1. **Polimorfismo**: ¿Cuál es el mensaje polimórfico? ¿Qué objetos lo implementan? ¿Qué objeto se aprovecha de ello?

El polimorfismo se utiliza en varias ocasiones a lo largo de todo el código como se ve en la siguiente tabla:

|  |  |  |
| --- | --- | --- |
| Mensaje polimórfico | Objetos que lo implementan | Objeto que aprovecha el mensaje |
| objetivoDe(unEnemigo) | * Tímida * Cazadora * Tontita * Stalker * Exploradora | Fantasma |
| puedeSerComidoPor(unPersonaje) | * Fantasma * FireBall * Fruta * Menu(¿?) * Orbe * Pacman (?) * Estructura(¿?) | Pacman |
| encarcelar() | * Fantasma * Fruta * Pacman | mrMsPacman |
| mover() | * Pacman * Fantasmas | mrMsPacman |
| interactuarCon(unPersonaje) | * Fruta * Fantasmas * FireBall * Portal * Pacman * Orbe | Pacman |
| esTraspasable() | * Wall * Consumible * Portal | Posicionador |
| Iniciar() | * Fantasma * mrPacman * msPacman * Coin * Banana * Cherry * Pizza * Chilly * Heart | mrMsPacman |

La importancia del polimorfismo es que nos permite agregar objetos, de ser necesario, que entiendan un mensaje que se implemente en otro objeto, sin necesidad de tener que cambiar todo el código.

1. **Colecciones**: ¿Qué operaciones de colecciones se utilizan? ¿Usaron mensajes con y sin efecto? ¿Para qué?

**if** (unaListaDePersonajes.all({ unPersonaje => !unPersonaje.jugando() || !unPersonaje.tieneVidas() })) **self**.reiniciar() **else** unaListaDePersonajes.forEach({ unPersonaje => unPersonaje.mover() })

En este caso el principal mensaje que utiliza la colección es **all,** que es un mensaje sin efecto de orden superior, y se lo utiliza para evaluar si el juego debe comenzar (si el juego cumplió las condiciones para terminar o si no se había iniciado aún). En caso de que no se cumpla la condición, se utiliza otro mensajes sobre la colección, el cual es **forEach,** que provoca que surta el efecto del bloque de código, que es **mover,** y el efecto que produce, tal como indica su nombre, es que los personajes se muevan (el/ los Pacman **y el fantasma).**

Generalizando una operación utilizada varias veces:

(1 .. unaCantidad).forEach{ n => posicionables.add(**new** Algo())}

Como ya explicamos anteriormente la implementación del forEach, ahora plantearemos la del mensaje **add**, que es un mensaje con efecto, el cual en este caso lo utilizamos para añadir a la colección **posicionables** el elemento indicado.

Con una operación similar (la colección en lugar de *posicionables* se llama **muro**) se generó el tablero de juego.

unasPos.removeAllSuchThat({ unaPos => !**self**.esPosUsable(unaPos)})

**removeAllSuchThat** es un mensaje con efecto que en este caso lo usamos para que los fantasmas no puedan volver hacia atrás a una posición que utilizaron anteriormente, es decir, que solo puedan avanzar hacia adelante.

unaPos.allElements().isEmpty()

En este caso tanto **allElements** como **isEmpty** son mensajes sin efecto, que fueron utilizados para saber si una una posición del tablero estaba libre.

1. **Clases**: ¿Usan clases? ¿Por qué? ¿Dónde o cuándo se instancian los objetos?

Si se utilizan clases, como por ejemplo: Portal, Posicionable, Posicionado, Fantasma, FireBall, Consumible, Fruta, etc.

El utilizar clases nos permite que podamos agrupar aquellos objetos que de base entienden los mismos mensajes y variables (Ej: Los fantasmas en base pueden comer al pacman, se mueven de manera independiente, y no interactúan con las frutas), y de esta manera se evita repetir lógica, ya que cada objeto creado con cada clase tiene variables que son “independientes” de los demás objetos creados con esa misma clase.

1. **Herencia**: ¿Entre quiénes y por qué? ¿Qué comportamiento es común y cuál distinto?

msPacman

mrPacman

Posicionable

Estructura

MostradorPunts

MostradorCorazones

Portal

Wall

Fruta

Fantasma

FireBall

Orbe

Consumible

Pacman

Heart

Pizza

Chilly

Coins

Cherry

Banana

En el diagrama se puede ver un “árbol” horizontal que representa la herencia entre las clases.

Teniendo en cuenta que las subclases heredan los mensajes y variables de la superclase de su clase padre por transitividad, solo se especifican los comportamientos diferentes (respecto las respectivas clases padre).

Nivel 0: Todas clases tienen en común las variables **position** y **nombre**.

Nivel 1:

a. **Consumibles:** puntos – cárcel - esUtilizable() – esTraspasable()

b. **MostradorPuntos:** image() – emitir(mensaje) – iniciarParaEn(unJugador, unaPosicion)

c. **MostradorCorazones:** image() - iniciarParaEn(unJugador, unaPosicion(unJugador, unaPosicion)

d. **Estructura:** diubujar() – puedeSerComidoPor()

Nivel 2:

a.1: **Fireball:** image() – comestiblesEnAreaDE(unPersonaje) – puedeSerComidoPor(unPersonaje)- interactuarCon(unPersonaje)- aparecerConPor(unPersonaje, unTiempo)

a.2 **Pacman:** image() – imagenEstado() – imagenCerrado() – encarcelar() – tieneVidas()- configGamePad(tUp, tDown, tLeft, tRight) – incializar() – setearTanterosEn(unaPosicion) - interactuarCon(unPersonaje)- animar() – estado() – enfurecerPor(unTiempo) – sumarPuntos(unosPuntos) – ganarVida() – vidas() – puedeMatar() – posicionador() – puedeSerComidoPor(unPersonaje)- serEliminado() – aparecer () – mover()- emitirMensaje(mensaje) – toggleMortal()

a.3 **Fantasma:** image() - encarcelar() – jugando() – tieneVidas() – lastPos() – iniciar() – setarPuntaje() – aparecer() – objetivo() – puedeSerComidoPor(unPersonaje)- serComidoPor(unPersonaje) – interactuarCon(unPersonaje) – mover()

a.4 **Orbe:** image() – mostrarse() – puedeSerComidoPor(unPersonaje) – interactuarCon(unPersonaje)

a.5 **Fruta:** posicionador – cooldown - image() – jugando() – tieneVidas() – puedeSerComidoPor(unPersonaje) – efectoPara() – inicializar() – encarcelar() – serComidoPor(unPersonaje)- interactuarCon(unPersonaje) – aparecer()

d.1: **Portal:** salida – esTraspasable () – esUtilizable()- image() – anexarCon(otroPortal) interactuarCon(unPersonaje) – teletransportarA(unPersonaje)

d.2: **Wall:** image()- esUtlizable()- esTraspasable()

Nivel 3: Las clases de este nivel tienen todas en común que el método diferente al de su clase padre es **iniciar().**

La herencia se da porque permite generalizar métodos que son comunes a todos, y que se vayan creando subclases en base a la necesidad de usar métodos más específicos, pero que no todos los objetos de esa clase padre necesiten conocer.

1. **Composición**: ¿Qué objetos interactúan? ¿Dónde se delega? ¿Por qué no herencia?

**method** reproducir(nombre) {

elGame.sound("sounds/" + nombre).play()

}

Los objetos que interactúan son **sonido**, que es el objeto propio que utiliza el método que usa composición, y los objetos que pertenecen a **sounds(click, comer, explosión**, **gameOver, inicio, teleport)** que son los que van a ir variando a partir del parámetro **nombre** en dicho método.

**const** posicionador = posicionadorFruta

**const** posicionadorFruta = **new** Posicionador()

**method** aparecer() {

posicionador.randomPosPara(**self**)

Los objetos que interactúan son **Fruta**, que es la clase que utiliza ese método, la clase **Posicionador**, que es la clase padre de los objetos **PosicionadorPacman** y **PosicionadorFantasma**, que son los que interactúan con Fruta, por medio de la constante **posicionador**, ya que va creando objetos de esas clases.

No se usa herencia porque, pese a que en un punto se usa jerarquía de clases, realmente los objetos que están interactuando son Fruta con PosicionadorFantasma y PosicionadorPacman, que son los que van variando en el método(todos los objetos que se van creando a partir de esas clases), por lo que realmente no hay una jerarquía de clases, y no hay necesidad de usar herencia.