

# Mansión Zombie

Izan Buil López

26/09/25

## Índice

Introducción .....	3
Decisiones de diseño .....	4
Arquitectura general.....	5
Organización por paquetes.....	6
Gestión de datos y flujo del juego .....	7
Representación del estado de la partida .....	7
Flujo de juego .....	7
Mejoras en la representación de datos .....	7
Problemas y soluciones.....	9
Redundancia de clases / código.....	9
Complejidad en SearchAction.....	9
Resultados únicos vs. múltiples.....	9
Reflexión personal .....	10

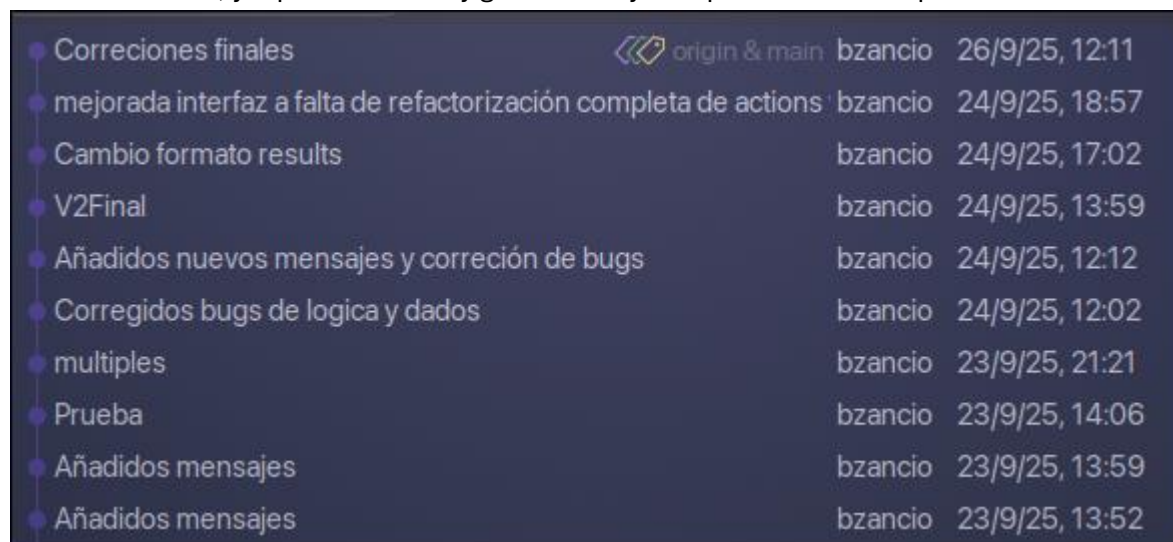
# Introducción

El programa a implementar consistía en un pequeño juego de rol, en el cual el objetivo era escapar de una mansión mediante la toma de decisiones y la gestión del riesgo. Inicialmente, el nivel necesario para realizar dicha tarea me pareció bajo; pero, al ser la idea interesante, decidí complicarlo y añadirle múltiples mejoras. No en lo jugable, ya que la funcionalidad que he entregado no es ni más ni menos la que se pedía en el enunciado, con excepción de alguna mejora en la representación de los datos de la partida. En mi opinión, lo que brilla de mi implementación es la separación lógico-visual del código, aplicando distintos patrones de diseño y características de Java relativamente modernas.

Como desarrollaré posteriormente en los próximos apartados del presente documento, a rasgos generales puedo señalar de mi proyecto unos objetivos claros, como son:

- **Escalabilidad:** es relativamente sencillo implementar nuevas funcionalidades, reutilizando código y tocando la estructura del proyecto lo menos posible.
- **Separación de responsabilidades:** cada clase es responsable de sus datos y de su comportamiento.
- **Claridad del código:** el código es fácilmente legible y entendible, incluso para alguien que lo ve por primera vez.

Para finalizar la introducción a mi trabajo, también quiero reconocer que mi proyecto no está exento de problemas. A veces el código es algo redundante y con muchas clases diferentes, aunque por lo general está justificado para garantizar los principios previamente establecidos. Podría haberlo hecho mucho más sencillo y funcionaría exactamente igual; pero quería experimentar y que este proyecto me ayudara de verdad a mejorar mis habilidades de elaboración de patrones de diseño y de Java en general. No exagero si digo que le he echado más de 50 horas, ya que ha sido muy gratificante y enriquecedor todo el proceso.

A screenshot of a commit history window from a code editor. It shows a list of commits with their messages, authors, and timestamps. The commits are listed in descending order of time.

● Correcciones finales	origin & main bzancio	26/9/25, 12:11
● mejorada interfaz a falta de refactorización completa de actions	bzancio	24/9/25, 18:57
● Cambio formato results	bzancio	24/9/25, 17:02
● V2Final	bzancio	24/9/25, 13:59
● Añadidos nuevos mensajes y corrección de bugs	bzancio	24/9/25, 12:12
● Corregidos bugs de logica y dados	bzancio	24/9/25, 12:02
● multiples	bzancio	23/9/25, 21:21
● Prueba	bzancio	23/9/25, 14:06
● Añadidos mensajes	bzancio	23/9/25, 13:59
● Añadidos mensajes	bzancio	23/9/25, 13:52

# Decisiones de diseño

En clase comentaste en repetidas ocasiones la importancia de este proyecto, ya que lo usaríamos en más ocasiones y, eventualmente, le añadiríamos una capa gráfica. Yo, siendo consciente de lo frustrante que es trabajar con un proyecto con una estructura endeble o con mala documentación, me decidí a investigar cuál era el mejor enfoque para realizarlo. Llegué a la conclusión de que la mejor solución era separar completamente (o lo máximo posible en su defecto) la parte gráfica de la lógica, ya que esto me ahorraría posibles quebraderos de cabeza en el futuro, cuando quisiera añadir algún tipo de interfaz con Java Swing, por ejemplo.

Como programador, uno de mis puntos débiles es la toma de decisiones a la hora de elegir una forma de afrontar un problema, sobre todo cuando hay multitud de soluciones disponibles. Así que, en bastantes ocasiones, he consultado a diferentes modelos del lenguaje para orientarme en cuanto a decisiones de este tipo. No obstante, todo el código lo he picado yo completamente a mano según mi criterio personal; por lo que puede haber errores. En general, creo que cuando me toque añadir la parte gráfica se me facilitará mucho el proceso gracias al trabajo ya hecho.

Esta separación de la parte visual y la parte lógica la he conseguido gracias al uso de varios patrones de diseño y herramientas que proporciona Java, tales como:

- **Clases *Record*:** me permiten definir estructuras de datos de forma concisa y clara.
- **Patrón *Factory*:** centraliza la creación de objetos y facilita la escalabilidad.
- **Patrón *Strategy*:** encapsula distintos comportamientos que puedo intercambiar fácilmente.
- **Comando simplificado:** organiza mejor las acciones principales del juego.

Por el camino se quedaron algunos patrones más complejos que, aunque interesantes, vi inviables de implementar. No quería perder el *scope* del proyecto, y para el tamaño de este no merecía la pena añadir más capas de abstracción.

# Arquitectura general

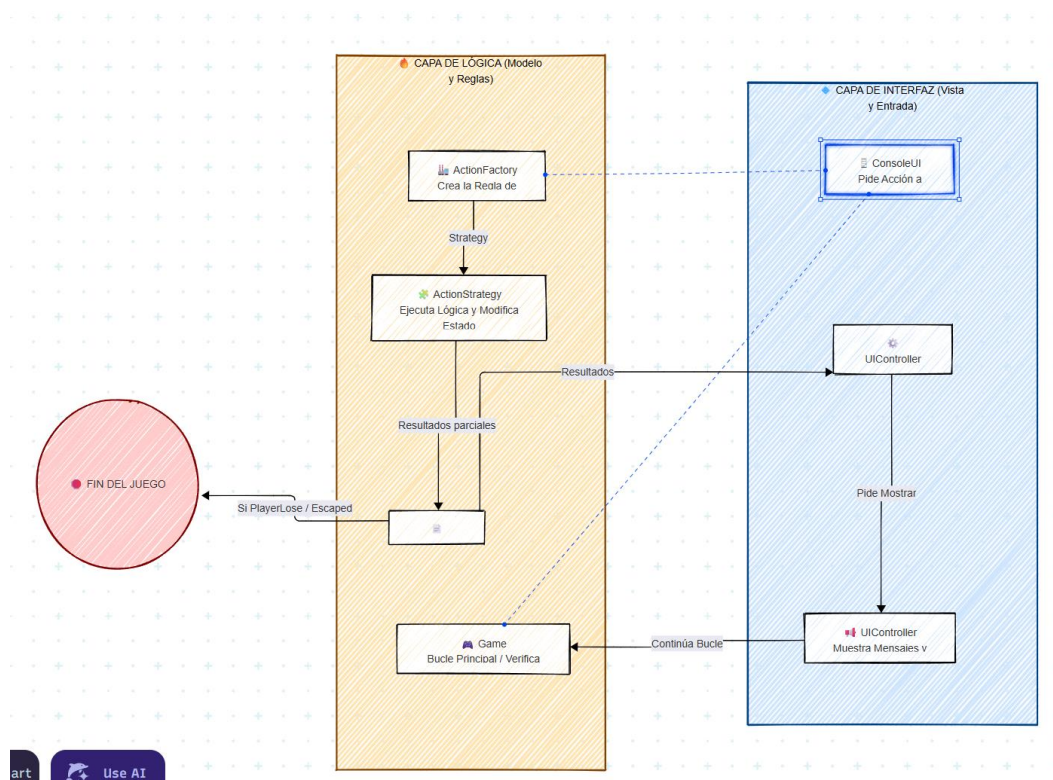
A grandes rasgos, mi programa funciona de la siguiente manera:

Cada acción que se puede tomar es una clase que tiene un método `isAvailable`, el cual devuelve un booleano en función de las condiciones necesarias para la disponibilidad de dicha acción. Si estas condiciones se cumplen, la acción se agrega a una lista de acciones que la interfaz lee y recita en orden, asociando cada una a un número. Este número se usa posteriormente para determinar cuál decisión ha tomado el jugador.

Una vez tomada la decisión, se crea el tipo de acción correspondiente mediante el patrón *Factory*, y en ella se definen las condiciones para su ejecución. Para garantizar la separación de lógica e interfaz, cada evento resultante de una acción (ya que puede haber varios, haciendo el sistema más flexible) se convierte en un objeto de la clase `ActionResult`. Estos objetos pueden ser de diferentes subtipos y se van acumulando como mini-eventos.

Después, el intérprete los lee y muestra lo sucedido de manera coherente. De este modo, cualquier intérprete recibirá un paquete de `ActionResult` que procesará en orden; en nuestro caso, el intérprete es `ConsoleUI`. Esto permite que, si en el futuro se añadiera otra interfaz gráfica, simplemente recibiría los mismos `ActionResult` sin necesidad de tocar la lógica del juego.

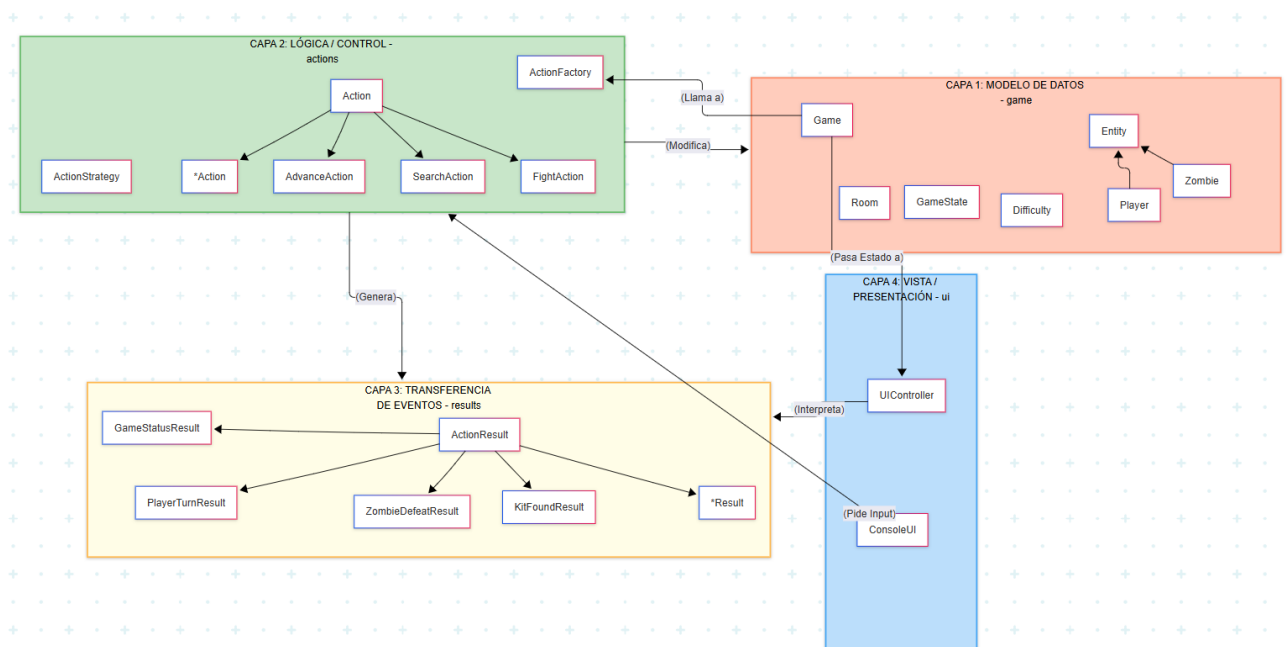
Gracias a este esquema ilustrativo, se puede apreciar cómo la lógica y la interfaz están completamente separados en todo momento, pudiendo sustituir unos módulos por otros de ser necesario.



## Organización por paquetes

Así mismo los paquetes de mi programa se dividen en varios:

- Actions: Define el comportamiento del jugador (**Controlador/Estrategia**). Ejecuta la lógica de las acciones y es la única capa que modifica el estado del *Modelo*.
- Game: Contiene el estado completo del juego (**Modelo**). Define las reglas básicas, la jerarquía de entidades y la gestión de habitaciones y vida.
- Results: Define el contrato de los eventos (**Datos/Comando**). Actúa como el puente de comunicación seguro entre la lógica (actions) y la vista (ui).
- Ui: Encargado de la Entrada/Salida (**Vista/Presentador**). ConsoleUI maneja la consola y el UIController traduce los ActionResult a métodos de la ConsoleUI.



# Gestión de datos y flujo del juego

## Representación del estado de la partida

El estado del juego se centraliza completamente en las clases del paquete `game`. La estructura de estas clases se diseña para representar el estado de manera fiel y aislada:

- **Game:** almacena el estado global (dificultad, número de habitación, `GameState`).
- **Player:** representa al jugador; hereda de `Entity`. Guarda HP, armas, protecciones y si tiene kit de curación. Métodos como `useKit()` y `takeDamage()` gestionan los cambios de estado.
- **Room:** contiene el estado local (intentos de búsqueda restantes, lista de zombies activos y su cantidad).
- **Zombie:** hereda de `Entity`. HP y ataque se ajustan dinámicamente según el número de habitación, para que la dificultad escale conforme el jugador avanza.

## Flujo de juego

El bucle principal del juego, implementado en `Game.java`, sigue un ciclo riguroso de estados:

1. **Estado Inicial:** se inicializa `Player` y `Room` (con un zombie inicial) tras seleccionar la dificultad.
2. **Obtención de Decisiones:** la clase `Game` invoca los métodos estáticos `isAvailable()` de las distintas `Strategies` (ej. `HealAction.isAvailable(this)`) para construir dinámicamente la lista de acciones válidas.
3. **Input:** se solicita al `ConsoleUI` que muestre las opciones y capture la elección del jugador.
4. **Ejecución de Lógica:** el `ActionFactory` crea la `ActionStrategy` seleccionada. Su método `execute()` modifica directamente el estado del modelo (por ejemplo, reduce el HP del zombie o cambia las protecciones del jugador) y genera la lista de `ActionResult`.
5. **Presentación:** `Game` pasa la `List<ActionResult>` al `UIController`, cuya única responsabilidad es traducir los objetos de datos del evento a mensajes mostrados en la consola.
6. **Nuevo Estado:** al finalizar cada ciclo, `Game` procesa el último `ActionResult` para verificar si se alcanza un estado final (`PlayerLoseResult` o `EscapedResult`) y termina el bucle si es necesario.

## Mejoras en la representación de datos

Una mejora clave es la creación de la clase `GameStatusResult`, que encapsula todo el estado de la partida (HP, armas, habitación actual, zombies activos) en un único objeto. Esto permite que la interfaz reciba una “foto” completa del estado del juego sin acceder directamente a los getters de `Game`, respetando la separación entre Modelo e Interfaz.

También me he creado un método que rodea cada salida de texto por una caja, es un detalle pequeño, pero creo que le aporta coherencia y enfoque a la interfaz, además también he añadido una función estilo clear para ir limpiando la pantalla y un efecto máquina de escribir en algunos mensajes

```
+-----+
| ESTADO ACTUAL |
| Habitación: 1/10 |
| Vida del jugador: 20/20 |
| Ataque: 4 + Armas: 0 |
| Protecciones: 0 |
| Botiquín disponible: false |
| Zombies activos: 1 |
| Búsquedas restantes: 3 |
+-----+
```

Presiona ENTER para continuar...



# Problemas y soluciones

## Redundancia de clases / código

- **Problema:** El proyecto terminó con unas veinte clases distintas de `ActionResult`. Podría haberse resuelto con una sola clase genérica con campos opcionales o mediante un gran `switch`.
- **Decisión:** Acepté esta redundancia y mantuve cada evento como una clase pequeña y específica (`PlayerTurnResult`, `ProtectionFoundResult`, etc.).
- **Motivo:** Esto mejoraba la claridad y la seguridad de tipos. Cada clase solo contenía los datos que necesitaba, evitando ambigüedades y facilitando un `cast` seguro en el `UIController`. Además, priorizaba la escalabilidad: añadir un nuevo evento se reducía a crear un nuevo `Record` y una línea adicional en el controlador.

## Complejidad en `SearchAction`

- **Problema:** Esta acción podía generar múltiples resultados con probabilidades distintas (hallar un kit, un arma, generar ruido, atraer zombies). El método `execute()` se volvía demasiado extenso y difícil de leer.
- **Solución:** Encapsulé la lógica de aparición de zombies en métodos auxiliares privados, como `noiseResult()`.
- **Motivo:** Esto redujo la complejidad del método principal, mantuvo la legibilidad y respetó el principio de responsabilidad única a nivel de clase.

## Resultados únicos vs. múltiples

- **Problema:** Al inicio, cada `ActionStrategy` devolvía un único `ActionResult`. Sin embargo, había acciones que generaban varios eventos obligatorios (ejemplo: `FightAction` con el turno del jugador y del zombie, o `SearchAction` con hasta tres eventos diferentes).
- **Decisión:** Cambié la firma de `execute()` para que devolviera `List<ActionResult>`.
- **Motivo:** Esto permitió manejar secuencias de consecuencias en el orden correcto sin que la lógica dependiera directamente del `UIController`. El sistema ganó flexibilidad y mantuvo intacta la separación entre capa.

```
public interface ActionStrategy { 7 usos 5 implementaciones  ⓘ bzancio
    List<ActionResult> execute(); 1 uso 5 implementaciones  ⓘ bzancio
}
```

## Reflexión personal

Este ha sido un proyecto sumamente enriquecedor. He aprendido patrones de diseño y características de Java que no conocía en absoluto, pero también he mejorado notablemente en la toma de decisiones, sobre todo al darme cuenta de que no todo salía como esperaba o de que el código no siempre quedaba tan limpio como pretendía.

Finalmente, me quedo con una sensación agri dulce: creo que mi programa es mejorable en algunos de los aspectos que ya he mencionado. Sin embargo, sabiendo que en el futuro lo retomaremos, me siento más tranquilo. Considero que es lo más ambicioso que he hecho en Java hasta ahora. Vengo de repetir un curso el año pasado, durante el cual perdí un poco la magia y la ilusión por programar. Además, una situación personal cambió, y el planteamiento de un proyecto tan interesante me ha ayudado a retomar el hábito de la programación, ya que el planteamiento inicial me resultó genuinamente motivador.

Es cierto que luego yo complico mucho el programa, pero nadie me quita lo aprendido. Por todo esto, concluyo mi defensa de este trabajo con la satisfacción de lo realizado y con ganas de implementar en el futuro una interfaz gráfica que complete el proyecto.

