Práctica 2

Tecnología de la Programación I

Parte II: Extensiones

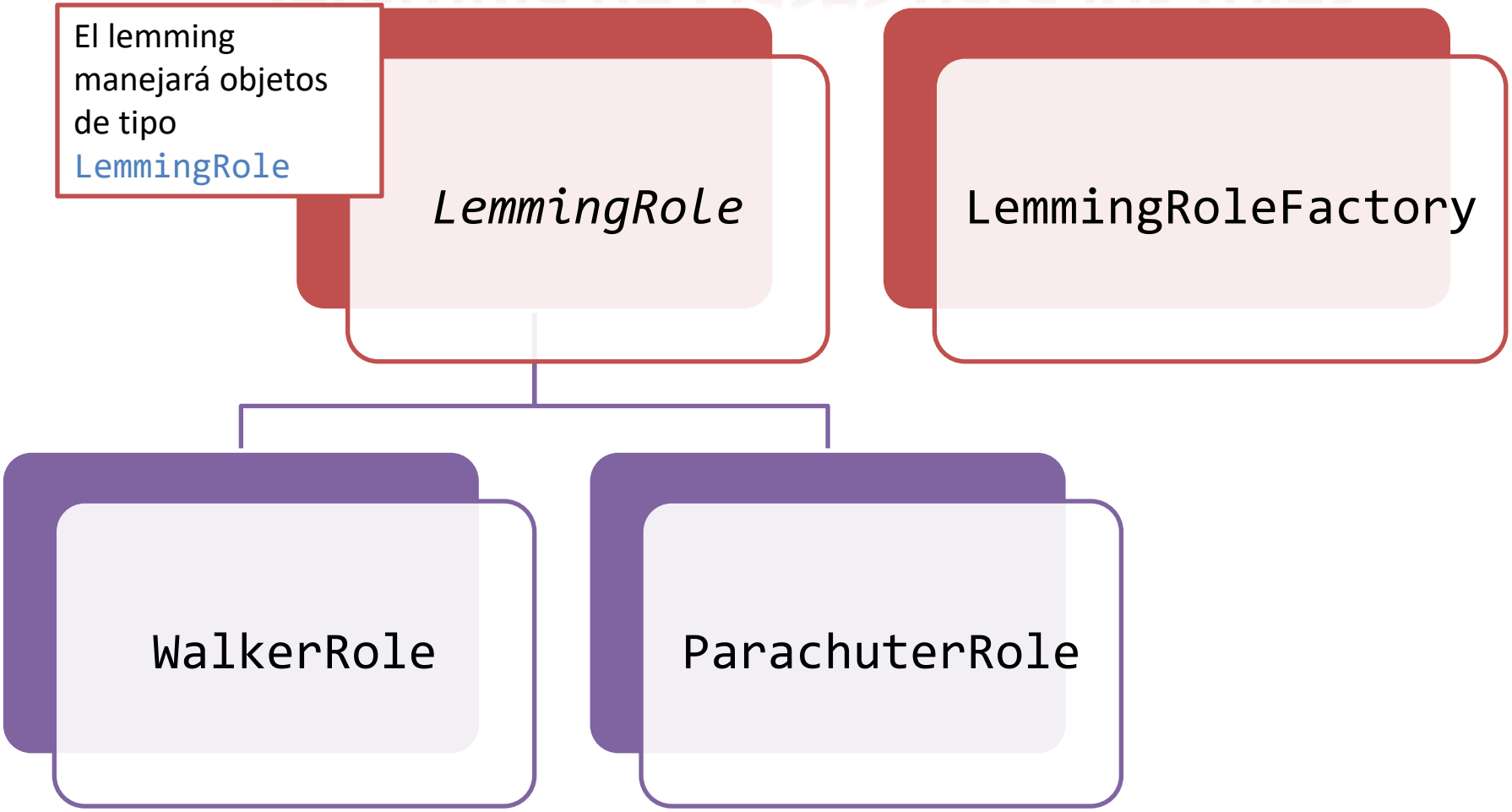
Extensiones

- Nuevo comando para cambiar el rol del lemming
- Interfaz *LemmingRole*
  - Nuevo rol *ParachuterRole*
- Factoría de roles: clase *LemmingRoleFactory*
  - Análogo al *CommandGenerator* de los comandos

- Generalizando las interacciones: técnica de “double-dispatch”
  - Interfaz *GameItem* con métodos para las interacciones
    - Implementada por *GameObject*
  - Nuevos métodos en *LemmingRole*
  - Nueva clase abstracta *AbstractRole* con implementación trivial de las interacciones
- Nuevos roles y objetos
  - Nuevo objeto *MetalWall*
  - Nuevo rol *DownCaver*

Jerarquía de clases para los roles

El lemming manejará objetos de tipo `LemmingRole`



Interfaz *LemmingRole*

```
public interface LemmingRole {

   public void start(Lemming lemming);
   public void play(Lemming lemming);
   public String getIcon(Lemming lemming);
      ……
}

public class WalkerRole implements LemmingRole {

   ……
}
```

Uso de *LemmingRole* en Lemming

```
public class Lemming extends GameObject {
    …
    private LemmingRole role;
    public Lemming (…….) {
        ……..
        role = new WalkerRole();
    }
    …….
    public void setRole(LemmingRole role) {  … }
    public void disableRole() {  … } //volver al WalkerRole
    public void update() {… role.play(this); …}
}
```

Polimorfismo

Vinculación dinámica

Patrón *Strategy*

- Mediante los roles estamos haciendo uso del patrón *Strategy*
- *Strategy* define una familia de algoritmos y los encapsula en clases separadas.
  - Se declaran los algoritmos en un interfaz (nuestro LemmingRole)
  - Las estrategias concretas implementan la interfaz (e.g., WalkerRole)
- Permite cambiar el algoritmo en tiempo de ejecución sin modificar el código cliente (en nuestro caso, basta un `setRole()`).
- Facilita la extensión de nuevas estrategias sin modificar código ya existente.

"Any problem in computer science can be solved with another layer of indirection, except of course for the problem of too many indirections".

David J. Wheeler (atribuida erróneamente a su estudiante, Bjarne Stroustrup)

Factoría de roles

Patrón *Factory*

- Patrón de creación que delega la instanciación de objetos a una clase especial, la “factoría“.

- Evitamos que una clase (cliente) cree directamente instancias de otras (proveedoras)
  - Es decir, la clase cliente deja de usar `new`
  - De lo contrario las clases cliente y proveedoras estarían estrechamente relacionadas (**acopladas**).

- Facilita la extensión del código, ya que permite ampliar la factoría sin tocar el código cliente.

- Nosotros usamos una versión simplificada y adaptada del patrón.

Factoría de roles

- Usamos la misma técnica que en `CommandGenerator`
  - Clase `LemmingRoleFactory` que mantiene una lista con los roles disponibles
  - Método estático

    `LemmingRoleFactory.parse(String input)`
  - Para parsear el `input` se recorre la lista de roles hasta que uno de ellos sea capaz de realizar el parseo

- Para que los roles puedan parsear será necesario añadir al interfaz `LemmingRole` algún nuevo método…

Nuevo comando: SetRoleCommand

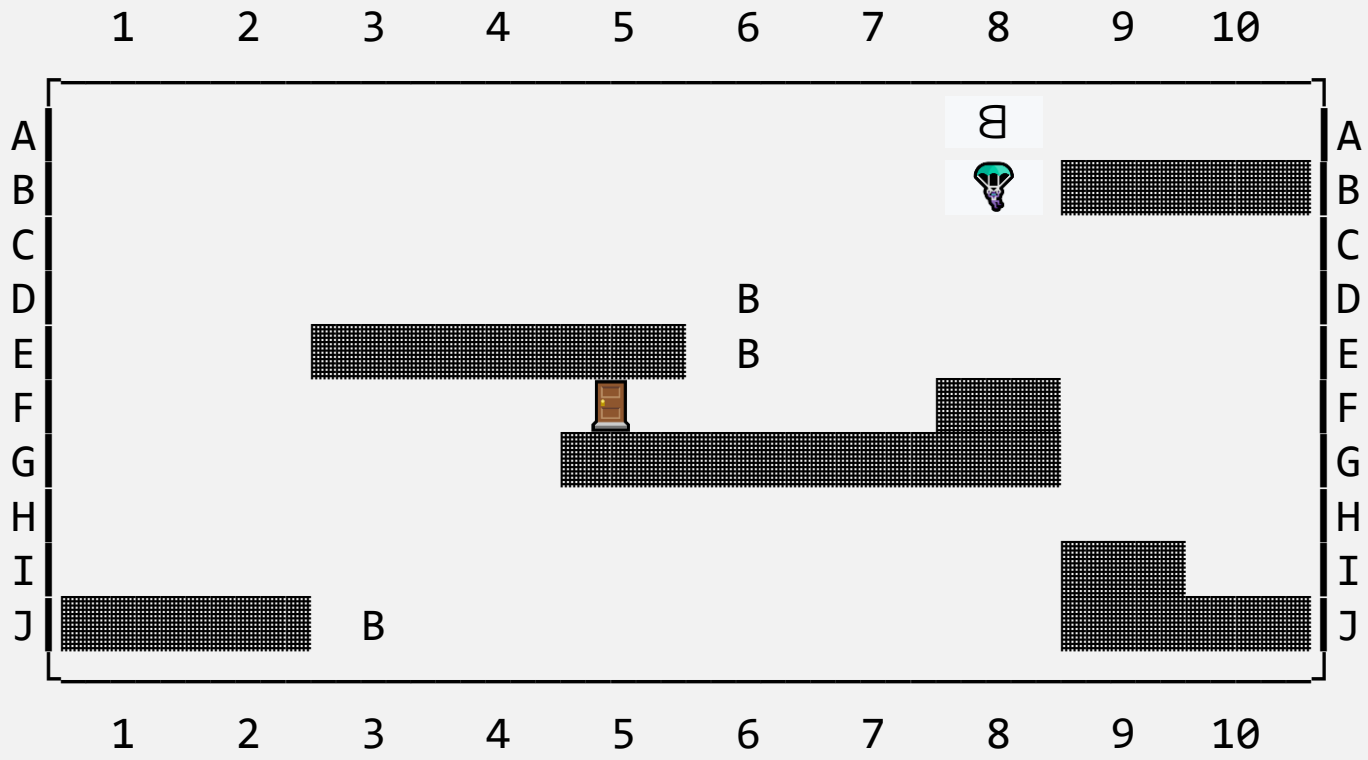
```
Command > h
[DEBUG] Executing: h

Available commands:
   [s]et[R]ole ROLE ROW COL: sets the lemming in position (ROW,COL) to role ROLE
      [P]arachuter: Lemming falls with a parachute
      [W]alker: Lemming that walks
   [n]one | "": user does not perform any action
   [r]eset: reset the game to initial configuration
   [h]elp: print this help message
   [e]xit: exits the game
```

Parseo y ejecución de *SetRoleCommand*

- Para parsear la entrada del usuario y obtener un SetRoleCommand es necesario:
  - Comprobar que la primera palabra se corresponde con el comando
  - Comprobar que la segunda palabra se puede parsear como rol usando la factoría de roles
  - Comprobar que el resto se puede parsear como una posición
- Para ejecutar el comando es necesario utilizar el rol y la posición obtenidos durante el parseo

SetRoleCommand en acción

Command >



Command > sr Parachuter A 8

Rol *Parachuter*

- Un lemming con el rol Parachuter cae con su **fuerza de caída siempre igual a 0**.
- El rol se desactiva en el momento en el que el lemming toca tierra.
- Si se intenta activar el rol cuando no está en el aire el rol se desactiva inmediatamente

Beneficios:

- Flexibilización de las interacciones entre objetos del juego
- Encapsulación de los métodos relacionados con las interacciones

Generalizado las interacciones: Interfaz GameItem de GameObject

Generalizando las interacciones

- Todos los objetos del juego deben tener la posibilidad de recibir interacciones de otros objetos
- Problema:
  - Los objetos se almacenan de modo abstracto, como *GameObjects* (¡y así queremos que sea!)
  - Pero el resultado de las interacciones depende de los tipos concretos:
    - Un excavador podrá cavar una pared normal pero no una de metal…

NECESITAMOS LA “MATRIZ” DE LAS INTERACCIONES

Generalizando las interacciones

- **Intento** de solución: en *GameObject* podríamos declarar

```
public abstract boolean receiveInteraction(GameObject other);
```

  - Cada subclase de *GameObject* puede implementarlo a su gusto (vinculación dinámica).
  - PERO: Java no permite elegir un método en función de los tipos dinámicos de los argumentos

Double-dispatch

- **Double-dispatch**: elección en tiempo de ejecución del método utilizado en función del tipo dinámico de dos objetos.
- Vinculación dinámica = single-dispatch.
- Java no permite (directamente) double-dispatch
- En la llamada a
  - `object.receiveInteraction(GameObject other)`

  se invoca al método implementado en el tipo dinámico de `object` y se ignora el tipo dinámico de `other`

  De `other` solo se usa su tipo estático para resolver sobrecarga, si la hubiese

Intento (fallido) de double-dispatch en Java

- Añadimos otro método a *GameObject*

```
public abstract boolean interactWith(GameObject obj)
```

```
@Override   //en la clase, por ejemplo, Wall
public boolean receiveInteraction(GameObject other){
      return other.interactWith(this);
}
```

En la llamada perdemos (se olvida) el tipo de **this (**`Wall`**)**

Solución: usar la sobrecarga para no olvidar el tipo de **this**

Double-dispatch en Java (¡ahora sí!)

- Necesitamos combinar vinculación dinámica y sobrecarga
- Definimos un interfaz para las interacciones (y más cosas)

```
public interface GameItem {

  public boolean receiveInteraction(GameItem other);

  public boolean interactWith(Lemming lemming);
  public boolean interactWith(Wall wall);
  public boolean interactWith(ExitDoor door);

  public boolean isSolid();
  public boolean isAlive();
  public boolean isExit();
  public boolean isOnPosition(Position pos);
}
```

```
public abstract class GameObject implements GameItem
```

Double-dispatch en Java (¡ahora sí!)

- Cada clase concreta tiene que tener su propia implementación de `receiveInteraction(GameItem other)`

```
public class Wall extends GameObject {

  ……
  public boolean receiveInteraction(GameItem other) {
        return other.interactWith(this);
  }

}
```

Tipo estático de **this**: `Wall`

Método invocado: `interactWith(Wall)`

- La resolución de la sobrecarga la realiza el compilador usando el tipo estático

¿Y si implementamos receiveInteraction() en GameObject?

```
public class GameObject implements GameItem {

 public boolean receiveInteraction(GameItem other) {
        return other.interactWith(this);
  }

}
```

Tipo estático de **this**:
GameObject

Método invocado:
interactWith(GameObject)

ERROR DE COMPILACIÓN

Implementación por defecto en GameObject

```
@Override
public boolean interactWith(Lemming lemming) { return false; }
@Override
public boolean interactWith(Wall wall) { return false; }
@Override
public boolean interactWith(ExitDoor door) { return false; }
```

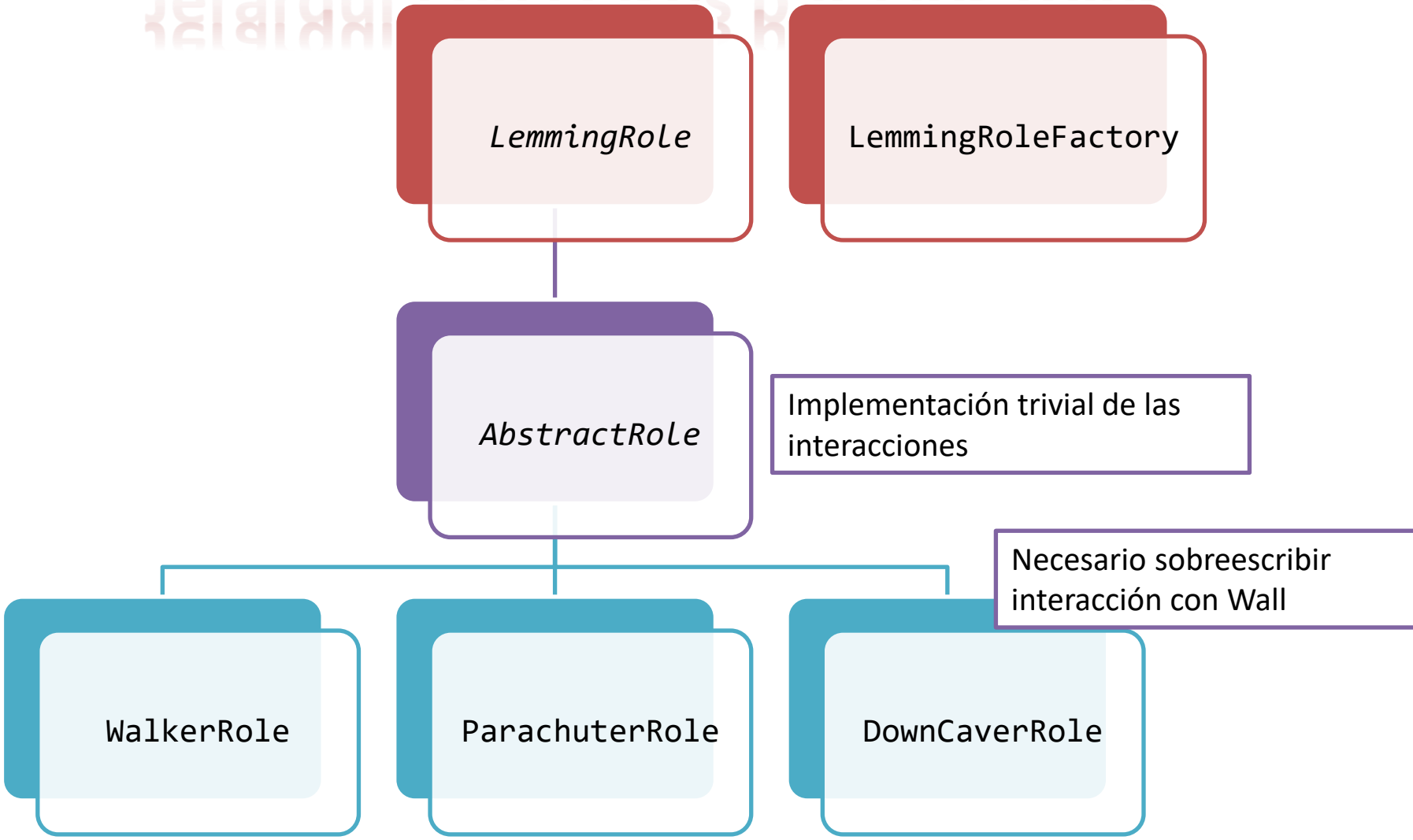
- Todos los objetos del juego deben tener la posibilidad de recibir una interacción
  - Si un objeto de juego recibe una interacción tendrá que sobrescribir el método como hemos visto
- Todos los objetos pueden interactuar con un Lemming/Wall/…
  - Por defecto, no lo hacen

Delegación de interacciones en los roles

- Nuevos métodos en *LemmingRole* para delegar las interacciones en los roles

```
public boolean receiveInteraction(GameItem other, Lemming owner);

public boolean interactWith(Lemming receiver, Lemming owner);
public boolean interactWith(Wall wall, Lemming owner);
public boolean interactWith(ExitDoor door, Lemming owner);
```

Jerarquía de clases para los roles



Generación y propagación de interacciones

- Las interacciones las generan (si lo necesitan) los GameObject al actualizarse
  - En el caso de los lemmings delegando (como casi siempre) en su rol
  - Para ello pueden hacer uso de un nuevo método de *GameWorld*

```
public boolean receiveInteractionsFrom(GameItem item)
```

- El *game* delega en el contenedor para generar las posibles interacciones de ese item con todos los *GameObject*.

Nuevo objeto *MetalWall*
Nuevo rol *DownCaver*

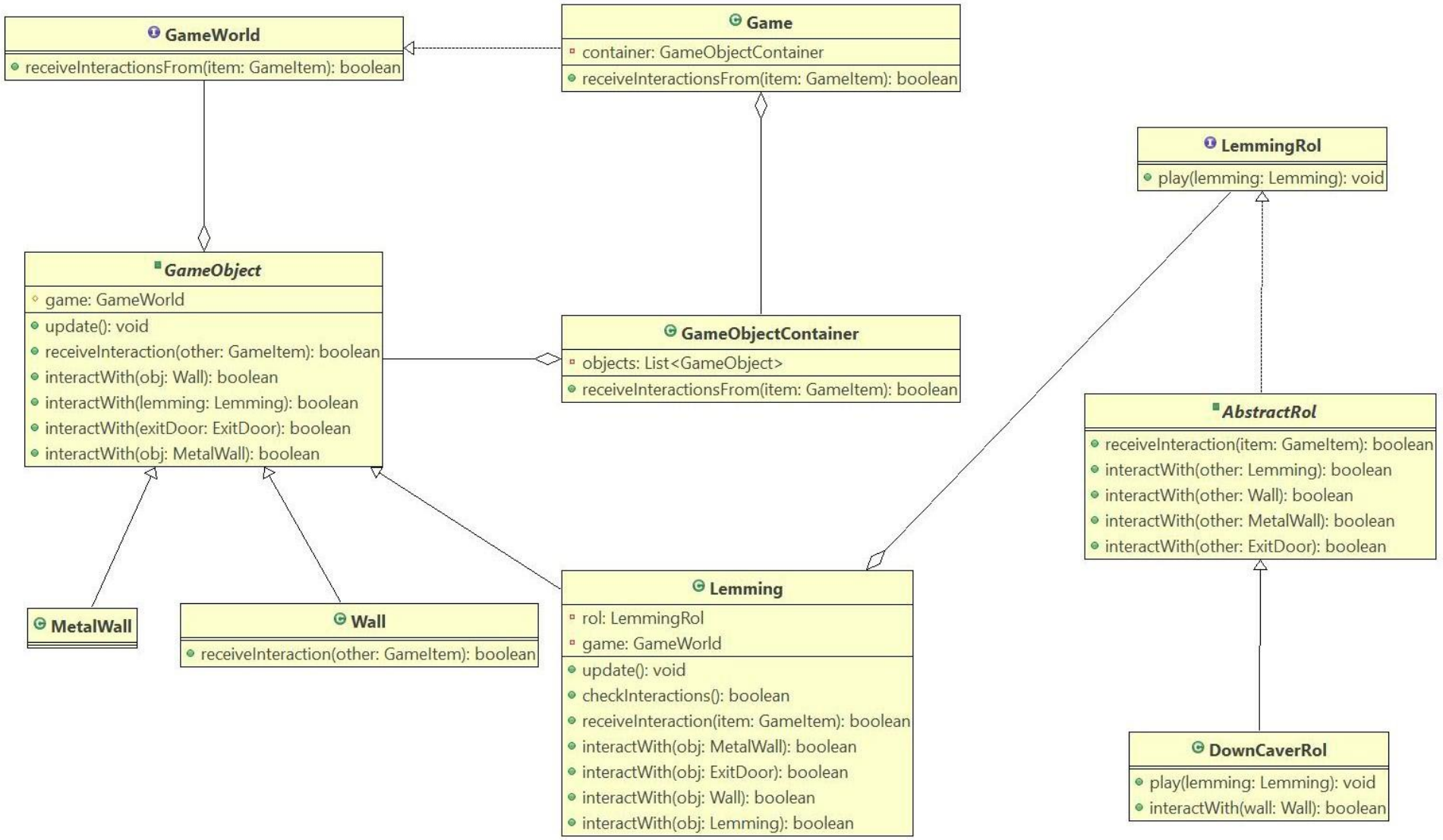
MetalWall

- Nuevo objeto similar a una Wall, salvo por el icono con el que se muestra (y las interacciones con los lemings cavadores…)

DownCaver

- Nuevo rol de lemmings que cava (destruye) la pared, *Wall*, sobre la que se encuentra el lemming
- No puede cavar una *MetalWall*
- Se desactiva el rol cuando se deja de cavar, bien por encontrarse con una *MetalWall* o por quedarse en el aire

Diagrama de clases: interacciones



Causas de suspenso directo en la práctica

- Mismas que en la P1
  - Falta o ruptura de encapsulación, métodos que devuelven listas, …
- Uso de `instanceof` o `getClass()`
  - Es incluso peor implementar un `instanceof` casero
    - Por ejemplo, haciendo que cada subclase de la clase `GameObject` contenga un conjunto de métodos `esX`, uno por cada subclase `X` de `GameObject`
    - El método `esX` de la clase `X` devuelve `true` y los demás métodos `esX` de la clase `X` devuelven `false`
  - Única posible excepción: implementación de `equals(Object)`