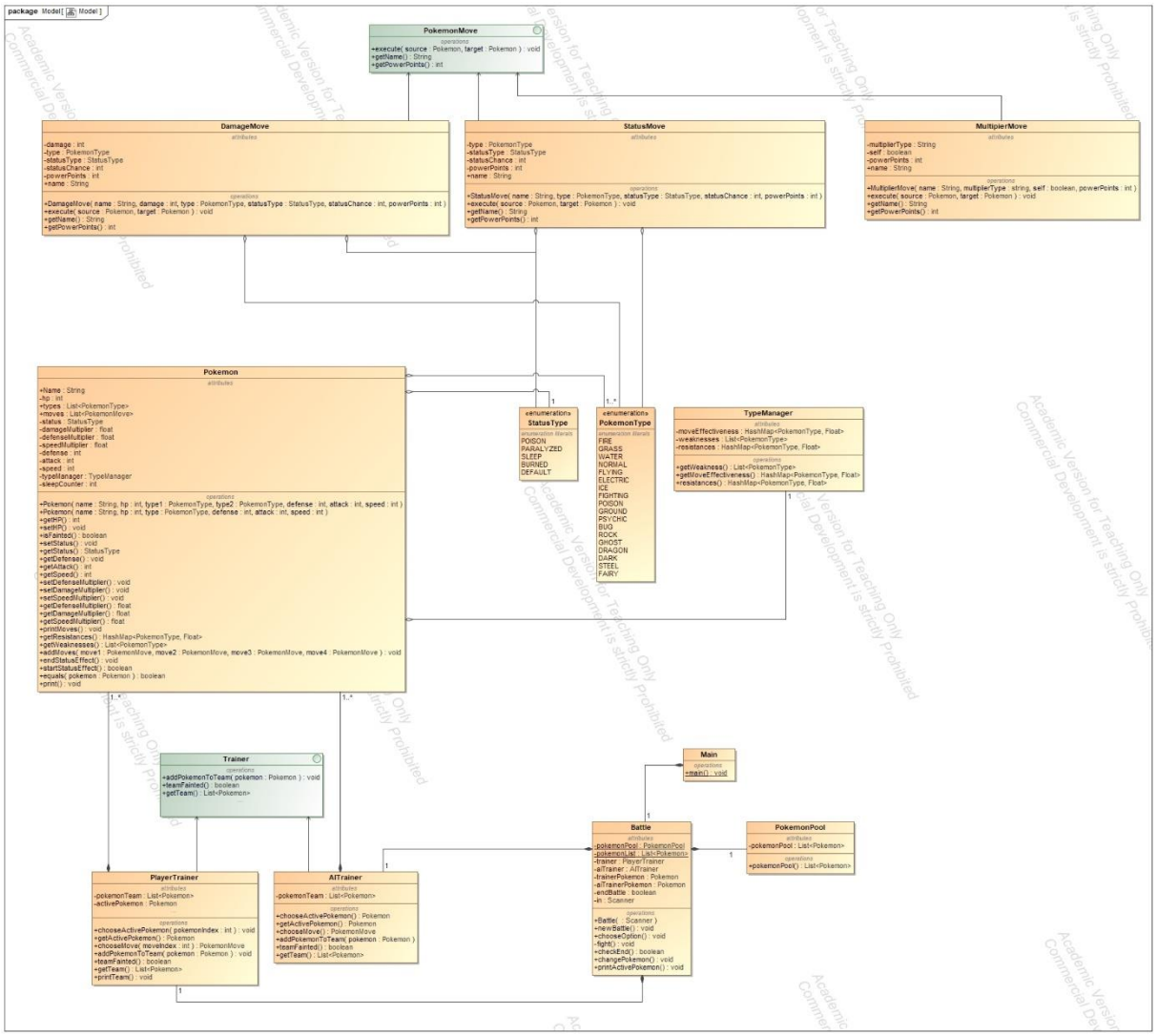

Figura 2 – Diagram UML das classes do package

No diagrama da figura 2 podemos observar quais as classes do nosso projecto e como estas se relacionam entre si. Começamos pela classe *Pokemon*, que é a classe central do nosso projecto. Esta classe relaciona-se com *StatusType*, *PokémonType* e *TypeManager*. *StatusType* e *PokemonType* são enumerados que contêm respectivamente todos os tipos de status e todos os tipos de Pokémon. *TypeManager* é responsável por gerir os tipos de Pokémon. Por fim um Pokémon relaciona-se com *PokémonMove*, podendo ter um ou mais objectos desta classe.

A interface *PokemonMove* é implementada por 3 classes : *DamageMove*, *StatusMove*, *MultiplierMove*. *DamageMove* e *StatusMove* têm uma agregação fraca com *StatusType* e *PokemonType*.

Por sua vez a interface *Trainer* é implementa por *PlayerTrainer* e *AITrainer*. Estas são as classes responsáveis pelos treinadores. Estas duas classes relacionam-se com Pokémon podendo ter um ou mais objectos desta classe. A classe *Battle*, responsável por gerir e controlar uma batalha, relaciona-se com *PlayerTrainer* e *AITrainer* tendo de conter unicamente 1 objecto de cada classe. *Battle* relaciona-se ainda com *PokemonPool*(classe responsável por conter a lista de todos os Pokémon), devendo conter 1 ou mais objectos desta classe.

4.Classe Pokémon

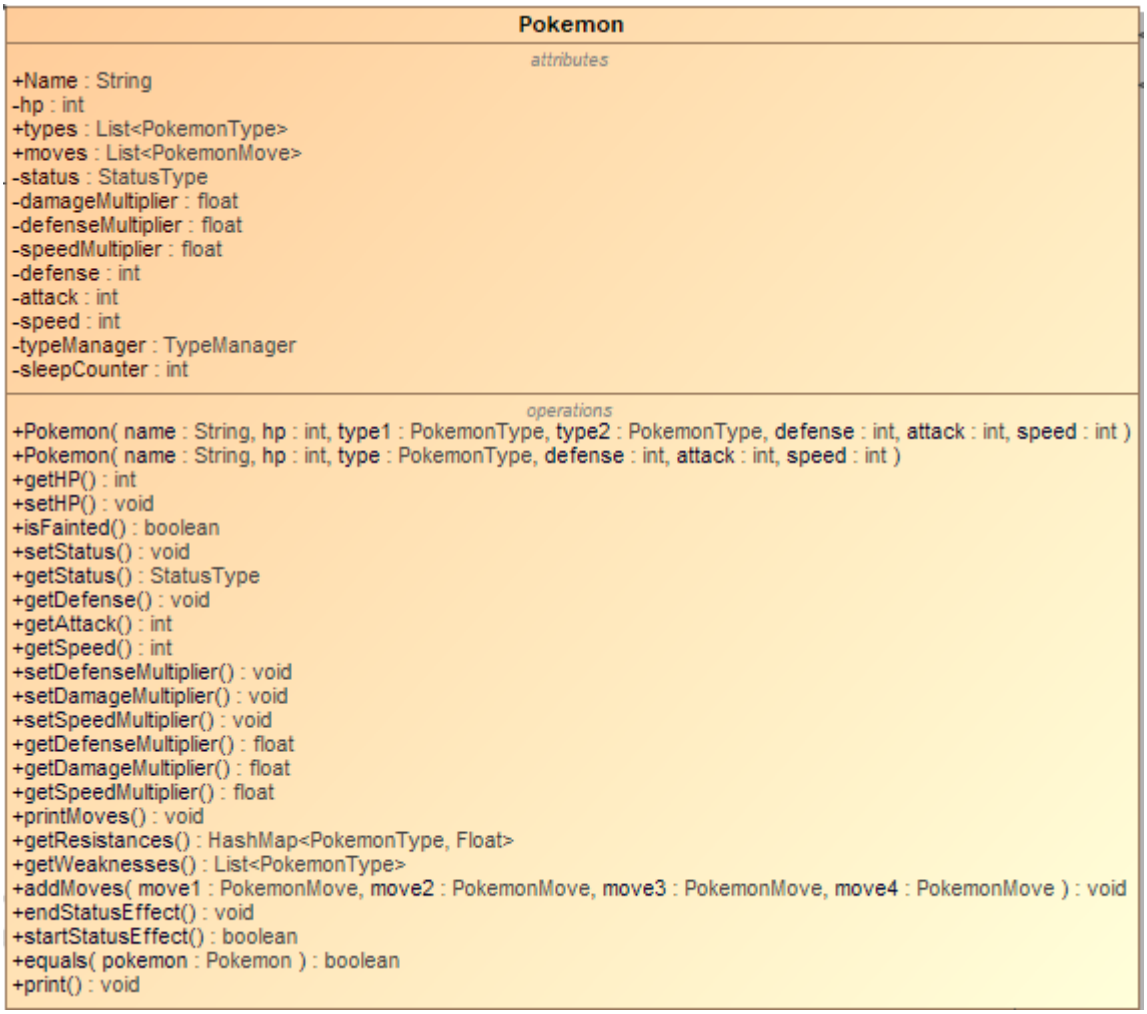

Figura 3 – Diagrama UML da Classe Pokémon

A classe Pokemon é sem dúvida a mais extensa em termos de métodos de toda a aplicação, sendo que esta é o pilar sobre o qual toda a aplicação revolve. Serão usados dois construtores, um para um Pokémon que apenas tenha um tipo, e outro para os que tenham dois tipos. Isto é feito porque com o modelo provisório utilizado, era usado como argumento no construtor a lista com os tipos, isto implicava que era necessário criar uma lista para cada Pokémon e adicionar a lista como argumento. Assim evitamos esta situação de criar várias listas, recebendo apenas como argumento os tipos que serão logo adicionados à lista, que está definida como *final* pelo que não poderá ser alterada posteriormente.

Ainda nos construtores são lançadas excepções caso o seguinte se verifique:

- Se o argumento hp for negativo.
- Se qualquer um dos argumentos defense, attack, speed forem negativos.

Um motivo pelo qual esta classe é tão extensa é devido à necessidade de métodos que obtenham e manipulem os valores de todos os dados referentes a um Pokémon, sendo estes: *hp*, *status*, *attack*, *defense*, *speed*, *damageMultiplier*, *defenseMultiplier*, *speedMultiplier*.

Para verificarmos se um Pokémon ainda pode lutar, temos de verificar se os seus pontos de vida são superiores a 0 e fazemos isto com recurso ao método *isFainted*.

O método *printMoves* tal como o nome indica irá dar print na consola à lista de ataques do Pokémon. Os ataques são adicionados através do método *addMoves* que recebe como argumentos 4 *PokémonMove* que podem variar entre as classes derivadas desta. Neste método verificamos se a lista *moves* se econtra vazia. Caso esta não se encontre vazia, é lançada uma excepção pois um Pokémon apenas pode ter quatro *PokemonMove*.

Para sabermos as fraquezas e resistências de um Pokémon utilizamos os métodos *getResistances* e *getWeaknesses* que recorrem ao objecto da classe *TypeManager* (mais sobre esta classe mais à frente).

O status de um Pokémon pode ter efeitos no início ou no fim de um turno, portanto temos de separar estes efeitos em dois métodos que irão ser corridos exactamente no inicio e fim do turno. No início do turno iremos correr o método *startStatusEffect* e no fim do turno iremos correr o método *endStatusEffect*. O *sleepCounter* é usado aqui para que quando o Pokémon estiver com o status *SLEEP*, acorde quando o *sleepCounter* for 3.

Para compararmos dois objectos da classe Pokémon iremos utilizar o método *equals* que recebe como argumento um objecto da classe Pokémon. Um objecto da classe Pokémon é igual a outro se os seus nomes coincidirem.

O *print* de um objecto Pokémon consiste em escrever na consola o seu nome e pontos de vida e ainda o seu status caso não seja o caso por definição.

Embora seja uma classe muita extensa em termos de métodos, é uma classe bastante directa e simples visto que muitos destes métodos são "*getters*" e "*setters*".

5. PokemonType e TypeManager

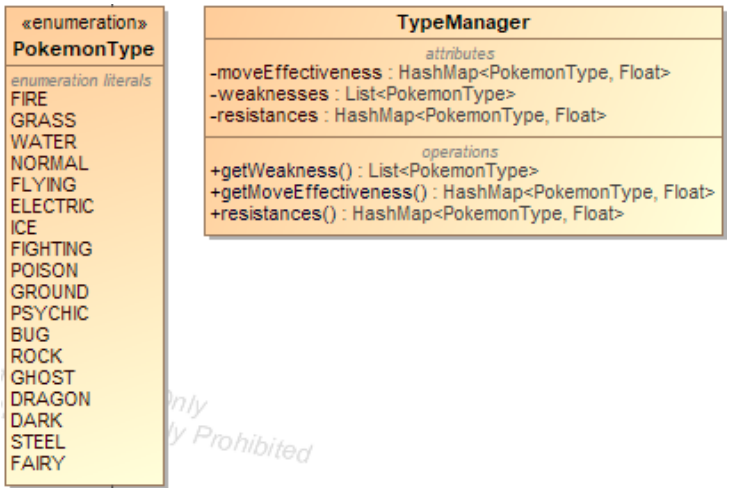

Figura 4 – Diagrama UML do Enumeration PokemonType e Classe TypeManager

Os tipos de Pokémon são valores que queremos que sejam iguais e inalteráveis ao longo de todas as suas instâncias, logo a melhor maneira para definirmos, isto é, recorrendo a um Enumerador. No enumerador *PokemonType* definimos apenas todas as suas instâncias possíveis, sendo estas os 17 tipos de Pokémon possíveis.

Como referido anteriormente cada tipo terá as suas fraquezas, resistências e tipos contra os quais são mais fortes. Para fazermos esta gestão recorremos à classe *TypeManager*. Com esta classe poderemos obter 3 listas:

- *moveEffectiveness* – tipos contra os quais o tipo em questão é mais eficaz.
- *weaknesses* – tipos contras os quais o tipo em questão é vulnerável.
- *resistances* – tipos contra os quais o tipo tem resistência.

Para *moveEffectiveness* e *resistances* queremos ter associado valores aos tipos. Por exemplo, um ataque do tipo *FIRE* faz o dobro do dano a um Pokémon do tipo *GRASS* mas apenas faz metade do dano a um Pokémon do tipo *WATER*. Para alcançar este propósito usámos HashMap's. HashMap é um tipo de lista que nos permite guardar um túpulo em que o primeiro valor é a chave e o segundo valor é um objecto de qualquer tipo que queiramos associar à nossa chave.

```
case NORMAL:
    moveEffectiveness.put(PokemonType.ROCK, 0.5f);
    moveEffectiveness.put(PokemonType.STEEL, 0.5f);
    moveEffectiveness.put(PokemonType.GHOST, 0f);
    break;
```

Figura 5 – Exemplo de uma lista moveEffectiveness

Na figura 5 podemos observar que para o tipo *NORMAL*, adicionamos à lista três tipos aos quais um ataque do tipo normal fará valores de dano diferentes. Em qualquer da construção destas listas recebemos um tipo como argumento e percorremos o enumerado de tipos com o uso de um switch case.

6. PokemonMove, DamageMove, StatusMove, MultiplierMove

Figura 6 – Diagrama da Interface PokemonMove

Os ataques dos Pokémon serão definidos através de classes que implementam a interface *PokemonMove*. Esta interface contém 3 métodos de implementação obrigatória nas classes. Serão estes:

- *execute* - que recebe como primeiro argumento o Pokémon que irá executar o ataque e como segundo argumento o Pokémon que irá ser afectado pelo ataque.
- *getName* – obtém o nome do ataque.
- *getPowerPoints* – obtém o número de vezes que o ataque pode ser utilizado.

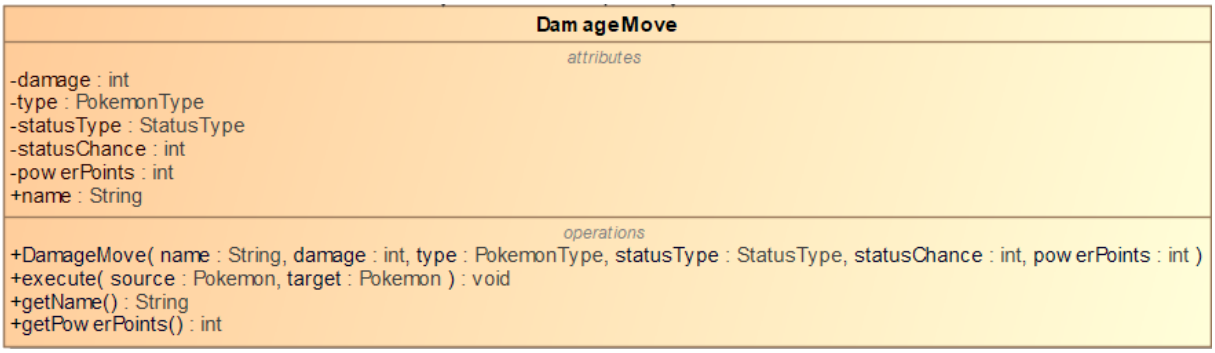

Figura 7 – Diagrama da classe DamageMove

DamageMove é a classe responsável pelos ataques que causam dano. Este tipo de ataque pode causar danos como também aplicar um tipo de status. Como tal, nos argumentos recebe como argumentos o tipo de status de causar, que pode ser o tipo *DEFAULT* e ainda a probabilidade de este status ser aplicado. No método execute, é aplicada uma fórmula adaptada da fórmula original dos jogos.

Calculate the values:

1. $\frac{2 \times Level + 10}{250}$
2. $\frac{Attack}{Defense}$

Multiply the numbers above, then add 2. (Multiply first)

1. $Base \times STAB \times Type \times Critical \times Others \times rand(\in [0.85, 1]$

Figura 8 – Fórmula dos jogos

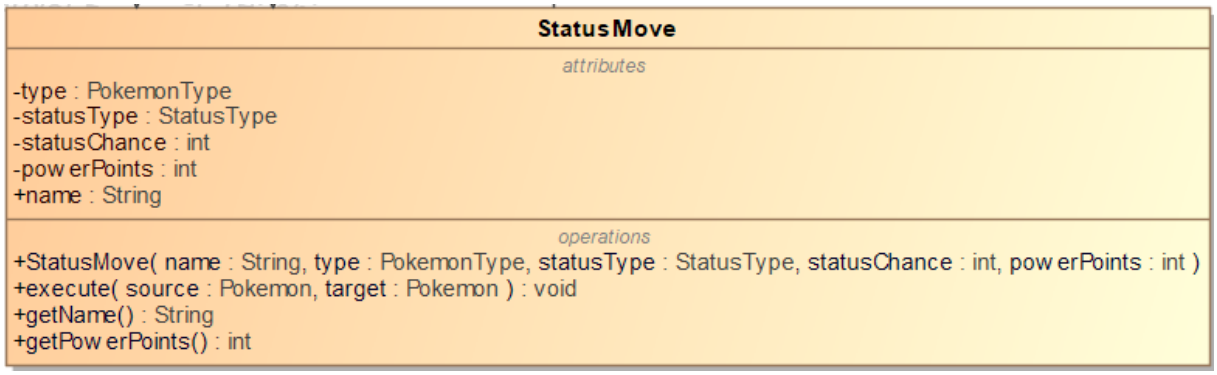

Figura 8– Diagrama da classe StatusMove

StatusMove é usada para criar um move em que a única finalidade é atribuir um *StatusType* ao Pokémon adversário. Como tal, recebe como argumentos o tipo de status e a chance dessa status ser aplicada. O método execute, simplesmente gera um número aleatório e vê se calhar dentro de um determinado valor para que o status seja aplicado. É necessário este método receber de que tipo de Pokémon é porque por exemplo, o ataque "*Toxic*" é do tipo *POISON* e embora apenas seja um ataque de status, não tem efeito em pokémon's com o tipo *STEEL*.

```
Jolteon used Toxic
Espeon has become POISON
----------------------------
Your Pokemon:
Espeon / 144 / POISON

Opponent's Pokemon:
Jolteon / 240
----------------------------

Espeon used Psychic
Espeon did 132 points of damage
----------------------------
Your Pokemon:
Espeon / 144 / POISON

Opponent's Pokemon:
Jolteon / 108
----------------------------

Espeon lost hp due to it's status condition!
----------------------------
Your Pokemon:
Espeon / 126 / POISON

Opponent's Pokemon:
Jolteon / 108
----------------------------
```

Figura 9– Exemplo de um turno em que é aplicado o efeito POISON

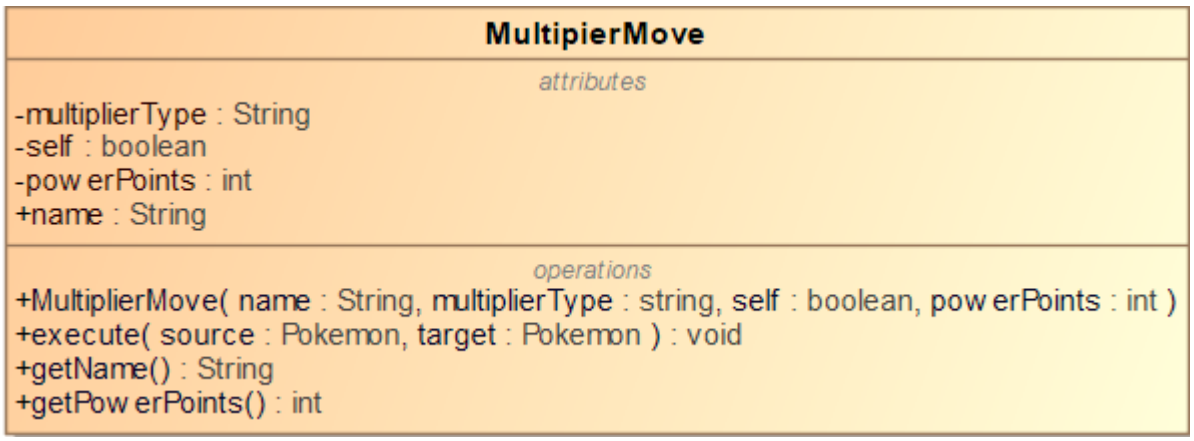

Figura 10– Diagrama da classe MultiplierMove

Por fim, *MultiplierMove* é a classe responsável por ataques que tenham como objectivo alterar o modificador de dano, defesa ou velocidade do próprio Pokémon que o executa ou do Pokémon alvo. Em quem o ataque deve ser usado é definida pela variável boolean *self*, se for *true*, é usada no Pokémon que executou o move e aumenta os seus modificadores. Se for falsa diminui os modificadores do Pokémon adversário. Um determinado modificador só pode ser alterado até a um certo limite, de forma que não hajam casos em que um Pokémon tenha 0 de ataque ou defesa.

7. Trainer, PlayerTrainer e AITrainer

Figura 11– Diagrama da Interface Trainer

A interface *Trainer* irá nos dar as bases para a construção das duas classes de treinador possíveis. Teremos 3 métodos que ambas estas classes irão partilhar:

- *addPokemonToTeam* – método para adicionar um Pokémon para a equipa do *Trainer*
- *teamFainted* – indica se todos os Pokémon da equipa do *trainer* têm 0 pontos de vida ou não.

- *getTeam* – retorna a lista de Pokémon do treinador.

Figura 12– Diagrama da classe PlayerTrainer

Como já referido anteriormente um treinador terá de ter uma equipa de Pokémon definida pela lista *pokemonTeam*. Um treinador a qualquer instante terá também um Pokémon activo em combate, que irá ser guardado na variável *activePokémon*. A classe *PlayerTrainer* é relativa ao utilizador da aplicação. Relativamente aos métodos, adicionalmente aos já implementados pela interface, teremos ainda um método que nos permite definir o Pokémon activo, *chooseActivePokémon*, que recebe o índice da lista onde se encontra o Pokémon pretendido e ainda um método que retorna este mesmo Pokémon activo, *getActivePokémon*.

Será também necessário registar qual o ataque que o treinador pretende que seja executado, sendo este então obtido através do método *chooseMove*, que recebe como argumento o número do índice da lista do ataque que é pretendido que seja usado.

Figura 13– Diagrama da classe AITrainer

AITrainer será a classe responsável pelo treinador controlado pelo computador. Como podemos ver comparando os diagramas da figura 11 e 12 temos duas classes em quase tudo semelhantes. A única diferença aqui é que nos métodos *chooseActivePokémon* e *chooseMove*, não recebemos qualquer argumento e a escolha é feita através de uma variável *Math.Random*.

```
public void chooseActivePokemon() {
    int pokemonIndex = (int) (Math.random() * (3));
    if(pokemonTeam.get(pokemonIndex).isFainted())
        chooseActivePokemon();
    else
        activePokemon = pokemonTeam.get(pokemonIndex);
}
```

Figura 14– Método chooseActivePokémon da classe AITrainer

Neste método é interessante notar duas coisas. A primeira é que precisamos de verificar se o Pokémon que escolhemos aleatoriamente tem mais de 0 pontos de vida para poder combater e a segunda directamente associada a este despiste, se efectivamente o Pokémon escolhido não tiver as condições necessárias para ser colocado como activo, iremos percorrer este método de forma recursiva.

8. PokemonPool

Figura 15– Diagrama da classe PokemonPool

Antes de podermos começar a desenvolver o combate da aplicação temos de ter uma lista de Pokémon pode onde o utlizador possa escolher a sua equipa. Para criarmos um Pokémon necessitamos de criar *PokemonMove's* e sabemos que um *PokemonMove* pode ser usado por diversos Pokémon. Foi então tomada a decisão de juntar a criação de todos estes objectos numa única classe auxiliar. A classe simplesmente tem uma variável e um método, a variável que contém a lista de Pokémon e o método que retorna esta lista.

No método declaramos então objectos da classe Pokémon, objectos da classe *PokémonMove*, adicionamos os ataques aos Pokémon e adicionamos estes à lista.

```
Pokemon espeon = new Pokemon("Espeon", 240, PokemonType.PSYCHIC, 238, 175, 202);
PokemonMove psychic = new DamageMove("Psychic" ,120, PokemonType.PSYCHIC, StatusType.DEFAULT, 0, 10);
PokemonMove dazzlingGleam = new DamageMove("Dazzling Gleam", 90, PokemonType.FAIRY, StatusType.DEFAULT,0, 15);
PokemonMove shadowBall = new DamageMove("Shadow Ball", 90, PokemonType.GHOST, StatusType.DEFAULT,0, 15);
PokemonMove calmMind = new MultiplierMove("Calm Mind", "attack", true, 20);
espeon.addMoves(psychic, dazzlingGleam, shadowBall, calmMind);
```

Figura 16– Exemplo da criação e adição de um Pokémon à PokémonPool

Na figura 15 podemos observar então os passos acima descritos. Todos os valores que podemos verificar aqui como argumentos são valores que é necessário pesquisar em bases de dados disponíveis online com todos os Pokémon. Este facto leva a que seja muito trabalhoso e extenso a adição de muitos Pokémon à aplicação sendo por isso o motivo pelo qual no início deste relatório é indicado que será utilizada uma lista restrita de Pokémon.

9. Battle

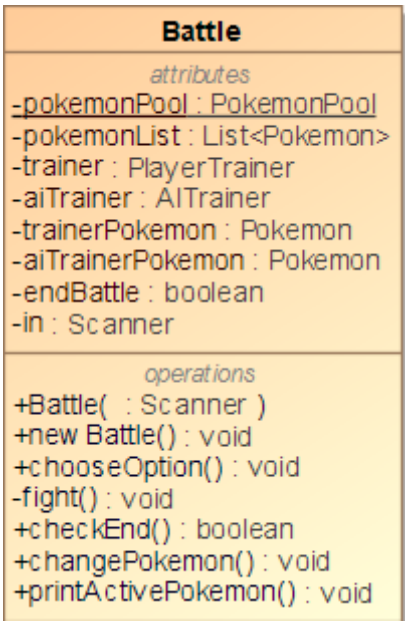

Figura 17– Diagrama da classe Battle

A classe *Battle* é responsável por gerir e controlar todo o fluxo de uma batalha. Como tal para termos uma batalha, necessitamos de ter dois treinadores - um controlado pelo utilizador (*PlayerTrainer*) e outro controlado pelo computador (*AITrainer*) – e cada um destes treinadores terá um Pokémon activo. Para escolhermos os Pokémon temos um objecto da classe *PokemonPool* que nos irá dar a lista de Pokémon disponíveis. Precisamos também de uma variável para verificar se a batalha terminou ou não, que será *endBattle* do tipo booleano.

No constructor desta classe recebemos como único argumento um Scanner, que será usado no modo de consola para introduzirmos que Pokémon é que queremos da lista, que opções queremos tomar durante a batalha, que ataques queremos utilizar e ainda que Pokémon pretendemos assumir como activo.

O método newBattle inicia um novo combate, iniciando então novos objectos para cada treinador e indicando na consola que o utilizador tem de escolher 3 Pokémon da lista disponibilizada. Este método só fica concluído quando o utilizador fizer as suas 3 escolhas. Em paralelo o computador também escolhe aleatoriamente os Pokémon para a sua equipa. Após concluído este processo este método define ainda os Pokémon activos de cada treinador.

O nosso menu de batalha será gerado pelo método *chooseOption*. Este método imprime na consola as opções disponíveis para o utilizador e fica a aguardar um input do mesmo. Após recebido este input. Este input é depois percorrido num *switch-case* e se alguma das opções corresponder ao input é executado o código correspondente. Neste caso no menu teremos as opções *Fight*(executar um turno de combate), *Change* (trocar de Pokémon) e *New Game*(começar uma nova batalha com novas equipas).

O método *fight* é responsável por tomar as decisões directas de batalha. Começa por imprimir na consola os dois Pokémon activos e mostra ao utilizador a lista de ataques do seu Pokémon e fica a aguardar o input de uma opção desta lista. Recebendo a opção, vai verificar qual é o move do Pokémon correspondente a esta opção e escolher um move aleatoriamente do treinador do computador. De seguida verifica se ambos os Pokémon têm algum tipo de efeito associado a um status no início do turno. São depois executados vários *If* para determinar qual o primeiro Pokémon a actuar (determinado pelas velocidades), verificar se o move executado derrotou o Pokémon alvo, se sim, dependendo de a quem pertence este, pedir ao utilizador para escolher um novo Pokémon ou ao computador para escolher aleatoriamente um Pokémon. Depois de cada move é também verificado se algum dos treinadores não tem nenhum Pokémon que possa lutar. Se o turno chegar ao fim e a batalha não tiver terminado, é verificado se algum dos Pokémon tem um efeito associado ao seu status no final do turno.

Para se trocar de Pokémon, utilizamos o método *changePokemon*, que imprime a lista de Pokémon disponíveis ao utilizador (ou seja todos os Pokémon que não sejam o activo ou que tenham mais de 0 hp) e aguarda que seja introduzida a opção correspondente ao Pokémon que pretendemos mudar.

O método *printActivePokemon* simplesmente imprime na consola os dois Pokémon activos indicando qual o do utilizador e qual o do adversário.

10. Conclusões

Com a realização deste trabalho conseguimos cumprir os principais objectivos propostos: a realização de uma aplicação e a consolidação dos conhecimentos adquiridos ao longo do semestre. Embora tenha sido realizada uma versão adaptada e simplificada do que verdadeiramente é uma batalha num jogo original Pokémon, o que foi desenvolvido permitiu-nos testar todos os nossos conhecimentos e adquirir ainda mais.

Desde a implementação de ciclos, recursividade à implementação de interfaces e enumerados percorrermos toda a matéria abordada na unidade curricular e nos trabalhos práticos desenvolvidos. Com a própria pesquisa adquirimos ainda novos conhecimentos com a utilização de HashMap's.

Um dos aspectos que gostaríamos de melhorar seria a construção de objectos Pokémon, sendo que uma alternativa possível pode ser a construção de um ficheiro XML com todos os Pokémon, os seus dados, moves e que estes sejam lidos e construídos através de uma classe. No entanto isto continuaria a apresentar um dos obstáculos enfrentados que seria a pesquisa e obtenção de todas as informações sobre cada Pokémon e move.

No fim ficámos satisfeitos com o resultado final obtido, sendo que a aplicação faz exactamente o que nos tínhamos proposto a fazer, batalhas completas de Pokémon, com vários tipos de moves, aplicação de status e os seus efeitos e a possibilidade de trocar de Pokémon.