

17/05/2024

ÍNDICE

*[Información del proyecto](#_Toc1295202451)* [3](#_Toc1295202451)

[Sistema de juego 3](#_Toc1725490990)

[Edad objetivo 3](#_Toc648004947)

[Clasificación por edades 3](#_Toc1688910377)

[Resumen de una página 3](#_Toc1320750317)

[Clasificación por edades 3](#_Toc1269054503)

[Ventaja competitiva 3](#_Toc559675979)

[Productos de la competencia 3](#_Toc591698431)

*[Decisiones de diseño](#_Toc764601399)* [3](#_Toc764601399)

[A nivel de técnico 3](#_Toc1513828207)

[ Patrón composite 3](#_Toc1294362897)

[A nivel artístico 3](#_Toc1328781797)

**Stop Pollution**(Document Design)

# *Información del proyecto*

## Sistema de juego

* Lore ipsum

## Edad objetivo

* Lore ipsum

## Clasificación por edades

* Lore ipsum

## Resumen de una página

* Este resumen debe ser un resumen de la historia del juego, pero centrado en sus mecánicas

## Clasificación por edades

* Lore ipsum

## Ventaja competitiva

* Lore ipsum

## Productos de la competencia

* Lore ipsum

# *Decisiones de diseño*

## A nivel de técnico

### Composición en lugar de herencia

* GDScript es algo peculiar en cuanto a la estructuración de su código. La decisión de utilizar el [patrón composite](https://refactoring.guru/es/design-patterns/composite) en lugar de utilizar la herencia se debe, precisamente, a dos problemáticas:

1. *La falta de interfaces:* GDScript cuenta con la herencia, pero no es un lenguaje en el que se pueda utilizar el recurso de la herencia múltiple. Java, por ejemplo, tampoco lo permite, pero se simula a través de la implementación de interfaces. GDScript, sin embargo, depende de la simulación de la herencia a través de la carga de los nodos de la escena.

PONER EJEMPLOS GRÁFICOS

1. *Enlazamiento entre nodos y scripts*: Godot no permite enlazar más de un script por nodo, así que perdemos la posibilidad de simular la herencia múltiple de esta forma.
2. *La herencia obligada*: En Godot, cada nodo se comporta como una clase en GDScript y cada script hereda, extiende, del nodo al que es enlazado. Por lo que ya se nos obliga a utilizar una herencia.

PONER EJEMPLOS GRÁFICOS

1. *El patrón DuckTyping*: Este patrón de diseño nos permite falsear la herencia. Con él, creamos una clase capaz de instanciar a otras y comprobar que en ellas se encuentran los métodos que heredarían. Es ideal para hacer test de las clases más básicas, pero del todo ineficaz cuando el código se vuelve más complejo debido a que el proyeto crece.

* Por todo ello, el patrón composite es perfecto para Godot.
* Imaginemos por un momento que tenemos una clase llamada *Interact*; como es obvio, se encarga de la interacción de los elementos del entorno con el jugador. *Interact* es una clase que cuenta con los métodos *interaction(), hit(), scroll()* y *move()*. Falseamos la herencia y la clase Player y obtiene los cuatro métodos, pero también necesitamos que los objetos del entorno, como un árbol o una roca, reciban dichos métodos; al menos el referido a la interacción con el jugador, pero ¿qué hacemos con el resto? Un árbol no se desplaza cuando es golpead, ni devuelve el golpe, ni se mueve por sí mismo.
* En la clase *Tree* sólo estariamos utilizando un método heredado y duplicando código con el resto. Por otro lado, la clase *Enemy* utiliza los cuatro métodos, por lo que es perfecto para la herencia.
* Entonces ¿cómo podemos hacer que *Tree* reciba únicamente uno de los métodos? Creando componentes y componiendo la escena referida a ese objeto con cada uno de ellos.
* Además, este patrón estructural nos permite componer distintos objetos intercalando los componentes que necesarios para cada funcionalidad.



## Call Down, Signal Up

* Debido a que se desea que cada escena se comporte como un nodo, todo su comportamiento se encapsula y cada escena debe funcionar por sí sola con la mínima interacción externa. Este paradigma, único y propio de Godot, pretende paliar varias problemáticas: la falta de herencia múltiple y la falta de interfaces.

## A nivel artístico