DESARROLLO DE PROGRAMAS

Curso 2017/18

Grado en Ingeniería Informática en
Ingeniería del Software
Ingeniería de Computadores

DOCUMENTACIÓN EXTERNA DEL PROYECTO

INDICE

Blanco Mangut, Álvaro
Desarrollo de programas

MANUAL DEL PROGRAMADOR

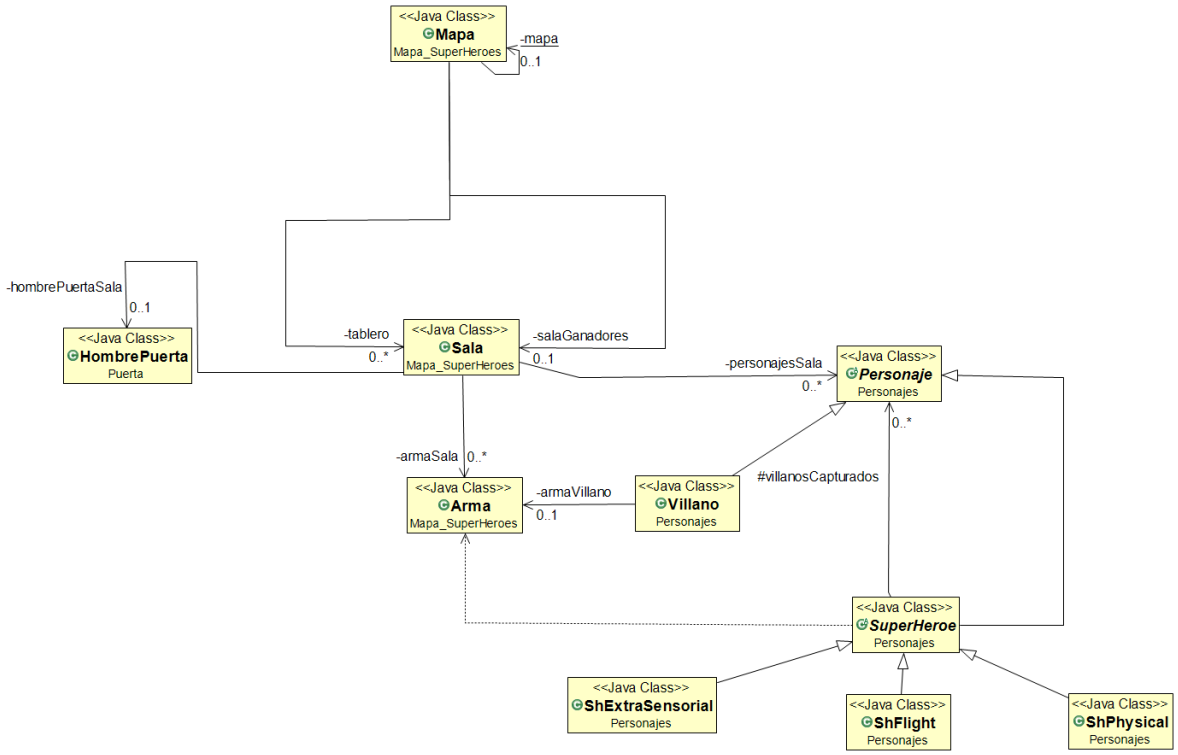
1.1. Introducción

El proyecto realizado se trata de un “videojuego”, entre comillas, que tiene por características, el estar compuesto por un tablero, en el que nos encontramos salas que albergan armas, y unos personajes, que podemos dividir en superhéroes, villanos y el hombre que guarda la puerta que se encuentra en la sala final del mapa.

No podemos realmente llamarlo video juego puesto que realmente no puedes interactuar con los personajes, el programa es totalmente automático y el problema que plantea es obtener un conjunto de armas, por parte de los personajes sin incluir al hombre puerta, capaces de vencer el conjunto de armas inicial del hombre puerta.

1.2. Análisis

1.2.1. Diagrama del modelo conceptual.



1.2.2. Identificación de colaboración y responsabilidad (CRC's)

Clase: **Mapa**	
Descripción: Clase que se encarga de crear el mapa en el que se desarrollara el juego	
Responsabilidad	**Colaborador**
Generar el mapa	Sala
Introducir a los personajes en el mapa	Personajes
Configurar las armas	Arma
Configurar al hombre puerta	Hombre puerta
Crea las paredes	
Crea los atajos	
Pinta o muestra la situación de un turno	

Clase: **Sala**	
Descripción: Clase que identifica a las salas que formaran parte del tablero de juego	
Responsabilidad	**Colaborador**
Gestión de las armas	Arma
Gestión de los personajes	Personajes
Gestión del hombre puerta	Hombre puerta
Pinta la sala del tablero en función de las paredes.	
Mostrar su información	

Clase: **Hombre puerta**	
Descripción: Clase que crea al hombre puerta	
Responsabilidad	**Colaborador**
Gestionar las armas con las que inicia la simulación	
Mostrar su información	

Clase: **Personaje**	
Descripción: Clase abstracta que define atributos comunes a los distintos tipos de personajes	
Responsabilidad	**Colaborador**
Realizar los métodos comunes de todos los personajes.	
Albergas los atributos comunes de los personajes	
Definir los métodos que se implementaran en las clases inferiores	Villano y Superhéroe

Clase: **Arma**	
Descripción: Define a las armas	
Responsabilidad	**Colaborador**
Atribuirles nombre y poder a las instancias creadas.	
Crear métodos que permitan ser comparadas tanto por poder como por nombre	

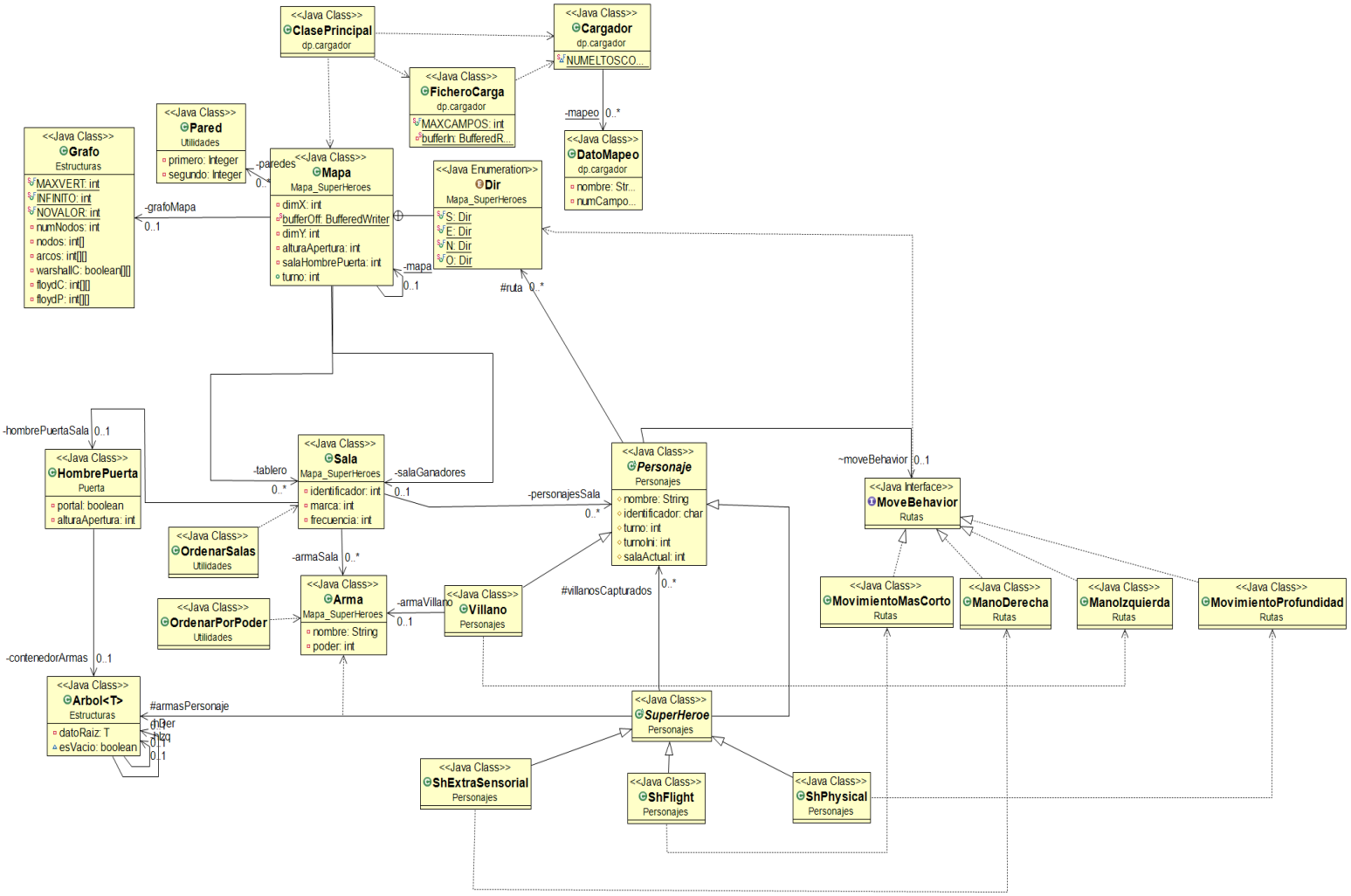
Clase: **Villano**	
Descripción: Tipo de personaje que hereda del padre	
Responsabilidad	**Colaborador**
Realizar acciones determinadas, distintas a los otros tipos de personaje	Personaje
Mostrar su información	
Realizar acciones con la única arma que posee	Arma

Clase: **Superhéroe**	
Descripción: Tipo de personaje que herida del padre	
Responsabilidad	**Colaborador**
Implementar acciones que son distintas para los tipos de personaje	Personaje
Definir las acciones que serán implantadas por los hijos	
Crear la estructura que almacenara las armas de los personajes	

Clase: **Extrasensorial, Physical y Fly**	
Descripción: Clase que heredan de superhéroe	
Responsabilidad	**Colaborador**
Implementar los métodos definidos en el padre	
Implementar las acciones que los diferencian	
Mostrar su información	

1.3. Diseño

1.3.1. Diagrama de clases del sistema



1.3.2. Contrato de operaciones a nivel de diseño

Precondiciones y Postcondiciones. Documentación generada con JavaDoc, Doxygen, etc. (no debe incluirse en la documentación externa)

1.3.3. Estructuras de Datos utilizadas

1.3.3.1. Estructuras de datos estáticas

La única estructura que encontramos de este tipo se encuentra en la clase mapa y hace referencia al atributo Tablero que es un tipo Matriz de Salas:

```
/**
 * Matriz que contiene las salas
 */
private Sala[][] tablero;
```

Este tablero alberga una matriz formada por salas que se generan automáticamente cuando se crea la instancia de Mapa. El número de salas

creada durante la ejecución del programa viene marcado por los atributos DimX y DimY que hacen referencia al número de filas y columnas.

Debido a que el tablero es un elemento fijo, el cual no va a albergar más salas durante la ejecución y además estas no van a ser eliminadas, es lógico pensar que ha de ser un elemento estático.

1.3.3.2. Estructuras de datos dinámicas

Encontramos diversas estructuras de este tipo:

1. Árbol

En el proyecto encontramos una clase que implementa esta estructura:

```
public class Arbol<T extends Comparable<T>>
```

Se trata de una clase de tipo publica, puesto que diversos objetos implementan esta estructura, ademas en la cabecera de la misma podemos apreciar que se trata de una estructura de tipo T para que acepte cualquier tipo de dato en su interior. Añadido a todo esto indicamos que hace uso de los métodos *compareTo* de la clase que está haciendo uso de la misma.

Encontramos esta estructura presente en los personajes de tipo Superhéroe y en el hombre puerta. Ambos lo usan para guardar las armas generadas.
Los Superhéroes recogen las armas de las salas por las que pasan, modificando su poder si tiene una del mismo nombre o dejándola si el enfrentamiento contra un villano no resulta satisfactorio.
El hombre puerta por su parte, almacena una serie de armas ya pre-configuradas que las ira perdiendo conforme los personajes se enfrente a el

2. Grafo

Al igual que el árbol, encontramos una clase que implementa la estructura:

```
public class Grafo
```

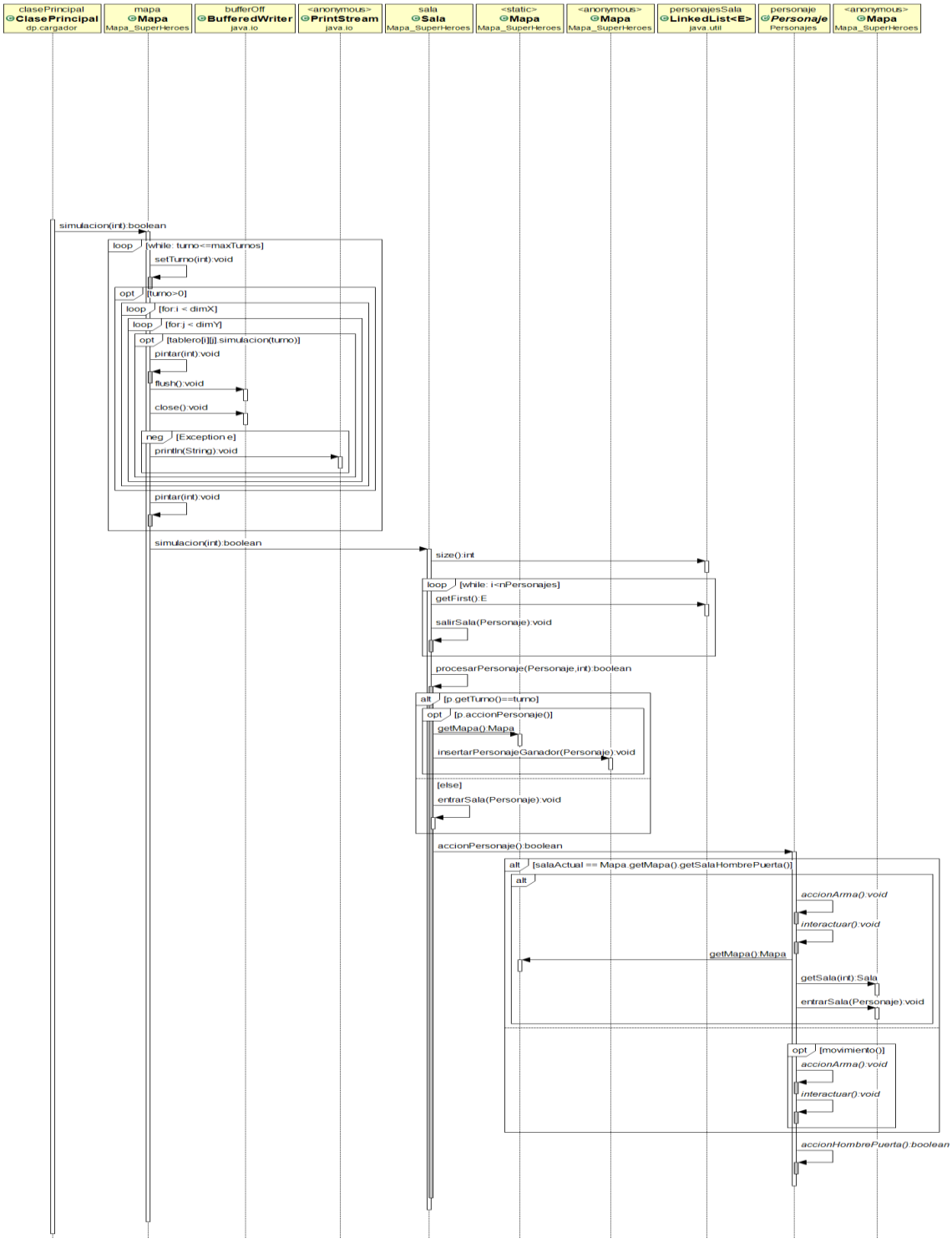
A diferencia del árbol esta estructura solo puede almacenar enteros, en concreto en el grafo se almacenará los identificadores de las salas que se han creado, así como sus conexiones o paredes. En general el grafo es una estructura que nos permite gestionar el tablero del tal manera que en el almacenamos las conexiones, así como la posibilidad de obtener los mejores camino de una sala a otra
Comentar que las dos clases anterior han sido implementadas inicialmente por los profesores de la asignatura y que los alumnos han ido implementando conforme el desarrollo del proyecto.

3. Lista, pila y cola

En las primeras fases de desarrollo fue necesario la inclusión de una lista, una pila y una cola para manejar ciertos datos correspondientes al programa.
Con el avance del proceso de desarrollo, se decidió hacer uso de las estructuras de datos que tiene propiamente java debido a su gran cantidad de métodos y la baja complejidad de los mismos.

Por lo tanto, ahora solo hacemos usos de la *LinkedList*, o lista doblemente enlazada, puesto que funciona como una lista una pila o una cola. Esta estructura esta presenta en diversas clases del proyecto, aunque a destacar podemos decir que las salas tienen esta estructura para almacenar de forma ordenada las armas.

1.3.4. Diagramas de secuencia



*Librería utilizada para la creación de los diagramas: ObjectAid.

1.4. Implementación

1.4.1. Algoritmos de especial interés

Se describen con mayor detalle los algoritmos más relevantes, los más interesantes por su implementación o por su forma de abordar un determinado problema.
Recomendable: incluir pseudocódigo e ilustrarlo con algún ejemplo

```
private int ObtenerFrecuencias(Integer origen, int destino, LinkedList<Integer>visitados,LinkedList<Sala>recorrido)
```

ObtenerFrecuencia es un método que encontramos en la clase mapa y es privado puesto que solo es necesario para obtener las salas con mayor frecuencia de paso, según las condiciones que los profesores nos proporcionaron en la tercera entrega.

```
Integer vertice=0;//Nodo vertice al nodo en el que estamos
        int bandera = 0;//Bandera booleana
        Integer caminosHastaAhora=0;//Entero que nos indica el numero de caminos validos hasta ahora

        TreeSet<Integer> ady= new TreeSet<Integer>();//creamos un treeSet para guardar los adyacentes del nodo con el que estamos trabajando
        Sala aux=getSala(origen);

        si(origen!=destino){//Si el origen es igual al destino el camino que se ha llevado a cabo es valido, si no lo es seguimos explorando caminos

            visitados.add(origen);//Añadimos el nodo con el que estamos trabajando al visitados

            adyacentes(origen, ady);//Cogemos los adyacentes del nodo que estamos tratando

            mientras(!ady.isEmpty()){//Mientras haya nodos por explorar y no hayamos llegado al final

                vertice=first();//Igualo el nodo axuliar al primer adyancente

                remove(vertice);//Elimino el adyacente que vamos a tratar

                si(!visitados.contains(vertice)){
                    bandera=obtenerFrecuencia(vertice, destino,visitados,recorrido);
                    si(bandera>0){//Quiere decir que el camino es correcto

                        aux.setFrecuencia(aux.getFrecuencia()+bandera);
                        si(recorrido.contains(aux)){
                            Collections.sort(recorrido, Collections.reverseOrder(new OrdenarSalas()));
                        }sino{
                            recorrido.add(aux);
                            Collections.sort(recorrido, Collections.reverseOrder(new OrdenarSalas()));
                        }
                        caminosHastaAhora=caminosHastaAhora+bandera;
                    }
                }
            }
            remove(origen);
            return caminosHastaAhora;
        }
        aux.frecuenciaMas();
        si(recorrido.contains(aux)){
            Collections.sort(recorrido, Collections.reverseOrder(new OrdenarSalas()));
        }sino{
            recorrido.add(aux);
            Collections.sort(recorrido, Collections.reverseOrder(new OrdenarSalas()));
        }
        bandera++;

        return bandera;
    }
```

```
private void kruskal()
```

Kruskal, al igual que ObtenerFrecuencia(), se encuentra en la clase mapa. A rasgos generales, este algoritmo se encarga de derribar las paredes en función de números aleatorios que genera la clase GenAleatorios.

```
int random;//Entero para guardas los numeros random generados
Pared aux;//Auxilar para guardar las paredes
Sala s1;
Sala s2;

mientras(!paredes.isEmpty()){

    random=GenAleatorios.generarNumero(paredes.size());
    aux=paredes.get(random);

    s1=tablero[identificarFila(aux.getPrimero(), dimY)][identificarColumna(aux.getPrimero(), dimY)];//Identificamos a la primera sala a la cual pertenece la pared
    s2=tablero[identificarFila(aux.getSegundo(), dimY)][identificarColumna(aux.getSegundo(), dimY)];//Identificamos a la segunda sala a la cual pertenece la pared

    si (s1.getMarca()!=s2.getMarca()){

        si(grafoMapa.nuevoArco(aux.getPrimero(),aux.getSegundo(), 1)
                &&
           grafoMapa.nuevoArco(aux.getSegundo(),aux.getPrimero(), 1)){
        }

        extenderMarcaSala(s1.getMarca(),s2.getMarca());
    }

    remove(random);
}

grafoMapa.warshall(); // actualizamos la matriz de warshall
grafoMapa.floyd(); // actualizamos la matriz de floyd
}
```

```
private void atajos()
```

Como los anteriores, se encuentra también en la clase mapa. Este método generar N atajos (N = 5% del número total de salas), estos atajos no son ni más ni menos que la eliminación de paredes. Como en el anterior los atajos se generan aleatoriamente.

```
public class ManoDerecha implements MoveBehavior
```

ManoDerecha es un algoritmo que implementa un tipo de personaje. Es un método que genera una ruta siguiendo la regla de la mano derecha (Lleva la mano derecha pegada a las paredes del mapa). Este algoritmo tiene además una variante, ManoIzquierda.

Debido a lo largo que es el algoritmo, voy a prescindir del seudocódigo completo del método y voy a intentar explicarlo con algunos ejemplos.
En esencia el algoritmo de la mano derecha se basa en saber de dónde vienes y pensar que siempre debes girar hacia la derecha. Ejemplo:

```
si(e){//Si vengo del este, signifca que tengo o tenia pared en el sur y que la mano derecha esta o estaba pegeda a ella
    e=false;

    si(Mapa.getMapa().comprobarMovimiento(origen, origen+Mapa.getMapa().getColumnas())){//Compruebo que pueda girar a la derecha

        origen=origen+Mapa.getMapa().getColumnas();//Actualizo el origen
        s=true;
        move(origen,destino,o,s,e,n,ruta);
        ruta.addFirst(Dir.S);

    }si no{

        si(Mapa.getMapa().comprobarMovimiento(origen, origen+1)){//No he podido girar a la derecha
                                                                 //compruebo que pueda seguir avanzando

            origen=origen+1;//Actualizo el origen
            e=true;
            move(origen,destino,o,s,e,n,ruta);
            ruta.addFirst(Dir.E);

        }si no{

            si(Mapa.getMapa().comprobarMovimiento(origen, origen-Mapa.getMapa().getColumnas())){//Compruebo que pueda ir al norte

                origen=origen-Mapa.getMapa().getColumnas();//Actualizo el origen
                n=true;
                move(origen,destino,o,s,e,n,ruta);
                ruta.addFirst(Dir.N);

            }si no{//Si no podemos hacer nada de lo anterior significa que tenemos que dar la vuelta

                origen=origen-1;//Actualizo el origen
                o=true;
                move(origen,destino,o,s,e,n,ruta);
                ruta.addFirst(Dir.O);
            }
        }
    }
```

Como observamos en el algoritmo si venimos del este significa que tenemos la mano pegada a una pared en una pared que está en el sur. Los más lógico sería pues, intentan girar a la derecha y si no seguir de frente

Además, como ya se comentó antes guarda mucha similitud, por no decir que es el mismo código, con el algoritmo de la mano izquierda. Básicamente es igual que el anterior, pero en vez de preguntar todo el rato si podemos girar a la derecha, debemos preguntar si podemos girar a la izquierda.

```
public class MovimientoMasCorto implements MoveBehavior
```

Al igual que el anterior, nos encontramos ante un algoritmo que genera una ruta para un tipo de personaje. En este caso el método crea el camino más corto entre la sala origen y la sala destino.

```
int sig=Mapa.getMapa().siguienteNodo(origen, destino);

si (origen!=destino){//Si me devuelve -1 significa que hemos llegado al origen

    move(sig,destino,ruta);

    si ((origen-Mapa.getMapa().getColumnas())==sig){//Significa que me muevo al norte
        ruta.addFirst(Dir.N);
    }si no{
        si((origen-1)==sig){
            ruta.addFirst(Dir.O);
        }si no{
            si((origen+Mapa.getMapa().getColumnas())==sig){
                ruta.addFirst(Dir.S);
            }si no{
                si((origen+1)==sig){
                    ruta.addFirst(Dir.E);
                }
            }
        }
    }
}
return ruta;
```

```
public class MovimientoProfundidad implements MoveBehavior
```

Como en los anteriores casos este algoritmo generar una ruta utilizando el algoritmo en profundidad que encontramos en un grafo.

```
int vertice=0;
        TreeSet<Integer> ady= new TreeSet<Integer>();//creamos un treeSet para guardar los adyacentes del nodo con el que estamos trabajando

        si(origen!=destino){//Si el origen es igual al destino el camino que se ha llevado a cabo es valido, si no lo es seguimos explorando caminos

            visitados.add(origen);//Añadimos el nodo con el que estamos trabajando al visitados

            Mapa.getMapa().getAdyacentes(origen,ady);//Cogemos los adyacentes del nodo que estamos tratando


            mientras(!ady.isEmpty()&&!cierto[0]){//Mientras haya nodos por explorar y no hayamos llegado al final

                vertice=ady.first();//Igualo el nodo axuliar al primer adyancente pues to que siempre sera el mas pequeño

                ady.remove(vertice);//Elimino el adyacente que vamos a tratar

                si(!visitados.contains(vertice)){//Si el adyacente esta dentro de los nodos vistados, significa que el camino no es valido

                    ruta=move(vertice, destino, visitados, ruta, cierto);

                    si (cierto[0]){
                        si ((origen-Mapa.getMapa().getColumnas())==vertice){//Significa que me muevo al norte
                            ruta.addFirst(Dir.N);
                        }si no{
                            si((origen-1)==vertice){
                                ruta.addFirst(Dir.O);
                            }si no{
                                si((origen+Mapa.getMapa().getColumnas())==vertice){
                                    ruta.addFirst(Dir.S);
                                }si no{
                                    si((origen+1)==vertice){
                                        ruta.addFirst(Dir.E);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }si no{
            cierto[0]=true;
            return ruta;
        }
        return ruta;
```

```
private Arma getMayorArmaRec(Arbol<Arma> aux)
```

Este algoritmo es utilizado tanto por los Superheroes como por el hombre puerta, y les permite obtener el arma de mayor poder dentro de su Árbol de armas que esta ordenado por nombres.

```
Arma aAux = null;       //Arma auxiliar
Arma acumulada=null;//Mayor arma hasta ahora

si(aux!=null){//Mientras que el personaje tenga arma

        si(aux.esHoja()){//Vemos si el nodo es hoja
                return aux.getRaiz();//si lo es devuelvo el arma

        }si no{
                if(aux.getHijoIzq()!=null){
                        aAux=getMayorArmaRec(aux.getHijoIzq());

                        intentamos{     //Debido a que puede que comparemos con un objeto igual a null en algunas recursiviades
                                si(aAux.getPoder()>acumulada.getPoder())
                                        acumulada=aAux;

                                si(aAux.getPoder()==acumulada.getPoder())
                                        si(aAux.compareTo(acumulada)<0)
                                                acumulada=aAux;
                        }si falla(NullPointerException e){

                                si(aAux.getPoder()>aux.getRaiz().getPoder())
                                        acumulada=aAux;

                                si(aAux.getPoder()<aux.getRaiz().getPoder())
                                        acumulada=aux.getRaiz();

                                si(aAux.getPoder()==aux.getRaiz().getPoder())
                                        if(aAux.compareTo(aux.getRaiz())<0)
                                                acumulada=aAux;
                                        si no
                                                acumulada=aux.getRaiz();
                        }
                }
                si(aux.getHijoDer()!=null){
                        aAux=getMayorArmaRec(aux.getHijoDer());

                        intentamos{     //Debido a que puede que comparemos con un objeto igual a null en algunas recursiviades
                                if (aAux.getPoder()>acumulada.getPoder())
                                        acumulada=aAux;

                                si(aAux.getPoder()==acumulada.getPoder())
                                        si(aAux.compareTo(acumulada)<0)
                                                acumulada=aAux;
                        }si falla(NullPointerException e){

                                si(aAux.getPoder()>aux.getRaiz().getPoder())
                                        acumulada=aAux;

                                si(aAux.getPoder()<aux.getRaiz().getPoder())
                                        acumulada=aux.getRaiz();

                                si(aAux.getPoder()==aux.getRaiz().getPoder())
                                        if(aAux.compareTo(aux.getRaiz())<0)
                                                acumulada=aAux;
                                        si no
                                                acumulada=aux.getRaiz();
                        }
                }
        }
}
return acumulada;
```

1.4.2. Definición de entradas/salidas

Los datos de entrada del programa se obtienen mediante un txt, que especifica el número de columnas y fila de la matriz de salas y los personajes que se van a generar, especificados con el tipo que son y el turno en que van a comenzar a moverse.

Con respecto a la salida, lo que se generar es un txt, con la denominación FicheroSalida, en el encontramos todo lo referido a la situación del juego en cada turno. Ejemplo:

```
(turn:1)
(map:99)
(doorman:closed:2:(Acido,1)(Anillo,11)(Antorcha,5)(Armadura,13)(Baston,22)(Bola,3)(Cadena
Fuego,11)(CampoEnergia,5)(CampoMagnetico,5)(Capa,10)(Cetro,20)(Escudo,3)(Espada,11)(Flecha,12)(Garra,22)(Gema,4))
 _ _ _ _ _ _ _ _ _ _
|W D|_  | |   _ _ V|
|  _  |  _ _| | |_ A|
| |  _ _  |     | | |
|_|_   _|_  | | | | |
|_   _     _ _| | |_|
|_ _  |_|_| |_   _  |
| | |_    | |       |
| | |  _|_|  _|_| | |
|E   _    |_| |   | |
|_|_ _|_|_ _ _|_|_ _|
(square:0:(Mjolnir,29)(Garra,27)(Red,25)(Armadura,3)(Anillo,1))
(square:1:(GuanteInfinito,21)(LazoVerdad,9)(Lawgiver,7)(Escudo,5))
(square:11:(CadenaFuego,19)(Flecha,17)(Antorcha,15)(Tridente,13)(Capa,11))
(square:12:(Baston,28)(MazaOro,26)(Tentaculo,24)(CampoMagnetico,4)(Latigo,2))
(square:22:(Cetro,22)(Laser,20)(Bola,10)(RayoEnergia,8)(CampoEnergia,6))
(square:26:(Nullifier,23)(Espada,18)(Acido,16)(Gema,14)(Sable,12))
(square:27:(Anillo,29)(Armadura,27)(Red,5)(Garra,3)(Mjolnir,1))
(square:37:(Escudo,25)(Lawgiver,23)(LazoVerdad,21)(GuanteInfinito,9)(Lucille,7))
(square:47:(Antorcha,28)(Capa,19)(Tridente,17)(Flecha,13)(CadenaFuego,11))
(square:57:(Latigo,26)(CampoMagnetico,24)(Baston,15)(Tentaculo,4)(MazaOro,2))
(square:68:(Sable,16)(Acido,12)(Espada,10)(Nullifier,3)(Gema,1))
(square:99:(CampoEnergia,22)(RayoEnergia,20)(Bola,18)(Laser,8)(Cetro,6))
(shphysical:W:0:4:)
(shphysical:D:1:1:(Lucille,23))
(villain:V:9:2:)
(villain:A:19:1:)
(shflight:E:80:1:)
```

Podemos observar en la salida que se indica tanto el tipo de personaje como su marca identificadora, así como el turno en que mueve y además la sala en la que esta

Por pantalla mostramos los mismo que en el fichero de salida salvo que si se requieres pintar algo más, sería necesario hacer uso de los flujos de escritura para que quedara representado en el fichero de salida.

1.5. Lista de errores que el programa controla

En general son controlados todos aquellos errores conforme al fichero de entrada del programa, es decir; si se describe un tipo de personaje que no existe, si se crea el mapa con dimensiones negativas, si el personaje comienza en un turno que nunca va a llegar a ejecutarse.

1.6. Pruebas

Para realizar las pruebas del sistema se ha recurrido a la utilización de la librería J-unit, de java, que funciona de la siguiente manera:
Dentro del paquete Pruebas, creamos una nueva clase de tipo J-unit y darle un nombre, por ejemplo, RutasTest y ademas seleccionar la casilla *setUpBeforeClase*. Una vez hecho esto nos quedara algo así:

```java
package Pruebas;

import static org.junit.Assert.*;

import org.junit.BeforeClass;
import org.junit.Test;

public class RutaTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @Test
    public void test() {
        fail("Not yet implemented");
    }

}
```

Pues bien; si necesitamos algún objeto, como por ejemplo un mapa, fuera de ambos métodos deberemos especificar un atributo de tipo *static* y el tipo al que pertenece (*static* Mapa m). Dentro del método *setUpBeforeClase* podremos configurar los objetos que hayamos declarado.

Una vez realizado esto, por cada algoritmo de la clase que quiera ser probado, se deberá crear un método *public void "nombre del algoritmo"+test.*

Para comprobar si los resultados son correcto deberemos hacer uso de los *assert* (*assert+True o assert+False*).

```java
package Pruebas;
import static org.junit.Assert.*;
import java.io.IOException;
import org.junit.BeforeClass;
import org.junit.Test;
import Mapa_SuperHeroes.Arma;
import Puerta.HombrePuerta;

public class HombrePuertaTest {

    static HombrePuerta hp;

    @BeforeClass
    public static void inicio() throws IOException{
        Arma [] armasSalas={new Arma("Baston",28),new Arma("Anillo",1),new Arma("Escudo",5),new Arma("CampoMagnetico",4)};
        hp=new HombrePuerta();
        hp.configurar(armasSalas);

    }

    //Tras la configuración las armas deberían quedar así

//          B
//      A
//  null   null   C   null

//Siendo B=Baston el arma mas poderosa

    @Test
    public void getMayorArmaTest() {

        assertTrue(hp.getArmaMayor().getNombre()=="Baston");
        assertTrue(hp.getArmaMayor().getPoder()==28);
        assertFalse(hp.getArmaMayor().getPoder()<28);

    }
}
```

1.7. Historial de desarrollo

Explicar, a grandes rasgos, los objetivos de cada avance o modificación que se ha hecho y los problemas que se han encontrado en cada fase del desarrollo.
En una primera instancia nos pedían crear un prototipo del juego, siendo capaz el programa de generar las armas, distribuirlas, crear ciertos personajes, que se movieran, generar el hombre puerta…Todo esto con las estructuras proporcionadas por los profesores (Pila, Cola, Árbol y Lista). Además de esto se debía generar un fichero de salida, que pudiera ser comparado con los aportados por los profesores.
En la última etapa del desarrollo, las cosas cambiaron, lo que antes se hacía de forma manual (reparto de armas entre salas, creación de rutas…) ahora debía ser automático siguiendo las directrices indicadas por los profesores. También fue necesario añadir nuevas acciones a los personajes, como la interactuación con otros personajes que se encontraran en las salas.

Para concluir comentar que con respecto a las dificultades encontradas durante el desarrollo podemos destacar dos momentos:

1º Ordenación de las armas en los árboles y búsqueda de las armas de mayor poder dentro de los mismos.

Fue bastante complejo entender la forma en la que los arboles se ordenaban puesto que, en definitiva, los más importante era el poder de las armas y sin embargo quedaban ordenados por el nombre.
2º Reparto de armas por las salas.

Al igual que antes, realizar el algoritmo de reparto de armas entre las salas con mayor frecuencia de paso, me supuso más de un quebradero de cabeza.

MANUAL DE USUARIO

2.1. Introducción

El programa que se va a utilizar se basa en un juego que se realiza por si solo. Consta de un tablero, formado por salas, que los personajes deberán cruzar para llegar a la última sala, durante el camino esto irán recogiendo armas para ser más poderosos, y así enfrentarse al hombre que guarda la puerta.

2.2. Guía de instalación del proyecto

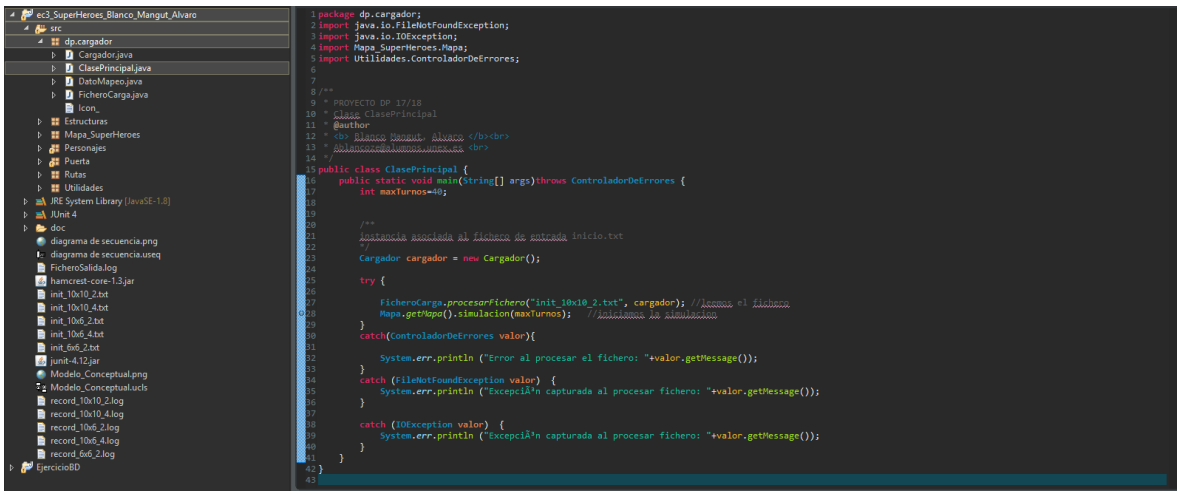
Sera necesario disponer de un entorno de desarrollo, que además soporte java, así como la versión número 7 del Jre y el Jdk.
La guía de instalación se utilizará como ejemplo el entorno de desarrollo *Eclipse.*

El programa se encuentra contenido en un archivo .zip comprimido, por lo tanto, será necesaria su descompresión antes de poder llevarlo al entorno de desarrollo. Una vez descomprimido, se ejecutará Eclipse y en la barra de navegación superior:

File>Import>General>Existin Projects into Workspace>Select root directory

Después de lo anterior seleccionamos la carpeta que contiene el proyecto y ya lo tendremos listo para ejecutarlo.

Una vez dentro, será necesario ir a la carpeta src, dentro de la misma ir al paquete dp.cargador y una vez dentro seleccionar clase principal.



En la línea 27 podremos cambiar el fichero con el que queremos que se inicie el sistema

2.3. Ejemplo de funcionamiento

El programa se ejecuta por si solo, genera un mapa con un tablero formado por salas las cuales contienen armas que los personajes irán recogiendo para ganar poder y poder enfrentarse al hombre puerta. Durante este proceso los Villanos y los Superhéroes podrán interactuar entre ellos siendo los primeros capturados y los segundos perdiendo armas, estos además avanzarán por las salas en función del turno y siguiendo una ruta predefinida. Llegados a la sala final deberán enfrentarse al hombre que gurda la puerta. El primero en vencerle será el ganador del juego y la simulación del mismo concluirá

VALORACIÓN FINAL DEL PROYECTO

Realmente puedo decir que he aprendido muchísimos conceptos y no solo de java, sino, en general de programación. El proyecto pasa por muchos cambios y hace que te plantees si realmente lo que tenías hecho te sirve para algo o tienes que quitarlo porque el usuario final no lo quiere o no lo necesita. Te pone en la tesitura de tener que hacer un proyecto software real, en el que encuentras trabas y barreras por donde mires

También he de decir que el proyecto de este año es ligeramente más difícil y a la vez mas satisfactorio, es la segunda vez que curso la asignatura. Te das cuenta de esto cuando, al fin, ves que se están repartiendo las armas en las salas como deben, después de haber estado horas programando con el debug y que no entiendes que has tocado, pero funciona… y ves que funciona. A mí, sinceramente, me gustan los retos y el proyecto de este año me ha supuesto un reto.

Comentar que, si he notado que la documentación, aportada por los profesores, no era todo lo clara que uno podría esperar y que por mi parte me ha generado muchísimas dudas que el año pasado no tuve, quizás todo esto venga debido al cambio de tipo de proyecto.