Universidad de Cádiz

Escuela Superior de Ingeniería

Diseño de Sistemas Software

Ejercicio evaluable

POOkemon

Nombre
Jesús Serrano Gallán

Fecha
19 de mayo de 2021

Profesor
Juan Manuel Dodero

Índice

1. Introducción

El año pasado, mientras repasaba para el examen final de la asignatura de Programación Orientada a Objetos, se me ocurrió crear un pequeño programa que aplicaba algunos conceptos vistos en la asignatura como la herencia o el `typeid` vistos durante la asignatura. El objetivo del programa es conseguir generar distintos tipos de Pokemons y simular una batalla entre ellos. En esta pelea lo que queremos conseguir es que se pongan en juego sus debilidades y fortalezas. Sin embargo, con los recursos que tenía en ese momento, el programa presentaba multitud de dependencias innecesarias, que hacían evidente la necesidad de aplicar un rediseño.
Este año, he decidido reimplementarlo utilizando un patrón de diseño, para eliminar esas dependencias. Tras consultarlo con el profesor, vimos que el patrón más apropiado que podría aplicar es el patrón Mediator que veremos a continuación.

2. Patrón Mediator

El patrón Mediator es un patrón de comportamiento que precisamente nos permite reducir las dependencias entre los objetos. Básicamente lo que hace es meter un nivel de indirección en las comunicaciones que tienen los objetos entre sí, de tal forma que no dependan directamente los unos de los otros, si no que es el objeto Mediator el que resuelve estas interacciones entre objetos, sin que los participantes tengan que enterarse de nada.
Veámoslo con nuestro ejemplo. Imaginemos que queremos modelar las batallas entre tres tipos de Pokemon: los de tipo Agua, los de tipo Planta y los de tipo Fuego. Para conseguir esto, necesitaremos modelar en cada tipo de Pokemon cuán eficaz es el ataque hacia otros tipos de Pokemon. Es decir, si yo tengo un Pokemon de tipo planta tendré que modelar una operación `PokemonTipoPlanta::atacar(Pokemon& destino)` que se comporte de distinta forma para cada tipo de pokemon destino. Es decir, un comportamiento para cuando ataque Pokemons de tipo Planta, otro para cuando ataque Pokemons de tipo Agua y otro para cuando ataque Pokemons de tipo Fuego. Y esto, claro está, en cada una de las clases de TipoPokemon. Es decir, todas clases de tipos de Pokemon dependen de todas las clases de tipos de Pokemon, lo cual es una barbaridad. En el momento en el que me apetezca añadir un nuevo tipo de Pokemon, voy a tener que o crear un nuevo método en todas las clases para resolver el ataque al nuevo tipo Pokemon o que tocar los métodos de todas las clases de tipos Pokemon de mi programa para incorporarles el nuevo comportamiento. Y eso, como es de intuir, no es nada bueno.
Con el patrón Mediator, lo que hacemos es delegar esa responsabilidad de resolver las comunicaciones entre objetos a otro objeto distinto, el Mediador. De esta forma, cuando un Pokemon ataque a otro, lo que hará es llamar a un método de nuestro Mediador que resolverá esta interacción entre los objetos. Ahora el Pokemon atacante no sabe si su ataque será eficaz o no, ni tiene que saberlo, puesto que nuestro objeto Mediador es el que resolverá esta comunicación y le otorgorá el comportamiento deseado. Ahora los Pokemon no interaccionan directamente entre sí, si no que es el Mediador el que maneja sus interacciones. Si ahora quisieramos añadir un nuevo tipo de Pokemon, tan solo tendríamos que tocar nuestra clase Mediator, ya que es la que implementa la lógica que hay detrás de los ataques de nuestros Pokemon. De esta forma, hemos conseguido eliminar una ingente cantidad de dependencias y será mucho más fácil mantener y 'evolucionar' nuestro código.

3. POOkemon

Principalmente nuestro programa se basa en dos clases:

- **La clase abstracta Pokemon**: modela a los pokemons que combatirán. Esta clase abstracta encuentra su definición en tres clases, una para cada tipo de pokemon, que básicamente lo que implementan es una pequeña celebración que será distinta para cada tipo de Pokemon, la cual se usará cuando se gane en el combate.
- **La interfaz AtaquesMediator**: nuestro Mediador, será la intrerfaz que representará la lógica de los ataques. Es decir, la clase AtaquesImp, que implementa esta interfaz es la que sabrá si el ataque de un Pokemon tipo agua a un Pokemon de tipo fuego es supereficaz, eficaz o poco eficaz.

También se puede notar que los tipos de Pokemon están modelados en Tipo, que es simplemente un enumerado de los distintos tipos de Pokemon que tendrá nuestra aplicación (Agua, Planta y Fuego). En la siguiente página se puede ver un diagrama de clases para su mejor comprensión. A continuación veremos el código de la aplicación.

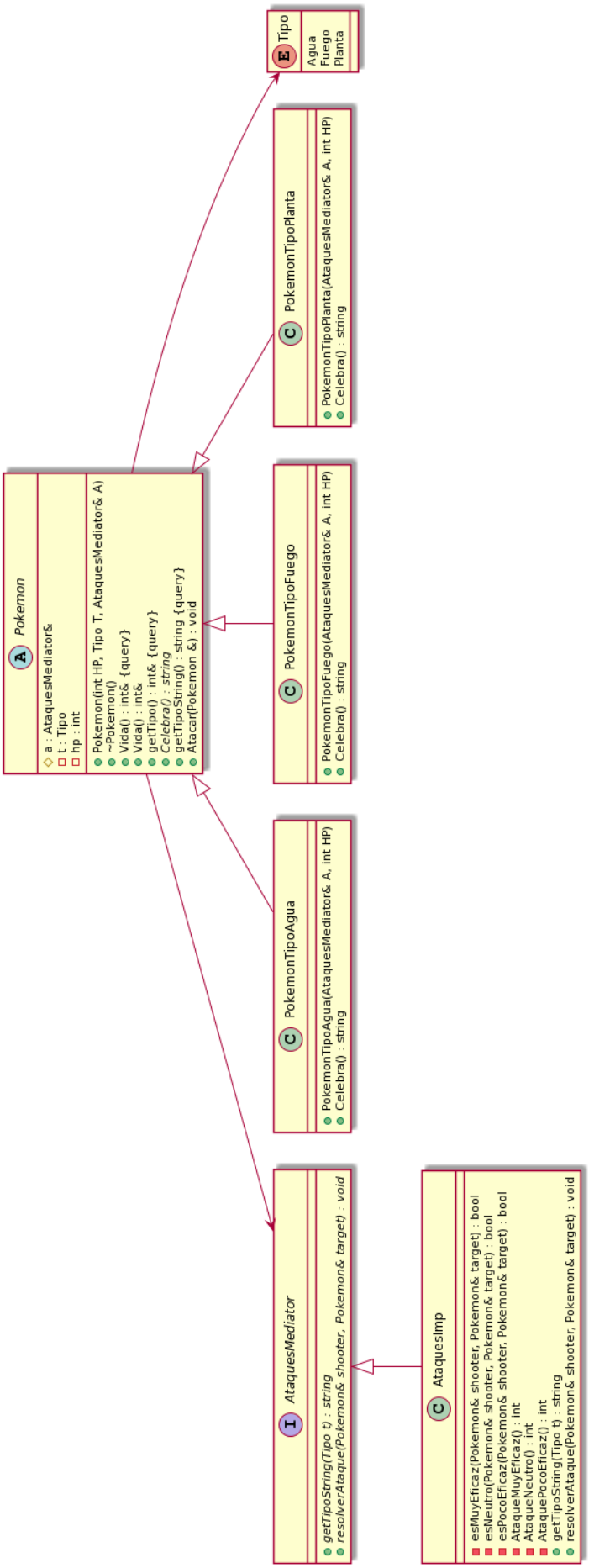

Figura 1: Diagrama de clases del programa

3.1. POOkemon.hpp

```
#ifndef POKEMON_HPP
#define POKEMON_HPP

#include <string>

using namespace std;

//Declaración adelantada
class Pokemon;

//Enumerado de tipos
enum Tipo
{
    Planta, Agua, Fuego
};

//Interfaz Mediator
class AtaquesMediator
{
    //Se trata de una interfaz porque todos sus métodos son virtuales puros
    public:
        virtual void resolverAtaque(Pokemon& shooter, Pokemon& target) = 0;
        virtual string getTipoString(Tipo t) = 0;
};

//Implementación del mediator
class AtaquesImp: public AtaquesMediator
{
    public:
        void resolverAtaque(Pokemon& shooter, Pokemon& target) override final;
        string getTipoString(Tipo t) override final;
    private:
        bool esPocoEficaz(Pokemon& shooter, Pokemon& target);
        bool esNeutro(Pokemon& shooter, Pokemon& target);
        bool esMuyEficaz(Pokemon& shooter, Pokemon& target);
        int AtaquePocoEficaz();
        int AtaqueNeutro();
        int AtaqueMuyEficaz();
};

//Clase abstracta
class Pokemon
{
    public:
        //Constructor
        Pokemon(int HP, Tipo T, AtaquesMediator& A): hp{HP}, t{T}, a{A} {}

        //Devuelve el enumerado del tipo del pokemon
        const Tipo& getTipo() const {return t;}
```

```
        //Devuelve un string segun el tipo del pokemon
        string getTipoString() const;

        //Método de ataque
        void Atacar(Pokemon&);
        //Celebración de la victoria (virtual puro)
        //Se trata de una clase abstracta porque al menos uno de sus métodos es virtu
        virtual string Celebra() = 0;

        //Consultor de la vida
        const int& Vida() const {return (hp);}
        //Modificador de la vida
        int& Vida() {return(hp);}

        //Destructor
        virtual ~Pokemon() {};

    private:
        int hp; //Atributo con la vida
        Tipo t; //Atributo con el tipo del pokemon

    //Protegido
    protected:
        AtaquesMediator& a; //Mediador para los ataques
};


//Clase de pokemon tipo planta
class PokemonTipoPlanta: public Pokemon
{
    public:
        //Constructor, usa el de pokemon genérico
        PokemonTipoPlanta(AtaquesMediator& A, int HP = 10): Pokemon(HP, Planta, A){}

        //Celebración de la victoria del pokemon tipo planta
        string Celebra() override final;
};

class PokemonTipoFuego: public Pokemon
{
    public:
        //Constructor, usa el de pokemon genérico
        PokemonTipoFuego(AtaquesMediator& A, int HP = 10): Pokemon(HP, Fuego, A){}

        //Celebración de la victoria del pokemon tipo fuego
        string Celebra() override final;
};

class PokemonTipoAgua : public Pokemon
{
```

```
    public:
        //Constructor, usa el de pokemon genérico
        PokemonTipoAgua(AtaquesMediator& A, int HP = 10): Pokemon(HP, Agua, A){}

        //Celebración de la victoria del pokemon tipo agua
        string Celebra() override final;
};

#endif
```

3.2. POOkemon.cpp

```
#include <iostream>
#include "POOkemon.hpp"

using namespace std;

//Mediator
//Con este método resolvemos todos los ataques entre pokemons
void AtaquesImp::resolverAtaque(Pokemon& shooter, Pokemon& target)
{
    if(esNeutro(shooter, target))
    {
        target.Vida() += AtaqueNeutro();
    }
    else
    {
        if(esMuyEficaz(shooter, target))
        {
            target.Vida() += AtaqueMuyEficaz();
        }
        else
        {
            if(esPocoEficaz(shooter, target))
            {
                target.Vida() += AtaquePocoEficaz();
            }
            else
            {
                cout << "No se puede reconocer la situación" << endl;
            }
        }
    }
}

//Estos métodos nos marcan la eficacia del ataque realizado por el shooter hacia el t
bool AtaquesImp::esNeutro(Pokemon& shooter, Pokemon& target)
{
    return(shooter.getTipo() == target.getTipo());
}
```

```
bool AtaquesImp::esPocoEficaz(Pokemon& shooter, Pokemon& target)
{
    bool r;

    r = ((shooter.getTipo() == Planta)&&(target.getTipo() == Fuego)) ||
        ((shooter.getTipo() == Fuego)&&(target.getTipo() == Agua)) ||
        ((shooter.getTipo() == Agua)&&(target.getTipo() == Planta));

    return r;
}

bool AtaquesImp::esMuyEficaz(Pokemon& shooter, Pokemon& target)
{
    bool r;

    r = ((shooter.getTipo() == Planta)&&(target.getTipo() == Agua)) ||
        ((shooter.getTipo() == Fuego)&&(target.getTipo() == Planta)) ||
        ((shooter.getTipo() == Agua)&&(target.getTipo() == Fuego));

    return r;
}

//Estos métodos codifican el comportamiento de cada ataque según su eficacia (mensaje
int AtaquesImp::AtaquePocoEficaz()
{
    cout << "No es muy eficaz..." << endl;
    return(-1);
}

int AtaquesImp::AtaqueNeutro()
{
    cout << "Se ha realizado el ataque" << endl;
    return(-2);
}

int AtaquesImp::AtaqueMuyEficaz()
{
    cout << "¡Es supereficaz!" << endl;
    return(-4);
}

//Este método solo sirve para convertir un objeto Tipo en un String
string AtaquesImp::getTipoString(Tipo t)
{
    string s;
    switch (t)
    {
    case Planta:
        s = "tipo planta";
```

```
        break;
    case Fuego:
        s = "tipo fuego";
        break;
    case Agua:
        s = "tipo agua";
        break;
    default:
        s = "tipo no reconocido";
        break;
    }
    return s;
}

//Pokemon
void Pokemon::Atacar(Pokemon& target)
{
    //El comportamiento de los ataques se delega al mediador
    a.resolverAtaque(*this, target);
}

string Pokemon::getTipoString() const
{
    return(a.getTipoString(getTipo()));
}

//Celebraciones: cada tipo pokemon tendrá una distinta
string PokemonTipoPlanta::Celebra()
{
    return(" luce sus hojas al sol!");
}

string PokemonTipoFuego::Celebra()
{
    return(" lanza gigantescas llamas de alegria!");
}

string PokemonTipoAgua::Celebra()
{
    return(" forma un remolino gigantesco!");
}

//Resto de funciones
//Esta función simula el ciclo de un combate. Los pokemons atacan mientras que los do
//Después, el ganador celebra la victoria
void Combate(Pokemon& P1, Pokemon& P2)
{
    while (P1.Vida()>0 && P2.Vida()>0)
    {
        cout << endl << "¡P1 ha atacado!"<< endl;
```

```
            P1.Atacar(P2);
            if (P2.Vida()<=0)
                break;
            cout << endl << "¡P2 ha atacado!"<< endl;
            P2.Atacar(P1);
        }
        if (P1.Vida()<0)
        {
            cout << endl<< "¡P2 ha ganado el combate!" << endl;
            cout << "¡P2" << P2.Celebra() << endl;
        }
        else
        {
            cout << endl<< "¡P1 ha ganado el combate!" << endl;
            cout << "¡P1" << P1.Celebra() << endl;
        }
}

//Los métodos a continuación sirven para devolver Pokemons de tipos aleatorios para q
//Uno devuelve una referencia y otro un puntero.
Pokemon& Pokemon_Aleatorio_Referencia(AtaquesMediator& A, int HP = 10)
{
    switch (rand()%3)
    {
        case 0:
            return *new PokemonTipoPlanta(A, HP); //UpCasting, se devuelve un objeto
            break;
        case 1:
            return *new PokemonTipoFuego(A, HP);
            break;
        case 2:
            return *new PokemonTipoAgua(A, HP);
            break;
        default:
            return *new PokemonTipoPlanta(A, HP);
            break;
    }
}

Pokemon * Pokemon_Aleatorio_Puntero(AtaquesMediator& A, int HP = 10)
{
    switch (rand()%3)
    {
        case 0:
            return new PokemonTipoPlanta(A, HP); //UpCasting
            break;
        case 1:
            return new PokemonTipoFuego(A, HP);
            break;
        case 2:
```

```
            return new PokemonTipoAgua(A, HP);
            break;
        default:
            return new PokemonTipoPlanta(A, HP);
            break;
    }
}
```

3.3. combate.cpp

```
#include <ctime>
#include <iostream>
#include "POOkemon.hpp"

void Combate(Pokemon& P1, Pokemon& P2);
Pokemon& Pokemon_Aleatorio_Referencia(AtaquesMediator& A, int HP = 10);
Pokemon * Pokemon_Aleatorio_Puntero(AtaquesMediator& A, int HP = 10);

int main()
{
    //Creamos nuestro mediador
    AtaquesMediator * mediator = new AtaquesImp();

    srand(time(0));
    //Generamos nuestros pokemons que usarán nuestro mediador
    Pokemon * P1 = Pokemon_Aleatorio_Puntero(*mediator, 10);
    Pokemon& P2 = Pokemon_Aleatorio_Referencia(*mediator, 10);

    //Mostramos los tipos de los pokemon
    cout << "P1 es de " << P1->getTipoString() <<endl;
    cout << "P2 es de " << P2.getTipoString() << endl;

    //Simulamos el combate
    Combate(*P1, P2);

    return(0);
```

3.4. Makefile

Se adjunta el Makefile por si se quisiera probar el programa.

```
CXX = g++
CXXFLAGS = -g -Wall -std=c++11 -pedantic
CPPFLAGS = -I.

EXES = combate
CLASES = combate.o POOkemon.o

.PHONY: all clean

all: $(EXES)
```

```
combate: $(CLASES)
	$(CXX) $(CXXFLAGS) $^ -o $@

clean:
	@$(RM) $(EXES) *.o
```

4. Ejemplo de ejecución

A continuación se muestran varios ejemplos de ejecución para que se vea que el programa consigue el comportamiento deseado.

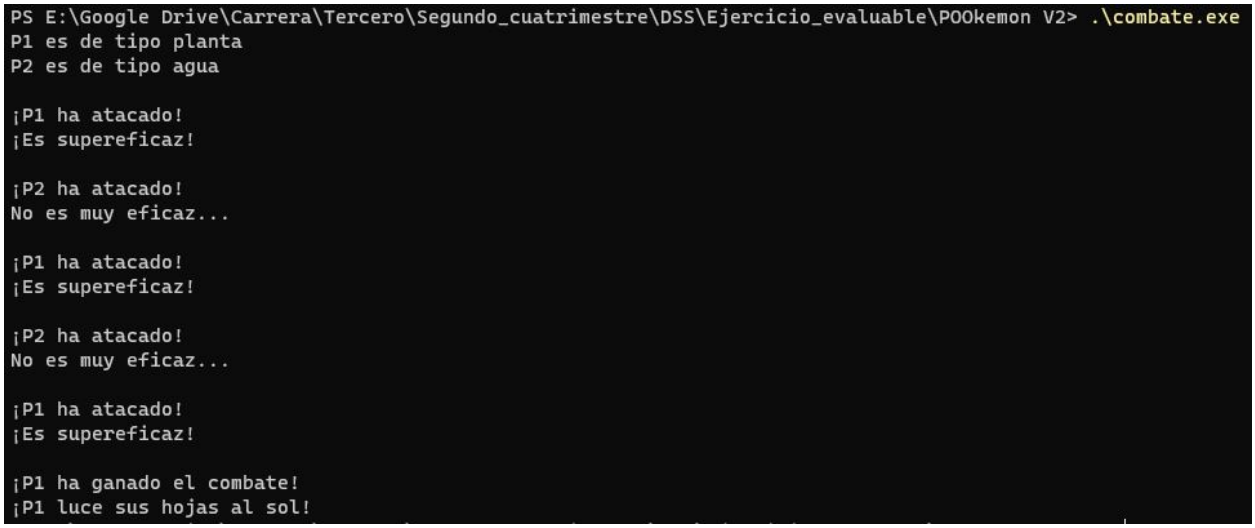

Figura 2: Victoria de un pokemon tipo Planta

Figura 3: Victoria de un pokemon tipo Agua

Figura 4: Victoria de un pokemon tipo Fuego